Software for Finite Element Methods and its Application to Nonvariational Problems

by

Lloyd Connellan

Thesis
Submitted to The University of Warwick
for the degree of
Doctor of Philosophy

Mathematics
The University of Warwick
September 2018
Contents

List of Tables 4
List of Figures 6
Acknowledgments 8
Declarations 9
Abstract 10

Chapter 1 Introduction 1
  1.1 A Basic Introduction to Finite Element Methods . . . . . . . . . . . . 1
  1.2 History of FEMs and Software Packages . . . . . . . . . . . . . . . . 3
  1.3 Nonvariational Problems . . . . . . . . . . . . . . . . . . . . . . . . 6
  1.4 Overview of Thesis . . . . . . . . . . . . . . . . . . . . . . . . . . . 10

Chapter 2 Dune-Fempy 13
  2.1 Finite Element Methods in DUNE-FEMPY . . . . . . . . . . . . . . . . 13
    2.1.1 Grids . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 16
    2.1.2 Spaces . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 18
    2.1.3 Grid Functions . . . . . . . . . . . . . . . . . . . . . . . . . . 19
    2.1.4 Schemes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 21
    2.1.5 Solving . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 23
  2.2 Alternate Solve Methods . . . . . . . . . . . . . . . . . . . . . . . . 26
  2.3 Model Generation . . . . . . . . . . . . . . . . . . . . . . . . . . . . 35
    2.3.1 Elliptic Models . . . . . . . . . . . . . . . . . . . . . . . . . . 36
    2.3.2 Integrands Models . . . . . . . . . . . . . . . . . . . . . . . . 38
    2.3.3 C++ Models . . . . . . . . . . . . . . . . . . . . . . . . . . . . 38
  2.4 Adaptive Mesh Refinement . . . . . . . . . . . . . . . . . . . . . . . 40
    2.4.1 Re-entrant Corner Problem . . . . . . . . . . . . . . . . . . . 41
# Appendix D  List of Dune-Python modules

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>D.1  Grids</td>
<td>154</td>
</tr>
<tr>
<td>D.2  Spaces</td>
<td>155</td>
</tr>
<tr>
<td>D.3  Grid Function</td>
<td>157</td>
</tr>
<tr>
<td>D.4  Schemes and Operators</td>
<td>159</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Runtimes for Forchheimer solve time .......................... 40
3.1 Interpolation error in the Laplace case for different norms ........ 100
3.2 Interpolation error with non-constant A for different norms ....... 101
3.3 Interpolation error for a non-smooth solution with different norms . 101
3.4 Table indicating which methods are symmetric .......................... 115
3.5 Levels of drop tolerance necessary for ILU ............................ 116
3.6 Variational method applied to Poisson’s equation ......................... 117
3.7 Nonvariational (DG) method applied to Poisson’s equation ............ 118
3.8 $L^2$ minimization method applied to Poisson’s equation ............... 118
3.9 $H^{-1}$ minimization method applied to Poisson’s equation ............. 118
3.10 $L^2$ minimization method with $H[u]$ .................................. 119
3.11 $H^{-1}$ minimization method with $H[u]$ .................................. 119
3.12 Table of EOCs for the Laplace example ................................. 119
3.13 Variational method applied to the AD equation ......................... 121
3.14 Table of EOCs for the AD problem .................................... 121
3.15 Table of EOCs for the nonD example .................................. 123
3.16 Table of EOCs for $k = 1$, for the nonD example ...................... 125
3.17 Table of EOCs for $k = 3$, for the nonD example ....................... 125
3.18 Condition numbers for the $L^2$ minimization method .................. 126
3.19 Condition numbers for the $H^{-1}$ minimization method ............... 126
3.20 Table of EOCs for the nonvariational p-Laplace ....................... 134
3.21 Table of EOCs for the variational p-Laplace ............................ 134
3.22 Table of EOCs for the simple nonlinear problem ....................... 135
3.23 Table of EOCs for the Monge-Ampère equation ........................ 136
D.1 Grids .......................................................... 155
D.2 Gridviews ....................................................... 155
D.3 Discrete Functions ............................................... 156
D.4 Spaces ................................................................. 156
D.5 Grid Functions ..................................................... 158
D.6 Solvers ............................................................... 160
List of Figures

2.1 Plot of a 2D grid for two different levels of refinement . . . . . . . . 17
2.2 Node maps of two Lagrange reference elements . . . . . . . . . . . . 18
2.3 The matplotlib plot of the initial function . . . . . . . . . . . . . . 20
2.4 Plot of solutions at each level of refinement . . . . . . . . . . . . . . 26
2.5 Plot of solution for Python-side Newton scheme . . . . . . . . . . . . 29
2.6 Plot of solution with Df operator . . . . . . . . . . . . . . . . . . . . 31
2.7 Plot of solution using PETSc . . . . . . . . . . . . . . . . . . . . . . 32
2.8 Plot of solution using PETSc and a Krylov method . . . . . . . . . . 33
2.9 Plot of solution using SNES . . . . . . . . . . . . . . . . . . . . . . . 34
2.10 The first three plots of the solution . . . . . . . . . . . . . . . . . . . 45
2.11 The second three plots of the solution . . . . . . . . . . . . . . . . . 45
2.12 The final three plots of the solution . . . . . . . . . . . . . . . . . . . 45
2.13 Zooming in on the re-entrant corner . . . . . . . . . . . . . . . . . . 46
2.14 Plot of the level function of the grid . . . . . . . . . . . . . . . . . . 46
2.15 The initial adapted grid and phase field . . . . . . . . . . . . . . . . 51
2.16 The grid, phase field and temperature after the final timestep . . . 52
2.17 The plot of the surface at three different timesteps . . . . . . . . . . 56
2.18 Comparison of the error over time for varying levels of refinement . 58
2.19 The domain, a cell split into three parts . . . . . . . . . . . . . . . . . 61
2.20 The initial plot of c and φ . . . . . . . . . . . . . . . . . . . . . . . . 70
2.21 The plot after the final timestep . . . . . . . . . . . . . . . . . . . . . 70
2.22 Comparison of time taken between the two calcRadius methods . . 73

3.1 Plot of $\|\Delta(u - I_h u)\|$ . . . . . . . . . . . . . . . . . . . . . 101
3.2 Plot of $\|\nabla N_h \Delta(u - I_h u)\|$ . . . . . . . . . . . . . . . . 101
3.3 Plots of $L^2$ errors for Poisson’s equation . . . . . . . . . . . . . . 120
3.4 Plots of $L^2$ EOCs for Poisson’s equation . . . . . . . . . . . . . . . 120
3.5 Plots of $H^1$ errors for Poisson’s equation . . . . . . . . . . . . . . . 120
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.6</td>
<td>Plots of $H^1$ EOCs for Poisson’s equation</td>
<td>120</td>
</tr>
<tr>
<td>3.7</td>
<td>Plots of $L^2$ errors for the AD problem</td>
<td>122</td>
</tr>
<tr>
<td>3.8</td>
<td>Plots of $L^2$ EOCs for the AD problem</td>
<td>122</td>
</tr>
<tr>
<td>3.9</td>
<td>Plots of $H^1$ errors for the AD problem</td>
<td>122</td>
</tr>
<tr>
<td>3.10</td>
<td>Plots of $H^1$ EOCs for the AD problem</td>
<td>122</td>
</tr>
<tr>
<td>3.11</td>
<td>Plots of $L^2$ errors for the nonD problem</td>
<td>123</td>
</tr>
<tr>
<td>3.12</td>
<td>Plots of $L^2$ EOCs for the nonD problem</td>
<td>123</td>
</tr>
<tr>
<td>3.13</td>
<td>Plots of $H^1$ errors for the nonD problem</td>
<td>124</td>
</tr>
<tr>
<td>3.14</td>
<td>Plots of $H^1$ EOCs for the nonD problem</td>
<td>124</td>
</tr>
<tr>
<td>3.15</td>
<td>Comparison of the condition numbers for different methods</td>
<td>127</td>
</tr>
<tr>
<td>3.16</td>
<td>Plot of the iteration count for the nonD problem</td>
<td>128</td>
</tr>
<tr>
<td>3.17</td>
<td>Plot of the total time taken for the nonD problem</td>
<td>129</td>
</tr>
</tbody>
</table>
Acknowledgments

This work has been supported by the Engineering and Physical Sciences Research Council (EPSRC) within the MASDOC DTC at the University of Warwick.

First of all I would like to thank my supervisor Andreas Dedner for his continual guidance in all things mathematical and computational throughout my PhD. Without his help, often going above and beyond what I expect, I am sure this thesis would not be possible.

Secondly I would like to thank Björn Stinner for always looking out for me and giving me advice through my time at MASDOC. To Matteo Icardi who provided assistance with the battery problem, I am also very grateful.

To my colleagues at MASDOC, in particular my friends Neil and Adam, I would like to say thanks for all the support and friendly conversations over the years. To all my friends that I have met online during this time, I would also like to offer my thanks for their company.

Finally I wish to thank my family for always being there for me, and giving me their love and support. I could not have done this without you.
Declarations

The work described in this thesis is the author’s own, conducted under the supervision of Andreas Dedner (University of Warwick), however we note that the software package DUNE-FEMPY showcased in chapter 2 is a collaborative project with Andreas Dedner, Martin Nolte (University of Freiburg), Robert Klöfkorn (International Research Institute of Stavanger), Matthew Collins (University of Warwick) and other developers. We also note that the derivation of the finite element Hessian in section 3.5 was previously carried out by Andreas Dedner and Tristan Pryer (University of Reading) in an unpublished paper (Dedner and Pryer [2013]). None of the material contained in the thesis has been used by the author in any previous publication or degree.
Abstract

We begin by introducing an extension to the software package DUNE (a C++ based toolbox for solving PDEs with the finite element method) which has the main objective of providing a Python user interface to it. First of all we explain how we have structured the interface and go into some detail about the components typical to a FEM. We then go on to demonstrate different features available in the context of worked examples. For instance, we consider the integration of different software packages such as PETSc and SciPy, as well as FEM features such as grid adaptivity, moving domains, and partitioned grids. Throughout this we highlight design decisions that are different to other similar packages and the reasoning behind them. We conclude by demonstrating how C++ code development can be integrated into the process and how that affects efficiency.

We go on to consider an application of this software to nonvariational PDEs. The key contribution of this section is the development of a new method for solving this class of problems based on minimization. We derive this method and provide results for existence and uniqueness and error convergence. We also compare this method to existing methods and highlight the advantages it has. We then derive a second aspect of this method which involves a finite element version of the Hessian. We combine these features and look at numerical results for linear nonvariational problems. We compare the new methods along with other existing methods using our software in terms of convergence rates and efficiency. Finally we take an experimental look at solving nonlinear nonvariational problems using the finite element Hessian, and an application to the Monge-Ampère equation.
Before talking about finite element methods (FEMs), it is only right that one first talks of the partial differential equations (PDEs) they look to solve. PDEs have existed as a mathematical model for all manner of physical phenomena for centuries, with equations describing how fluids flow, how heat transfers, and how sound waves propagate. Indeed capturing the essence of how the world works around us inherently requires complexity, meaning that in many cases the simpler ordinary differential equation is not enough. Yet with this complexity requires an added effort to solve them, and often finding an analytical solution to all but the simplest PDEs is a difficult task, and at times an impossible one. Thus in modern times we typically look to numerical solutions and computers to solve PDEs, the most common methods being the finite difference method, the finite volume method, and of course the finite element method.

To describe FEMs in an introductory sense, the general concept is to split up a problem’s domain into separate smaller components (finite elements), upon which it is much easier to approximate the solution on. By moving to a finite-dimensional version of the function space, one concretely solves a discretized solution on each individual element. These elements are then combined to give the whole picture of the problem.
Let us mathematically describe this method with a relatively simple example. Let \( \Omega \subset \mathbb{R}^d \) be our domain. Then \textbf{Poisson’s equation} is

\[
-\Delta u = f, \quad \text{in } \Omega,
\]
\[
u = 0, \quad \text{on } \partial \Omega.
\]

Whilst elementary, this equation sees use in many areas, including electrostatics and fluid mechanics. Now in FEMs, the usual procedure is to obtain the \textbf{weak form} of the PDE by multiplying by a test function \( v \) in a function space \( V \), and integrating by parts. This results in the following equation. We want \( u \in V \) such that

\[
\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} fv \, dx, \quad \forall v \in V.
\]

(1.1)

As this weak form is a common feature of FEMs, there exists a generalized form,

\[
a(u, v) = (f, v), \quad \forall v \in V,
\]

(1.2)

where \( a(\cdot, \cdot) \) is a coercive bilinear form on \( V \) and \( (\cdot, \cdot) \) is the \( L^2 \) inner product. In this case, the weak form of Poisson’s equation can be obtained by choosing \( V = H^1_0(\Omega) \) and \( a(\cdot, \cdot) \) in (1.2) to be the \( H^1_0(\Omega) \) inner product.

Following this, it is necessary to convert this equation into an algebraic system that can be solved elementwise, i.e. to discretize it. One aspect of this is dividing the domain \( \Omega \) into a mesh of polygonal shapes \( \{K_i\} \) called a \textit{triangulation}. Specifically, we derive an approximation \( \Omega_h \) such that

\[
\Omega \approx \Omega_h = \bigcup_{i=1}^{N} K_i.
\]

Here \( \Omega_h \) is dependent on \( h \) (the mesh size) which is defined by

\[
h_K := \max_{x, y \in K} |x - y| \quad \text{and} \quad h := \max_{K \in \Omega_h} h_K.
\]

Additionally, it is necessary to choose a discrete space \( V_h \) (typically a polynomial...
space) that approximates $V$. With this, we are able to write the discrete solution as a linear combination of basis functions, i.e. $u_h(x) = \sum_{i=1}^{N} u_i \varphi_i(x)$ (where $\{\varphi_1, \ldots, \varphi_N\}$ is a basis of $V_h$). We can also write the discrete counterpart to (1.2) as follows

$$a(u_h, v_h) = (f, v_h), \quad \forall v_h \in V_h,$$

(1.3)

Finding $u_h \in V_h$ is known as the Galerkin method. Consequently, we can rewrite (1.3) in terms of basis functions, due to the linearity of $a(\cdot, \cdot)$.

$$\sum_{j=1}^{N} u_j a(\varphi_j, \varphi_i) = (f, \varphi_i), \quad i = 1, \ldots, N.$$

From here, one can form an algebraic system of equations by defining a matrix $A$ with entries $A_{ij} = a(\varphi_i, \varphi_j)$ and a vector $b$ with entries $b_i = (f, \varphi_i)$. We can then solve the linear system

$$A u = b,$$

where $u = (u_i)_{i=1}^{N}$ are the degrees of freedom of $u_h$. This system of linear equations can then be solved via an appropriate algorithm, e.g. a conjugate gradient (CG) method or generalized minimal residual (GMRES) method.

All in all this procedure allows one to consistently solve PDEs in a numerical sense, provided they can be put into a weak form.

### 1.2 History of FEMs and Software Packages

Now that we have introduced what FEMs are in an elementary sense, let us expand upon their history and development.

Among the different numerical methods for solving partial differential equations, finite element methods are one of the most popular. They have been used for a broad range of engineering and scientific problems, with the first computational applications originating as early as Turner et al. [1956]. Over the years there has been an extensive amount of literature analysing FEMs in general and their uses (see e.g. Khoei [2015] and Babuška and Strouboulis [2001]).
Just as the development of the theory of FEMs has progressed over the years, the landscape of FEM software itself has undergone much change. As a multidisciplinary method involving many different techniques, the scope for which direction to develop features is very high. Even within the realm of standard FEMs there exist a multitude of different options available. For instance with regards to types of finite elements, if one considers conforming finite elements (i.e. where $V_h \subset V$) then one has possibilities such as the well-known Lagrange element, the $H(\text{div})$ conforming Brezzi-Douglas-Marini element used for instance for the elastic stress tensor, the $H(\text{curl})$ conforming Nédélec element used in electromagnetism, and so on. Then, provided one uses an appropriate penalty method, one can further expand this to have nonconforming elements such as the $H^2$ Hermite (cubic) elements, the Morley (quadratic) elements used for fourth order problems, and the $H^1$ Crouzeix-Raviart element used in Stokes flow\(^1\). In addition one can consider the mesh itself; one can have structured grids that are more computationally efficient or unstructured ones that allow for more flexibility. Furthermore one could have different shapes such as squares, triangles, cubes, pyramids, hexahedrons and so on. This is all without going into more complex forms of FEM such as hp-FEM (see [Melenk, 2002, §1.4.3]), spectral element methods (see [Karniadakis and Sherwin, 2013, §1.2.2]) and extended finite element methods (XFEM) (see [Fries, 2008, §2]).

All in all this diversity of choice has lead to the situation of numerous competing packages that offer slightly different flavours of FEM. One preventative measure to this has been the development of large modular software libraries that offer many optional extensions in one place, thus forgoing the need to install different packages for different problems. Such examples include DUNE ([Bastian et al. [2008]]), deal.II ([Alzetta et al. [2018, accepted]]), FreeFem++ ([Hecht [2012]]) and Elmer ([Lyly et al. [1999]]). These large packages are typically written in languages such as C++ and Fortran that are efficient for large-scale computations.

In recent years however there has been a trend towards packages that favour usability. Such packages look to lower the learning curve for new developers and

\(^1\)For a more complete list of types of elements, see Kirby et al. [2012]
non-computer focused researchers, allowing for more time to be spent productively solving problems. Additionally, higher level programming languages facilitate the use of rapid prototyping, i.e. allowing one to quickly construct new models and test their viability without having to write an intricate program. Python and MATLAB are both examples of commonly used languages that prioritize usability; in particular Python has risen to become one of the most popular programming languages of recent times (see e.g. Tio [2019]). Yet there are downsides to these languages from the standpoint of a researcher in mathematics or engineering; namely that they are not as efficient as their traditional counterparts in C, C++ and Java. Thus the goal of many new packages has been to unify an interface that combines aspects of being easier to pick up and use, without compromising the functionality and efficiency of traditional packages.

There exist many ways to go about tackling this problem. One strategy is to make use of more modern features of C++ (and other similar languages), such as auto types, range-based for loops and lambda functions, to increase usability. Yet arguably even the most user-friendly versions of these languages remain intimidating to programming novices, due to their core design elements that cannot be changed. A different approach is to use a language that attempts to unify usability and efficiency in one place. Julia (Bezanson et al. [2014]) is one example of such a language. The principal downside is the lack of popularity or wide-spread use of any such language in comparison to Python or C.

A third alternative, one growing in popularity, is to use two languages such as Python and C++ in the same package. The core idea behind this approach is to use a simplified interface attached to a back end with lower level code, which is typically achieved via the use of an automatic code generation tool like SWIG or Cython. This tying together of front end to back end does require additional code and maintenance of the interfacing between them, but its merit is in that it effectively combines the best of both worlds. In particular FEniCS (Alnaes et al. [2015]) and Firedrake (Rathgeber et al. [2017]) are examples of this kind of software.

One large component of both these packages is the use of Unified Form
Language (UFL) (Alnäes et al. [2013]), a domain-specific language (DSL) which allows one to write variational equations directly. For instance for equation (1.1) we have the following simple code.

Code Listing 1.1: Poisson’s equation in UFL

\begin{verbatim}
1 a = inner(grad(u), grad(v)) * dx
2 b = inner(f, v) * dx
\end{verbatim}

We do however note that the code generation-style approach used in FEniCS and Firedrake does come with inherent weaknesses. In particular this generated code is not suited to direct editing, so should a binding not exist for a feature on the Python side, editing these generated files to add the feature on the C++ side is not an option. Furthermore, user interactibility with the C++ interface is not prioritized, which means porting code over to C++ for efficiency reasons or the writing of additional features are not viable.

In the first chapter of this thesis we introduce DUNE-FEMPY, a DUNE module that is an extension to DUNE-PYTHON (Dedner and Nolte [2018]) specifically aimed at adding high-level FEM features based on the DUNE-FEM module (Dedner et al. [2010]) to DUNE. The aim of both of these packages is to bring the usability and speedier writing of code to DUNE and its large array of existing modules whilst preserving the features available to a C++ developer. In particular the structure and functionality is designed to be analogous in many ways to DUNE code, making it less difficult to port code to C++ if necessary. Additionally, attempts to increase usability have been made, such as library caching to reduce the runtime of repeated computations, and integration with modern C++11/C++14 via pybind11 (see Jakob et al. [2017]) to interface between C++ and Python.

1.3 Nonvariational Problems

Continuing onwards, there exists another reason, besides the potential for optimization, for maintaining a similar structure to DUNE and other traditional C++ programming. Namely that is to facilitate the extensibility of code. In DUNE-FEMPY,
additional C++ code can be simply added to the interface via the use of DUNE modules and pybind11 functionality (a process that is explained in-depth in Dedner and Nolte [2018]). Considering that many interesting research topics by nature are nontrivial, it is crucial to be able to cater for problems that do not necessarily fit into the neat interface provided by many Python-style software packages.

Having said this, we note that there is a large range of problems that fit into the variational framework, and by extension the myriad of numerical software available for solving them. Indeed, since such a large variety of partial differential equations can be put into variational (or weak) form, regardless of complexities such as nonlinearity, it is usually not required to go beyond this scope.

There are however PDEs for which it is ill-advised, or sometimes impossible to put into a variational form. In particular, chapter 3 of this thesis looks at the class of PDEs that take the form

\[-A : D^2 u = f.\]

Here \(D^2 u\) is the Hessian of \(u\), \(f\) is a prescribed function and \(A\) is a matrix.

In the case that the matrix \(A\) is differentiable, we note that this has an obvious equivalence with standard variational methods. For instance in the case where \(A\) is the identity matrix, the above equation simply equates to Poisson’s equation. In such cases the above is equivalent to its variational sibling.

\[-\nabla \cdot (A \nabla u) + (\nabla \cdot A) \nabla u = f.\]

However we note that because of the existence of the \(DA\) term, this cannot be done in the case where \(A\) is not differentiable. In fact even in cases where the derivatives are close to zero, the PDE becomes advection dominated, making it probably unsuited for conforming FEMs. Because of this possibility, in general PDEs of the above form are classed as nonvariational.

We also note that this linear case can be extended to a nonlinear version that
takes the general form
\[ F(D^2u) = f(x, u, Du). \]

In this case \( F \) can be any kind of function acting on the second derivatives of \( u \), which increases the scope further. In particular such nonvariational problems occur in a variety of different contexts, for instance the Monge-Ampère equation (see e.g. Gutiérrez [2001]) and the Hamilton-Jacobi-Bellman equations (which have many applications such as in economics (Cao and Wan [2009]) and engineering (Ioslovich et al. [2009])); a review can be found in Katzourakis and Pryer [2018]).

The Monge-Ampère equation especially has many applications. The Dirichlet version (in a domain \( \Omega \)) takes the general form
\[
\det(D^2u) = f(x, u, Du), \quad \text{in } \Omega,
\]
\[ u = 0, \quad \text{on } \partial \Omega. \]

The most well-known application of this is the problem of prescribed Gauss curvature on a convex domain (Trudinger and Urbas [1983], Urbas [2004]). This is a specific example of the Monge-Ampère equation which takes the form
\[
\det(D^2u) = K(x)(1 + |Du|^2)^{(n+2)/2}.
\]

Another application is the mass-transfer problem (Benamou and Brenier [2000], Evans [1997]). This originates from the Monge-Kantorovich equation, which describes the transfer of mass from one area to another. This is realized via density functions \( \rho_0 \) and \( \rho_T \), and a map \( M \) between them. For a smooth one-to-one map this reduces to
\[
\det(\nabla M(x)) \rho_T(M(x)) = \rho_0(x)
\]
It is then possible to prove that for some convex function \( \Phi(x) \) that \( M(x) = \nabla \Phi(x) \), which once again returns the Monge-Ampère equation.

Furthermore more recently there has been an application to r-adaptivity on the sphere in McRae et al. [2016], i.e. moving the mesh points of a numerical
grid by solving the Monge-Ampère equation. Another recent work (Ożanski [2015]) demonstrates that it can also be applied to the Navier-Stokes equations.

On the whole however, whilst due consideration has been given to specific examples of nonvariational problems, as a class of equations themselves they have not been studied extensively. In particular numerical methods that target this problem are relatively few. Historically speaking, the first numerical methods developed for tackling nonvariational problems were finite difference methods (FDMs). For instance Oberman [2008] and Loeper and Rapetti [2005] studied the Monge-Ampère equation with such an approach. The most likely reason for the popularity of FDMs compared to FEMs for nonvariational problems is due to their compatibility with viscosity solutions, which are a natural type of solution for nonvariational problems. However that is not to say there is no disadvantage to FDMs, for in particular one is only able to consider structured meshes. By considering the problem from a FEM perspective, we open up the possibility of unstructured grids, and other useful tools such as grid adaptivity.

One of the first papers to consider a finite element approach to nonvariational models was Lakkis and Pryer [2010], where the concept of a finite element Hessian $H[u]$ was first introduced. This form comes from applying the distributional equation for the Hessian, i.e.

$$
\int_\Omega D^2 w \varphi \, dx = - \int_\Omega \nabla u \otimes \nabla \varphi \, dx + \int_{\partial \Omega} \nabla u \otimes n \varphi \, ds.
$$

This method has later been applied to nonlinear problems in Lakkis and Pryer [2012], and then developed into a discontinuous Galerkin (DG) method in Dedner and Pryer [2013]. Furthermore, we note that another DG finite element method was proposed around the same time in I. Smears [2013], which uses an hp-FEM, and later on an approach using a discrete version of the Hessian similar to the above was derived in Wang and Wang.

The problem has continued to see development from a FEM context. In particular a more recent paper Mu and Ye [2017] uses a symmetrised discretization
of \(-A : D^2 u = f\).

\[
\int_{\Omega} (A : D^2 u_h)(A : D^2 \varphi_h) \, dx + s(u_h, \varphi_h) = \int_{\Omega} f A : D^2 \varphi_h \, dx, \quad \forall \varphi_h \in V_h,
\]

where \(s(\cdot, \cdot)\) is a stabilization term. One singular property of this method is that the analysis is made easier by its inherent symmetry, which allows for less assumptions to be made of the regularity of the problem. Nonetheless the fourth-order nature of the method causes it to be less efficient numerically.

One of the main aims of chapter 3 is the development of a method that combines the symmetric properties of the above method with the numerical efficiency of other methods, and the use of the finite element Hessian. We also note that in section 3.2 we will present a more in-depth look at the methods from the literature and how they tie in with the method developed in this thesis.

1.4 Overview of Thesis

Let us provide an overview of the chapters of this thesis. On the whole it is divided into two main parts, each of which is planned to become a paper in the future.

In chapter 2 we provide an overview of the software package DUNE-FEMPY, the features it provides, and a discussion of the design decisions. We begin in section 2.1 by introducing the interface for a simple FEM step-by-step, where we go through the process for solving a nonlinear parabolic PDE, the Forchheimer equation. Through this section we detail each component of the FEM and why they are considered necessary. Following this, in section 2.2 we consider different ways we can solve the PDE and in doing so demonstrate how the Python interface can be fully taken advantage of. In 2.3 we instead focus on an aspect which involves the C++ back end, by considering different ways of generating the model, and how flexibility has been provided for C++ programmers. For a change of pace, the remaining sections of the chapter look at additional features in the context of more complex examples. Beginning with section 2.4, we look at two examples that use adaptive mesh refinement. The first considers a non-standard domain that requires
more precision around a certain point, and the second features a time-dependent problem that requires the grid adaption to change over time. In section 2.5 we then consider a mean curvature example, i.e. an example where the surface evolves over time due to a smoothing condition. For the last example, in section 2.6 we look at a model of a Li-ion battery where the domain is divided into three separate regions. Finally we discuss the comparison to C++ code in section 2.7, and how virtualization has been taken into consideration in section 2.8.

For chapter 3, the second part of this thesis, we consider nonvariational problems and their discretization in DUNE-FEMPY. We first concretely define the problem and corresponding notation in section 3.1. Then in section 3.2 we review existing methods in the literature, and compare them along with a new method proposed in this paper. We begin the analysis of this method in section 3.3, where we formally introduce this new method based on minimization, show existence and uniqueness, and derive a saddle point formulation. We then proceed in section 3.4.1 to provide error analysis for this method in terms of a bound between the solution and its discrete approximation. We also demonstrate that this error estimate may be suboptimal compared to the empirical results in part 3.4.2. We continue to consider alterations to the numerical implementation in section 3.5, where we derive a numerical version of the Hessian, which we will use to improve the previous method further. Following from this analysis, in section 3.6 we proceed to present the numerical implementation of the previous methods, implemented in DUNE-FEMPY. We compare them in the context of the linear case, considering convergence rates and efficiency of the approaches. Finally we will look at a purely numerical implementation of the nonlinear case in section 3.7, in which we look to solve the Monge-Ampère method and other nonlinear problems.

To summarize things, section 4 reiterates what has been achieved, and the future directions available to continue on from this project.

We note that there are two key findings to this thesis, the first of which is the contributions to developing DUNE-FEMPY\(^2\), a tool for writing and developing

finite element methods with a Python interface, based on the well-known open
source software package DUNE. DUNE-FEmPY is the first attempt to bring Python
scripting to DUNE, is aimed at maintaining the flexibility of the DUNE module, and
is already used in other projects. The second key finding is the minimization method
for solving nonvariational problems, and the application of the finite element Hessian
to said method and later on to nonlinear problems.

In particular, due to the collaborative nature of the project, I also will also
emphasize the following contributions which are uniquely my own. To begin with,
in DUNE-FEmPY I created the initial framework for the UFL to C++ conversion
for models. Over the course of the project I have added to the underlying infra-
structure, most notably to code involving grid functions and models. The code for
the Forchheimer example (shown in section 2.1), the battery example (shown in sec-
tion 2.6) and a Navier-Stokes example (not shown here) were written by me (other
examples shown were written by others and adapted to this thesis). Otherwise, all
written parts of chapter 2 (except for section 2.8) were written by me. For the
nonvariational section, the code contained within the DUNE-FEmNV module\(^3\) is al-
most all my own, with the Monge-Ampère code and original Finite Element Hessian
computation written by my supervisor. In terms of the analysis, the minimization
method posed in sections 3.3 and 3.4 and the experiments in sections 3.6 and 3.7
were done by me with help from my supervisor.

\(^3\)Publically available at https://gitlab.dune-project.org/lloyd.connellan/dune-femnv.
Chapter 2

Dune-Fempy

2.1 Finite Element Methods in Dune-Fempy

When designing any software package, a natural challenge that arises is trying to make the user interface as simple and easy to use as possible. At the same time however, we also want to create an interface that retains all the functionality we need.

In the context of finite element methods, this leads to the question of what the minimal functional structure for a FEM looks like. In order to try to address this question, we will first outline from a mathematical standpoint the general structure we have in mind for a FEM.

To begin with, the original problem we typically want to apply a finite element method to is a continuous PDE in some infinite-dimensional space $V$. First let $\Omega \subset \mathbb{R}^d$ be a polygonal domain for our problem. We then choose a conforming finite element space $V_h = \{ \varphi_h : \Omega \rightarrow \mathbb{R}^r \} \subset V$, where $\dim V_h = N$. This involves choosing a basis for $V_h$, which can vary depending on the problem, but typically involves piecewise polynomial functions.

Next the variational (or weak) form of the equation is defined. For the purpose of illustration, let us assume to start with we have a parabolic PDE of the
Following general form.

\[ \partial_t u + L[u] = f(x), \quad \text{in } \Omega \times [0, T], \]

\[ u(x, 0) = u^0(x), \quad \text{in } \Omega, \]

\[ D \nabla u \cdot n = g(x), \quad \text{on } \partial \Omega \times [0, T], \]

(2.1)

where the elliptic operator \( L \) is defined as

\[ L[u] := -\nabla \cdot D(x, u, \nabla u) \nabla u + m(x, u, \nabla u), \]

(2.2)

and where \( u^0 \) and \( g \) are the initial and boundary conditions and \( n \) is the outward pointing normal. We note that we are only considering Neumann boundary conditions here for simplicity, although Dirichlet boundary conditions are also a possibility.

To obtain the discrete form, we begin by discretizing the PDE in time. This results in the following method: given \( u^0 \), for \( n \in \mathbb{N}_0 \), find \( u^{n+1} \in V_h \) such that

\[ \frac{u^{n+1} - u^n}{\Delta t} + L_I[u^{n+1}] + L_E[u^n] = f(x, t^n), \]

(2.3)

where \( \Delta t \) is the time step, and \( L_I \) and \( L_E \) are the implicit and explicit parts of \( L \), defined using (2.2) as

\[ L_I[u] = -\nabla \cdot D_I(x, u, \nabla u) \nabla u + m_I(x, u, \nabla u), \]

\[ L_E[u] = -\nabla \cdot D_E(x, u, \nabla u) \nabla u + m_E(x, u, \nabla u), \]

and \( D_I + D_E = D, \ m_I + m_E = m. \)

The variational form is then obtained from equation (2.3) by multiplying by a test function \( \varphi \in V_h \) and integrating by parts.

\[ \int_{\Omega} \frac{u^{n+1} - u^n}{\Delta t} \varphi + (D_I \nabla u^{n+1} + D_E \nabla u^n) \cdot \nabla \varphi + (m_I + m_E) \varphi \, dx = \int_{\Omega} f \varphi \, dx + \int_{\partial \Omega} g \varphi \, ds, \quad \varphi \in V_h. \]

(2.4)
We note that in terms of the actual solving of this form, there exists potential variation in terms of the solver used and possible nonlinearity of the problem. We also note that this is a simple scheme for demonstration, and more complex examples involving higher order schemes or nonconforming spaces can be easily implemented along the same lines.

With this general form in mind, in DUNE-FEMPY we have designed the structure to take as similar a style as possible, which results in the following breakdown of parts.

- **2.1.1 Grid.** The computational domain \( \Omega \) the problem is set in.
- **2.1.2 Space.** The finite element space \( V_h \) and type of basis functions.
- **2.1.3 Grid functions.** Functions defined on the grid that store the solution \( u_h \) and other variables.
- **2.1.4 Scheme.** The weak form of the equation, its boundary conditions, and method for solving.
- **2.1.5 Solving.** The actual solving process and data output.

We note that there exist even further simplifications that can be made in terms of this design choice; for instance a FEM could be distilled to simply choosing a weak form (an operator) and a grid, and having all other things set to sensible defaults. Additionally the code itself used to represent these methods could be simplified to a large degree depending on the aim of the software.

Ultimately as a FEM package aimed more at extensibility and for researchers who are willing to commit to some degree of programming, we have opted for more complexity in some cases at the expense of this simplicity. In general this is quite a nuanced design decision that must be made without a clear *right* answer.

With that in mind, for the remainder of this section we shall demonstrate in more detail how each of these concepts are implemented in the context of a worked example, the FEM applied to the Forchheimer equation: a scalar, nonlinear
parabolic equation derived in Kieu [2015]. A full derivation of this equation is described in appendix B, but for the following the final form suffices.

\[
\int_{\Omega} \frac{1}{\Delta t}(u^{n+1} - u^n)\varphi + \frac{1}{2} K(\nabla u^{n+1})\nabla u^{n+1} \cdot \nabla \varphi
+ \frac{1}{2} K(\nabla u^n)\nabla u^n \cdot \nabla \varphi \, dx = \int_{\Omega} f \varphi \, dx + \int_{\partial \Omega} g \varphi \, ds, \quad \varphi \in V_h, 
\]

(2.5)

where \(K(\nabla u)\) is a scalar function. We note that this corresponds to taking \(D_E = D_I = \frac{1}{2} K(\nabla u)I\) (where \(I\) is the identity matrix) and \(m_E = m_I = 0\) in equation (2.4).

### 2.1.1 Grids

The first aspect of FEMs that we consider is probably the most fundamental aspect, that of the grid (or mesh). Naturally, before looking at the equations we want to solve themselves, we must look at the computational domain and how we want to discretize it. Ideally numerical software looking to emulate FEMs should be able to construct both simple triangulated 2D domains and more complex surfaces and meshes.

For now we will look at a simple example. Let us suppose we have a domain of the following form.

\[
\Omega = \{(x, y) \in \mathbb{R}^2 : 0 \leq x \leq 1, 0 \leq y \leq 1\}. 
\]

In creating a computational grid for this domain, it will be necessary to specify the following things.

1. The shape of the domain (a square) and its vertices.
2. The number of elements.
3. The type of elements (e.g. square elements or triangles).

With these points in mind, we implement the grid in the following way in DUNE-FEMPY and plot the result in figure 2.1.
Here we create a simple square domain by specifying two opposite corners
(0,0) and (1,1), and the number of elements in each direction (4,4). We then
refine the grid and plot the results, before coarsening it again. We note that this
is a simplified example and in general grids in DUNE-FEMPY can additionally be
constructed via a dictionary containing vertex and element information, gmsh files or
dune grid format (dgf) files when more complexity is required, which is demonstrated
in the DUNE-PYTHON paper Dedner and Nolte [2018]. A list of more complicated
grids and other modules is given in appendix D.

Conceptually it is worth stating that from a design standpoint, assumptions
could potentially be made to cut down on the complexity needed. For instance in
situations where the exact details are not necessary, a basic square grid could simply
be made with grid = square(). However such a design comes at the cost of it being
unclear how to make small modifications.

2.1.2 Spaces

The next key part of a FEM after constructing the grid is defining the kind of elements we want to use, and by extension their space. In particular this is important because the order of elements used as well as the type of element space can dictate the solvability and the efficiency of the method.

Let us consider a simple case of Lagrange elements. Since we have a 2D domain with a quadrilateral mesh, we consider shape functions that are 1 on each separate node, and 0 on the others. For orders 2 and 3, the shape functions would be quadratic and cubic polynomials respectively (as shown in figure 2.2). The creation of such a Lagrange space in DUNE-FEMPY is done by the following code.

Code Listing 2.2: Creating a Lagrange space with polynomial basis functions

```python
1 import dune.create as create
2 space = create.space('lagrange', grid, dimrange=1, order=2)
```

We note that the above space is called with two default arguments and two keyword arguments.

- `'lagrange'` indicates that we will use a space with Lagrange basis functions.
- `grid` passes in the grid we constructed previously.
- `dimrange=1` (optional) sets the dimension of the range space to 1 (deduced from the UFL expression by default).
• \textbf{order=2} (optional) sets the order of the finite elements to 2 (2 is the default).

Of particular note is that the first argument corresponds to a \textsc{Dune} discrete space realization that can come from anywhere within a \textsc{Dune} installation, provided Python bindings are created for it. For instance we could use a discontinuous Galerkin space with orthonormal basis functions instead by using \textit{`dgonb’}.

\subsection{2.1.3 Grid Functions}

Having defined the computational domain and function space, we look towards functions that we may need to define, e.g. for containing the solution. In particular we want to be able to store what initial values it can take, its value at the previous time step and so on.

Let us begin by just considering a function for the initial condition. In \textsc{Dune-Fempy}, we use Unified Form Language (UFL) (Alnæs et al. [2013]) to define equations, which is essentially a human-readable way of writing a variational form. We can also use UFL to define a simple function. To this end, we must begin by defining a variable.

\textbf{Code Listing 2.3: Creating an x variable in UFL}

\begin{verbatim}
from ufl import SpatialCoordinate
x = SpatialCoordinate(space)
\end{verbatim}

Here we create \textit{x} as a spatial coordinate from UFL by using the \texttt{space} object from the previous section. The \texttt{space} gives UFL the dimensions of the grid and the range space, so it knows \textit{x} is two dimensional. So now for initial condition \( u = \frac{1}{2}(x_0^2 + x_1^2) - \frac{1}{3}(x_0^3 - x_1^3) + 1 \), we would have the following code.

\textbf{Code Listing 2.4: Creating a grid function using UFL}

\begin{verbatim}
initial = 1/2*(x[0]**2 + x[1]**2) - 1/3*(x[0]**3 - x[1]**3) + 1
\end{verbatim}

Now this function can be used in a variety of ways. Let us first show how we would compute the \( L^2 \) norm of the initial function. We do this using the \texttt{integrate} function, which we note takes as arguments the \texttt{grid} defined previously, the function
initial and the quadrature order 5. Also note that in DUNE-FEMPY functions are vectors by default, so we add \([0]\) so that it is treated as a scalar.

Code Listing 2.5: Integrating the initial data

```python
from dune.fem.function import integrate
mass = integrate(grid, initial**2, order=5)[0]
print(mass)
```

Output

```
1.840079345703125
```

We can also plot functions fairly easily. The two main ways to do this in DUNE-FEMPY are either a quick plot in matplotlib (see Hunter [2007]), or writing to a VTK file for use in Paraview (see Ahrens et al. [2005]), which we do below, resulting in figure 2.3.

Code Listing 2.6: Plotting a function using two different methods

```python
from dune.fem.plotting import plotPointData as plot
plot(initial, grid=grid)
grid.writeVTK('initial', pointdata={'initial': initial})
```

Figure 2.3: The matplotlib plot of the initial function

For the vtk output the function needs to be assigned a name, which is given by the key argument of the dictionary passed as the pointdata argument.

Note that so far we have simply evaluated the UFL expression initial directly, i.e., without using any approximation. It is equally possible to do the
above with a discrete function, which can be created through interpolation into the
discrete function space as shown below.

Code Listing 2.7:

```python
u_h = space.interpolate(initial, name='u_h')
```

So we have created a discrete function $u_h$ over the discrete finite element
space to contain the solution and used an interpolation over the space to assign its
initial value to the UFL expression $initial$. The name is used later for plotting
purposes, for example in the VTK output.

To define the weak formulation given by (2.5) we need two discrete functions,
one to store the next time step ($u^{n+1}$) and a second one ($u^n$) containing the approx-
imation of the previous time step. We use $u_h$ to store the former and construct a
copy, $u_{h\_n}$, to store the latter.

Code Listing 2.8: Copying a discrete function

```python
u_h_n = u_h.copy(name="previous")
```

### 2.1.4 Schemes

In DUNE-FEMPY, we define schemes as the object containing the weak form of the
PDE, its boundary conditions and the method used to approximate the inverse e.g.
the iterative linear solver to use. Specifically, for an operator $L : V_h \rightarrow V_h^*$, schemes
have two main methods.

1. Apply the operator. That is to calculate $w_h = L[v_h]$ given some $v_h \in V_h$.

2. Solve the PDE. That is to compute the solution $u_h$ to $L[u_h] = v_h$ given some
$v_h \in V_h^*$, by using a solve method.

**Remark.** In the case where only the operator application is required/possible (e.g.
when $L : V \rightarrow W \neq V$), an operator object can be constructed instead of a scheme
which comes without a solve method.

Recall the parabolic equation (2.5), which we will focus on in the following
example. To begin with, it is necessary to define the variables that are used in the
equation.

Code Listing 2.9: Setting up UFL variables to be used

```python
from ufl import TestFunction, TrialFunction
from dune.ufl import NamedConstant
u = TrialFunction(space)
v = TestFunction(space)
dt = NamedConstant(space, "dt")  # time step
t = NamedConstant(space, "t")    # current time
```

The trial function $u$ and the test function $v$ are defined on the same space as before. Additionally $\Delta t$ and $t$ are defined asNamedConstant, which is simply a UFL Constant variable that can be given a name so it can be more easily modified later on.

Now for the equation (2.5) itself, let us prescribe the following value for $K$.

$$K(\nabla u) = \frac{2}{1 + \sqrt{1 + 4|\nabla u|^2}}.$$  \hspace{1cm} (2.6)

This results in an implementation of the following form.

Code Listing 2.10: Implementing the weak form

```python
from ufl import dx, grad, div, inner, sqrt
abs_du = lambda u: sqrt(inner(grad(u), grad(u)))
K = lambda u: 2/(1 + sqrt(1 + 4*abs_du(u)))
a = (inner((u - u_h_n)/dt, v) \
    + 0.5*inner(K(u)*grad(u), grad(v)) \
    + 0.5*inner(K(u_h_n)*grad(u_h_n), grad(v))) * dx
```

For the exact solution we will use the following (which is consistent with the initial data)

$$u(x,t) = e^{-2t} \left( \frac{1}{2}(x_0^2 + x_1^2) - \frac{1}{3}(x_0^3 - x_1^3) \right) + 1$$  \hspace{1cm} (2.7)

We can use `initial` to define this using some algebra, and we write a lambda function that takes $t$ as argument.

Code Listing 2.11: The exact solution

```python
from ufl import as_vector, exp
```
To set the right hand side of the equation, i.e. $f$, we put the exact solution into the strong form of the equation (i.e. $u_t - \nabla \cdot (K(\nabla u) \cdot \nabla u)$). We also add in Neumann boundary conditions by substituting the exact solution into the boundary term (obtained after differentiation by parts).

Code Listing 2.12: Setting up the right hand side

```
from ufl import dot, FacetNormal, ds
n = FacetNormal(space)
b = inner(-2*exp(-2*t)*(initial - 1) \
- div(K(exact(t))*grad(exact(t)[0])), v[0]) * dx \
+ K(exact(t))*dot(grad(exact(t)[0]), n) * v[0] * ds
```

Finally, having defined the weak form and right hand side, we can now set up a scheme object which we can use to solve the PDE.

Code Listing 2.13: Creating an $H^1$ scheme

```
scheme = create.scheme("galerkin", a == b, solver='cg')
```

The above function creates a simple Galerkin method for $H^1$ conforming elements, with the space and equation passed in. We note that DUNE automatically solves nonlinear PDEs using Newton’s method so it is sufficient to simply pass in the weak form as shown. As before we also note there exist other such premade DUNE schemes for different problems (see D.4)

Additionally the linear solver for the method can be specified, so for this instance we use 'cg' for a conjugate gradient method, since the PDE is symmetric and positive definite.

Lastly we note that it is possible to explicitly define a model object to hold the method, and we investigate the different ways of doing this in section 2.3.

2.1.5 Solving

The last natural part of a FEM is the solving, which includes time loops, mesh refinements, data output, plotting, and so on. Let us begin by setting up the time
step, $\Delta t = 0.001$, by assigning it in the model (using the name given to the coefficient previously).

Code Listing 2.14: Setting up time variables before the loop

```python
scheme.model.dt = 0.001
```

Next we write the following method for solving the problem over the time range. Since the problem is time-dependent, we solve over a for loop with $t_0 = 0$ and $t_N = 1$, using $u_{h,n}$ for the old solution and $u_h$ for the new one.

Code Listing 2.15: Evolve method for solving in time

```python
def evolve(scheme, u_h, u_h_n):
    time = 0
    endTime = 1.0
    while time < (endTime - 1e-6):
        scheme.model.t = time + 0.5*scheme.model.dt
        u_h_n.assign(u_h)
        scheme.solve(target=u_h)
        time += scheme.model.dt
```

Lastly we want to have a way of computing the error. Say for instance we want to look at the $L^2$ and $H^1$ errors for our computed solution. For the error we will consider the difference between an exact solution $u$ at the final time of the simulation and our computed solution, as follows.

$$L^2 \text{ error} = \left( \int_{\Omega} |u - u_h|^2 \, dx \right)^{1/2}, \quad H^1 \text{ error} = \left( \int_{\Omega} |\nabla(u - u_h)|^2 \, dx \right)^{1/2}. \quad (2.8)$$

We can calculate the squared norm with the following code.

Code Listing 2.16: Writing expressions for the error computed at the final time

```python
exact_end = exact(1)
l2error_fn = inner(u_h - exact_end, u_h - exact_end)
hierror_fn = inner(grad(u_h - exact_end), grad(u_h - exact_end))
```

First of all we define the exact solution (exact_end) at the end time $T = 1$. Then we simply write expressions in UFL to calculate the $L^2$ and $H^1$ errors. We
note that this works even though $u_h$ is a discrete function and not a UFL term itself, since the expression is extracted from it automatically.

We also want to compute the estimated order of convergence (EOC), to test our method.

$$EOC = \frac{\log(e_{\text{new}}/e_{\text{old}})}{\log(h_{\text{new}}/h_{\text{old}})}.$$  

This is calculated by refining the grid and comparing the errors ($e_{\text{old}}$ and $e_{\text{new}}$) to the grid sizes ($h_{\text{old}}$ and $h_{\text{new}}$), where the errors are computed using the error function $l2error_{\text{fn}}$ from 2.16. In particular for a grid size that is being halved at each step, we do the following after each solve step.

**Code Listing 2.17: Calculating the EOCs**

```python
1 error_old = error  # store old error
2 error = sqrt(integrate(grid, l2error_fn, 5)[0])  # integrate
3 eoc = log(error/error_old)/log(0.5)  # do the EOC calc
4 grid.hierarchicalGrid.globalRefine(1)  # refine the grid
```

Combining these concepts into one solve method in DUNE-FEMPY, we have the following program (with resulting figure 2.4).

**Code Listing 2.18: Solving the Forchheimer equation in time and refining the grid**

```python
1 from math import log
2 error = 0
3 for eocLoop in range(3):
4     print('# step:', eocLoop, ', size:', grid.size(0))
5     u_h.interpolate(initial)
6     evolve(scheme, u_h, u_h_n)
7     error_old = error
8     error = sqrt(integrate(grid, l2error_fn, 5)[0])
9     if eocLoop == 0:
10         eoc = '-'
11     else:
12         eoc = log(error/error_old)/log(0.5)
13     print('|u_h - u| =', error, ', eoc =', eoc)
14     plot(u_h)
15     grid.writeVTK('forchheimer', pointdata={'u': u_h, 'l2error':
```
\[ l2error_f, \ 'herror': herror_f, \]
\[ \text{number}=\text{eocLoop} \]
\[ \text{grid.hierarchicalGrid.globalRefine}(1) \]
\[ \text{scheme.model.dt} /= 2 \]

Output

1
2
3
4
5
6

\[ \text{\# step: 0, size: 16} \]
\[ |u_h - u| = 2.9194982026064784e-05, \text{eoc} = - \]
\[ \text{\# step: 1, size: 64} \]
\[ |u_h - u| = 3.6106320903708674e-06, \text{eoc} = 3.0153970951632156 \]
\[ \text{\# step: 2, size: 256} \]
\[ |u_h - u| = 4.5004939236970754e-07, \text{eoc} = 3.0040961733992497 \]

Figure 2.4: Plot of solutions at each level of refinement

We compile a table of the errors and EOCs for additional refinement steps and also including the $H^1$ error below.

| Elements | $\|u - u_h\|_{L^2}$ | EOC | $|u - u_h|_{H^1}$ | EOC |
|----------|----------------------|-----|-------------------|-----|
| 16       | 2.919e-05            | -   | 8.917e-04         | -   |
| 64       | 3.611e-06            | 3.015 | 2.223e-04       | 2.000 |
| 256      | 4.500e-07            | 3.004 | 5.573e-05       | 2.000 |
| 1024     | 5.621e-08            | 3.001 | 1.393e-05       | 2.000 |
| 4096     | 7.031e-09            | 2.999 | 3.483e-06       | 2.000 |

2.2 Alternate Solve Methods

We carry on our explanation of different DUNE-FEMPY features by looking at the different methods of solving the PDE, which is facilitated by the different storage
back ends for spaces. DUNE-FEM allows one to store DoF vectors and matrices
directly based on the data structures from different linear algebra packages.

We can specify alternate storage types as follows.

Code Listing 2.19: Accessing different storage types

```python
space = create.space(‘lagrange’, grid, dimrange=1, order=2,
                      storage=‘istl’)
```

As before we construct the space, but now with the additional argument that spec-
ifies the usage of DUNE-Istl (see Blatt and Bastian [2006]) as a linear algebra
backend. By default we use a very simple storage structure directly provided in
DUNE-FEM, consequently not requiring any additional packages. A number of sim-
ple Krylov type solvers are available. Changing the storage argument in the con-
struction of the space makes it possible to use more sophisticated solvers (e.g., better
preconditioners or direct solvers). Available possibilities are shown in appendix D.3.

In particular one thing that we can do with certain storage methods is inte-
grate methods from SciPy (Jones et al. [2001–]) into our code. This allows for more
complex ways of writing numerical methods without the need to explicitly write it
on the C++ side. Additionally we will show that it is possible to store the degrees of
freedom in such a way that they can be treated as vectors from the NumPy package
(Oliphant [2006]) and an assembled system matrix can be stored in a SciPy sparse
matrix.

We present these methods once again via the Forchheimer example from
section 2.1.

In the following we implement a simple Newton solver: given an initial guess
$u^0$ (here taken to be zero) solve for $n \geq 0$,

$$u^{n+1} = u^n - DS(u^n)(S(u^n) - g),$$

where $g$ is a discrete function containing the Dirichlet boundary conditions if they exist.

Usually this would be automatically taken care of in DUNE-FEMPY by
scheme.solve, however this time we will use the call operator on the scheme to compute $S(u^n)$ as well as scheme.assemble to get a copy of the system matrix in form of a SciPy sparse row matrix. Note that this method is not available for all storage types. We present this alternative below, and plot the result in figure 2.5.

Code Listing 2.20: Creating a class to hold a different solve method

```python
import numpy as np
from scipy.sparse.linalg import spsolve
class Scheme:
    def __init__(self, scheme):
        self.model = scheme.model

    def solve(self, target=None):
        # create a copy of target for the residual
        res = target.copy(name="residual")

        # create numpy vectors to store target and res
        sol_coeff = target.as_numpy
        res_coeff = res.as_numpy

        n = 0
        while True:
            scheme(target, res)
            absF = math.sqrt(np.dot(res_coeff, res_coeff))
            if absF < 1e-10:
                break
            matrix = scheme.assemble(target).as_numpy
            sol_coeff -= spsolve(matrix, res_coeff)
            n += 1

        scheme_cls = Scheme(scheme)

        grid.hierarchicalGrid.globalRefine(-2)  # revert grid refinement
        u_h.interpolate(initial)  # reset u_h to initial
        scheme.model.dt = 0.05  # reset time step
        evolve(scheme_cls, u_h, u_h_n)
        plot(u_h)
```
We can redo the above computation using a Newton-Krylov solver from SciPy. We do this by constructing a class\texttt{Df} containing the derivative of the operator. This would normally be done within DUNE, but here we do it purely through Python, giving figure 2.6 which is identical to before.

Figure 2.5: Plot of solution for Python-side Newton scheme

![Plot of solution for Python-side Newton scheme](image)

We can redo the above computation using a Newton-Krylov solver from SciPy. We do this by constructing a class\texttt{Df} containing the derivative of the operator. This would normally be done within DUNE, but here we do it purely through Python, giving figure 2.6 which is identical to before.

```
from scipy.optimize import newton_krylov
from scipy.sparse.linalg import LinearOperator

def f(x_coeff):
    res = u_h.copy(name="residual")
    res_coeff = res.as_numpy
    x = space.numpyFunction(x_coeff, "tmp")
    scheme(x, res)
    return res_coeff

class Df(LinearOperator):
    def __init__(self, x_coeff):
        self.shape = (x_coeff.shape[0], x_coeff.shape[0])
        self.dtype = x_coeff.dtype
        # the following converts a given numpy array
        # into a discrete function over the given space
```
x = space.numpyFunction(x_coeff, "tmp")
# store the assembled matrix
self.jac = scheme.assemble(x).as_numpy
# reassemble the matrix DF(u) given a DoF vector for u

def update(self, x_coeff, f):
    x = space.numpyFunction(x_coeff, "tmp")
    # Note: the following does produce a copy of the matrix
    # and each call here will reproduce the full matrix
    # structure - no reuse possible in this version
    self.jac = scheme.assemble(x).as_numpy
    # compute DS(u)^{-1}x for a given DoF vector x

def _matvec(self, x_coeff):
    return spsolve(self.jac, x_coeff)

class Scheme2:
    def __init__(self, scheme):
        self.scheme = scheme
        self.model = scheme.model
    def solve(self, target=None):
        sol_coeff = target.as_numpy
        # call the newton krylov solver from scipy
        sol_coeff[:] = newton_krylov(f, sol_coeff, verbose=0, f_tol=1e-8, inner_M=Df(sol_coeff))

scheme2_cls = Scheme2(scheme)
u_h.interpolate(initial)
evolve(scheme2_cls, u_h, u_h_n)
plot(u_h)

We can also solvers from the PETSc package (see Balay et al. [2018]) to solve the problem. This can be done either through bindings available in DUNE-FEM or through the petsc4py package (Dalcin et al. [2011]).

The first step is to change the storage in the space. This also requires setting

---

1For this to work, one must make sure that DUNE has been configured using the same version of PETSc used for petsc4py.
up the scheme and discrete functions again to use the new storage structure.

We can directly use the PETSc solvers by invoking `solve` on the scheme as before. Note that to do this we must change the storage type by creating a new `space`. Then we have the following code, with the same results found once again in figure 2.7.

**Code Listing 2.22: Using petsc4py to solve using PETSc**

```python
space = create.space("lagrange", grid, dimrange=1, order=2, storage='petsc')
scheme = create.scheme("galerkin", a == b, space=space, parameters={"petsc.preconditioning.method":"sor"})
# first we will use the petsc solver available in the dune-fem package (using the sor preconditioner)
u_h = space.interpolate(initial, name='u_h')
u_h_n = u_h.copy(name='previous')
scheme.model.dt = 0.05
evolve(scheme, u_h, u_h_n)
plot(u_h)
```

Next we will implement the Newton loop in Python using **petsc4py** to solve the linear systems. We can access the PETSc vectors by calling `as_petsc` on the discrete function. Note that this property will only be available if the discrete function is an element of a space with storage `'petsc'`. The method `assemble` on the scheme now returns the sparse PETSc matrix and so we can directly use the
KSP class from petsc4py.

Code Listing 2.23: Using petsc4py and its Krylov solvers to define a Newton scheme and solve

```python
import petsc4py, sys
petsc4py.init(sys.argv)
from petsc4py import PETSc
ksp = PETSc.KSP()
ksp.create(PETSc.COMM_WORLD)
# use conjugate gradients method
ksp.setType("cg")
# and incomplete Cholesky
ksp.getPC().setType("icc")

class Scheme3:
    def __init__(self, scheme):
        self.model = scheme.model
    def solve(self, target=None):
        res = target.copy(name="residual")
        sol_coeff = target.as_petsc
        res_coeff = res.as_petsc
        n = 0
        while True:
            scheme(target, res)
            absF = math.sqrt(res_coeff.dot(res_coeff))
```

Figure 2.7: Plot of solution using PETSc
if absF < 1e-10:
    break
matrix = scheme.assemble(target).as_petsc
ksp.setOperators(matrix)
ksp.setFromOptions()
ksp.solve(res_coeff, res_coeff)
sol_coeff -= res_coeff
n += 1

u_h.interpolate(initial)
scheme3_cls = Scheme3(scheme)
evolve(scheme3_cls, u_h, u_h_n)

plot(u_h)

Figure 2.8: Plot of solution using PETSc and a Krylov method

Finally we will use PETSc’s nonlinear solvers (the snes classes) directly.

Code Listing 2.24: Using petsc4py and their nonlinear solvers (SNES) directly

def f(snes, X, F):
inDF = space.petscFunction(X)
outDF = space.petscFunction(F)
scheme(inDF, outDF)
def Df(snes, x, m, b):
inDF = space.petscFunction(x)
matrix = scheme.assemble(inDF).as_petsc
m.createAIJ(matrix.size, csr=matrix.getValuesCSR())
b.createAIJ(matrix.size, csr=matrix.getValuesCSR())
```python
class Scheme4:
    def __init__(self, scheme):
        self.scheme = scheme
        self.model = scheme.model

    def solve(self, target=None):
        res = target.copy(name="residual")
        sol_coeff = target.as_petsc
        res_coeff = res.as_petsc

        snes = PETSc.SNES().create()
        snes.setMonitor(lambda snes, i, r: print())
        snes.setFunction(f, res_coeff)
        matrix = self.scheme.assemble(target).as_petsc
        snes.setJacobian(Df, matrix, matrix)
        snes.getKSP().setType("cg")
        snes.setFromOptions()
        snes.solve(None, sol_coeff)

u_h.interpolate(initial)
scheme4_cls = Scheme4(scheme)
evolve(scheme4_cls, u_h, u_h_n)
plot(u_h)
```

Figure 2.9: Plot of solution using SNES
Remark. The methods `as_numpy` and `as_petsc` (returning the DoF vector either as a `numpy` or a `petsc` vector) do not lead to a copy of the data and the same is true for the `function` objects returned by the `function` method on the discrete space. In the `numpy` case we can use Python’s buffer protocol to use the same underlying storage. In the case of `petsc` the underlying `Vec` can be shared. In the case of matrices the situation is not yet as clear; `scheme.assemble` returns a copy of the data in the SciPy case while the `Mat` structure is shared between C++ and Python in the `petsc` case. But at the time of writing it is not possible to pass in the `Mat` structure to the `scheme.assemble` method from the outside. That is why it is necessary to copy the data when using the `snes` nonlinear solver as seen above.

2.3 Model Generation

After having looked at the basics of the Python interface, let us now consider features more aimed at C++ integration and code extensibility.

We briefly alluded to a way of creating a model independent of a scheme in section 2.1.4, so let us expand upon this idea here. We can separate the process into two steps as follows.

```python
model = create.model('elliptic', grid, a == b)
scheme = create.scheme("galerkin", model)
```

where `a == b` refers to the UFL expression used to represent the weak form, and `'elliptic'` refers to the elliptic operator class in DUNE. The purpose of this class is to have a set list of virtualized methods that represent the standard structure of what we consider is necessary for an elliptic PDE model. This is then implemented in DUNE-FEMPY by translating the UFL input into a DUNE class compatible format.

We note that this is a different approach to similar packages, like Fenics Alnæs et al. [2015] and Firedrake Rathgeber et al. [2017], which in general do not create a virtual class and instead use the UFL form directly. In fact such an approach is also available in DUNE-FEMPY, using the `'integrands'` identifier. The choice
between 'elliptic' and 'integrands' models each lead to separate advantages and disadvantages, which is what we would like to discuss in this section.

### 2.3.1 Elliptic Models

First let us consider the elliptic model. Summarized briefly, the elliptic model is the version of model generation in DUNE-FEMPY that follows as closely as possible the structure used in DUNE-FEM. In mathematical terms, consider the same general operator we defined previously.

\[
L[u] = -\nabla \cdot D(x, u, \nabla u) \nabla u + m(x, u, \nabla u). \tag{2.2}
\]

In variational form, after multiplying with a test function and integration by parts (ignoring boundary terms for now), we arrive at

\[
\langle L[u], v \rangle = \int_{\Omega} D(x, u, \nabla u) \nabla u \cdot \nabla v + m(x, u, \nabla u)v\,dx.
\]

Now the elliptic model class in DUNE-FEMPY has methods that represent the above form in general terms. Suppose for instance we were to take the case of \(m(x, u, \nabla u) = u\) above. In the model class, this would be defined under the method `source`.

**Code Listing 2.26: A function in the elliptic model C++ class**

```cpp
template < class Point >
void source ( const Point &x, const RangeType &u, const JacobianRangeType &du, RangeType &result ) const
{
    result[0] = u[0];
}
```

A similar method exists for \(D(x, u)\), as well as linearised versions for the purposes of nonlinear methods. Additionally there are methods for the associated Dirichlet or Neumann boundary conditions. Together these form the elliptic model class which is one way of expressing weak forms in DUNE-FEMPY. Generally this class is then
used to create a shared object file that is exported to Python using pybind11 for use in Python scripts and notebooks.

Another possibility provided for by the modular design of DUNE-PYTHON and the elliptic class structure is the ability to extend the model to more complex cases. This approach involves the writing of additional C++ classes (one example would be an elliptic discontinuous Galerkin model) based on the elliptic model class except with extra modifications that one might want to make to the underlying structure. Whilst this approach is more in-depth than simply editing a few lines in the model file, it allows one to change the functions themselves beyond what the default elliptic model accepts.

An example of this approach is the nonvariational model for the DUNE-femnv module (see chapter 3). This comes from the desire to write weak forms that can accept a Hessian as an argument as follows.

$$L[u] = -\nabla \cdot D(x,u,\nabla u) \nabla u(x) + m(x,u,\nabla u, D^2 u).$$

Such a change would require different arguments to be made available to the methods from the elliptic model. Suppose we wanted to implement the nonvariational Poisson equation, i.e. taking $m = -\Delta u$ above. Then we would need the following method.

Code Listing 2.27: A nonvariational method

```cpp
1 template< class Point >
2 void source ( const Point &x, const RangeType &u, const
3 JacobianRangeType &du, const HessianRangeType &d2u, RangeType
4 &result ) const
5 {
6    result[ 0 ] = d2u[ 0 ][ 0 ] + d2u[ 1 ][ 1 ];
7 }
```

Whilst this may not be immediately possible with the standard elliptic model, it is possible to create a model `nvdg` that can use such functions, which results in the ability to write functional DUNE-FEMPY code as follows.

Code Listing 2.28: The DUNE-FEMPY code for a nonvariational model
Thus it becomes possible to write schemes that expect different arguments from the operator.

### 2.3.2 Integrands Models

We note however that as mentioned before, there exists another way of constructing operators, by using 'integrands'. This method bypasses the virtual methods used in the elliptic operator class and creates methods purely using the UFL expressions given to it. This again allows for expressions that are not by default allowed in the default elliptic class, as shown below.

```python
a = -(grad(grad(u[0])[0])[0] + grad(grad(u[0])[1])[1])*v[0]*dx + jump(A*grad(u[0]), n)*avg(v[0])*dS
b = rhs(A, exact)
model = create.model("nvdg", grid, a == b)
```

Here we take the nonvariational equation from before and add a term defined only on the skeleton (the edges) of the mesh. Due to the ability to add such interior terms, the 'integrands' class is particularly useful for discontinuous Galerkin methods.

In summary, we state that the 'integrands' model is the most versatile version of model generation for most ordinary cases (it is also more efficiently implemented due to its recency); however the elliptic model lends itself better to code extension possibilities.

### 2.3.3 C++ Models

In addition to the automatic creation of a shared library object that is done when `create` is called, it is possible to generate a model class separately as a header file. That is, it is possible to generate a C++ file (e.g. 'model.hh') that can be used
flexibly in both DUNE-FEMPY and regular C++ compatible DUNE code. We can do this by writing a pure UFL file and calling cmake on it.

Let us examine what this file looks like for the Forchheimer model.

```plaintext
1 space = Space(2, 1)
2 u = TrialFunction(space)
3 v = TestFunction(space)
4 x = SpatialCoordinate(space.cell())
5 dt = NamedConstant(triangle, "dt") # time step
6 t = NamedConstant(triangle, "t") # current time
7 n = FacetNormal(space)
8 u_h_n = NamedCoefficient(space,"previous")
9
10 from ufl import as_vector, exp
11 exact = lambda t: as_vector([exp(-2*t)*(initial - 1) + 1])
12
13 initial = 1/2*(x[0]**2 + x[1]**2) - 1/3*(x[0]**3 - x[1]**3) + 1
14 abs_du = lambda u: sqrt(inner(grad(u), grad(u)))
15 K = lambda u: 2/(1 + sqrt(1 + 4*abs_du(u)))
16
17 a = (inner((u - u_h_n)/dt, v) \n      + 0.5*inner(K(u)*grad(u), grad(v)) \n      + 0.5*inner(K(u_h_n)*grad(u_h_n), grad(v)) ) * dx
18 b = inner(-2*exp(-2*t)*(initial - 1) \n      - div(K(exact(t))*grad(exact(t)[0])), v[0]) * dx \n      + K(exact(t))*dot(grad(exact(t)[0]), n) + v[0] * ds
19
20 F = a - b
```

Once the corresponding forchheimer.hh file has been generated, it can then be edited manually in C++, and then used in place of a UFL expression in DUNE-FEMPY. This choice of default shared library generation or usable header files falls in line with attempts we have made to improve extensibility of the code, since in particular it allows for the user to write in more complex features in C++ that do not necessarily have Python bindings written for them.

One natural question that arises is that of the efficiency between using the
full DUNE-FEMPY interface to run problems and simply using it to just generate a model file to be used in C++. To look at this problem we constructed an identical Forchheimer example in C++ which can be found in appendix C. We compare the runtime of the two solve-steps below.

<table>
<thead>
<tr>
<th>Elements</th>
<th>Time step (Δt)</th>
<th>Python runtime (s)</th>
<th>C++ runtime (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>256</td>
<td>6.25e-4</td>
<td>13.7</td>
<td>33.0</td>
</tr>
<tr>
<td>1024</td>
<td>3.125e-4</td>
<td>109.3</td>
<td>101.9</td>
</tr>
<tr>
<td>4096</td>
<td>1.5625e-4</td>
<td>890.0</td>
<td>864.3</td>
</tr>
</tbody>
</table>

Thus we see there is not a sizable difference between the DUNE-FEM and DUNE-FEMPY versions (and to begin with the Python version even appears to be faster).

Remark. We remark that both versions use preprocessing, since in use-cases such as long-running simulations, this extra time is negligible. We also note that the most costly aspect of pure Python code are generally callbacks, and in this example this amounts to just the solve call. An additional example that taxes the two versions differently can be found in section 2.7.

### 2.4 Adaptive Mesh Refinement

We shall now consider the implementation of adaptive mesh refinement in DUNE-FEMPY. Adaptive mesh refinement is a technique that allows for the targeted refinement of the computational domain in specific areas where there is greater turbulence or activity, for greater precision. In problems where uniform refinement of a mesh is not required, this allows for more precision of the results at less computational cost.

The method considered here uses so-called h-adaptivity that adds additional mesh points to the grid at areas of small scale activity. It does so based on a marking procedure that evaluates the gradient of the solution at each element and determines whether to refine the grid based on a level of tolerance.

In this section we present two examples which use adaptive grid refinement.
in slightly different ways.

2.4.1 Re-entrant Corner Problem

Here we will consider the classic re-entrant corner problem,

\[-\Delta u = f, \quad \text{in } \Omega,\]
\[u = g, \quad \text{on } \partial \Omega,\]

where the domain is given using polar coordinates,

\[\Omega = \{(r, \varphi): r \in (0, 1), \varphi \in (0, \Phi)\}.\]

For the boundary condition \(g\), we set it to the trace of the function \(u\), given by

\[u(r, \varphi) = r^\frac{\Phi}{\pi} \sin \left(\frac{\pi}{\Phi} \varphi\right)\]

Now we start by importing some necessary modules.

```python
import math
import numpy
import dune.create as create
import matplotlib.pyplot as pyplot
from dune.fem.view import adaptiveLeafGridView
from dune.fem.plotting import plotPointData as plot
import dune.grid as grid
import dune.fem as fem

Phi = 16/9*math.pi
order = 2

We set the angle for the corner \(\Phi\), (where \(0 < \Phi \leq 2\pi\)), and the order for the space.

We now define the grid for this domain, which has its vertices at the origin and 7 equally spaced points on the unit sphere, starting with \((1, 0)\) and ending at \((cos(\Phi), sin(\Phi))\). We also define the interior triangles using these numbered vertices.
vertices = numpy.zeros((8, 2))
vertices[0] = [0, 0]
for i in range(0, 7):
    vertices[i+1] = [math.cos(Phi/6*i),
    math.sin(Phi/6*i)]
triangles = numpy.array([[2,1,0], [0,3,2], [4,3,0],
    [0,5,4], [6,5,0], [0,7,6]])
domain = {"vertices": vertices, "simplices": triangles}
view = create.view("adaptive", "ALUConform", domain, dimgrid=2)
view.hierarchicalGrid.globalRefine(2)
space = create.space("lagrange", view, dimrange=1, order=order)

Next we define the model. We obtain $\phi$ from the $x$ and $y$ coordinates, define the exact solution $u(r, \phi)$ and the weak form $\int_{\Omega} \nabla u \cdot \nabla v \, dx = 0$.

```python
from ufl import *
from dune.ufl import DirichletBC
u = TrialFunction(space)
v = TestFunction(space)
x = SpatialCoordinate(space.cell())

phi = atan_2(x[1], x[0]) + conditional(x[1] < 0, 2*pi, 0)
# define the exact solution $u(r, \phi)$
exact = as_vector([inner(x, x)**(pi/2/Phi) * sin(pi/Phi * phi)])
# define the bilinear form
a = inner(grad(u), grad(v)) * dx

# set up the scheme
laplace = create.scheme("galerkin", [a==0,
    DirichletBC(space, exact, 1)])
uh = space.interpolate(lambda x: [0], name="solution")
```

For the following we use the well-known a-posteriori error estimate (see e.g. [Verfürth, 1994, p71])

$$
\int_{\Omega} |\nabla (u - u_h)|^2 \leq C \sum_K \eta_K^2,
$$
where on each element $K$ of the grid the local estimator is given by

$$
\eta^2_K = h^2_K \int_K |\Delta u_h|^2 \, dx + \frac{1}{2} \sum_{S \subset \partial K} h_S \int_S |\nabla u_h|^2 \, ds.
$$

Here $[\cdot]$ is the jump in the normal direction over the edges of the grid, $h_K = \max_{x,y \in K} |x - y|$ and $h_S$ is the length of side $S$.

We compute the elementwise indicator by defining a weak form

$$
\eta(u,v) = h^2_K \int_K |\Delta u_h|^2 v \, dx + \sum_{S \subset \partial K} h_S \int_S |\nabla u_h|^2 \{v\} \, ds,
$$

where $\{\cdot\}$ is the average over the cell edges. This weak form can be easily written in UFL and by using it to define a discrete operator $L$ from the second order Lagrange space into a space containing piecewise constant functions we have $L[u_h]|_K = \eta_K$.

```python
# energy error
h1_error = inner(grad(u - exact), grad(u - exact))
#
# define a FV space to do the error estimation
fvspace = create.space("finitevolume", view,
    dimrange=1, storage="istl")
#
# define hK, hS, n and the elementwise estimator
hK = MaxCellEdgeLength(space.cell())
hS = MaxFacetEdgeLength(space.cell())('+')
n = FacetNormal(space.cell())
estimator_ufl = hK**2 * (div(grad(u[0])))**2 * v[0] * dx + hS * 
    inner(jump(grad(u[0])), n('+'))**2 * avg(v[0]) * dS
# we define an operator that takes uh and applies the above formula
estimator = create.operator("galerkin", estimator_ufl == 0,
    space, fvspace)
```

Lastly, let us set up a marking function for the grid adaptivity. The function `mark` (that gets directly passed into the grid) estimates the error locally, compares it to the tolerance, and refines the grid if necessary.

```python
tolerance = 0.1
```

43
gridSize = view.size(0)
estimate = fvspace.interpolate([0], name="estimate")
def mark(element):
    estLocal = estimate(element, element.geometry.
        referenceElement.center)
    return grid.Marker.refine if estLocal[0] \
        > tolerance / gridSize else grid.Marker.keep

Let us solve over a time loop and plot the solutions side by side.

# adaptive loop (solve, mark, estimate)
fig = pyplot.figure(figsize=(10,10))
count = 0
while count < 20:
    laplace.solve(target=uh)
    if count%3 == 0:
        pyplot.show()
        pyplot.close('all')
        fig = pyplot.figure(figsize=(10,10))
    plot(uh, figure=(fig, 131+count%3), colorbar=False)
    # compute the actual error and the estimator
    error = math.sqrt(fem.function.integrate(view, h1_error, 5)[0])
estimator(uh, estimate)
    eta = sum(estimate.dofVector)
    if eta < tolerance:
        break
    if tolerance == 0.:
        view.hierarchicalGrid.globalRefine(2)
uh.interpolate([0])  # initial guess needed
    else:
        marked = view.hierarchicalGrid.mark(mark)
        fem.adapt(view.hierarchicalGrid, [uh])
        fem.loadBalance(view.hierarchicalGrid, [uh])
        gridSize = view.size(0)
        laplace.solve( target=uh )
        count += 1
        pyplot.show()
pyplot.close('all')
Let’s have a look at the center of the domain:

```python
fig = pyplot.figure(figsize=(15, 15))
plot(uh, figure=(fig, 131), xlim=(-0.5, 0.5),
```
ylim=(-0.5, 0.5), colorbar={"shrink": 0.25})
plot(uh, figure=(fig, 132), xlim=(-0.25, 0.25),
        ylim=(-0.25, 0.25), colorbar={"shrink": 0.25})
plot(uh, figure=(fig, 133), xlim=(-0.125, 0.125),
        ylim=(-0.125, 0.125), colorbar={"shrink": 0.25})
pyplot.show()
pyplot.close('all')

Figure 2.13: Zooming in on the re-entrant corner

Finally, let us have a look at the grid levels.

from dune.fem.function import levelFunction
plot(levelFunction(view), xlim=(-0.2,1), ylim=(-0.2,1))

Figure 2.14: Plot of the level function of the grid

We shall now look at a model for crystallization on the surface of a liquid.
2.4.2 Crystal Growth

Here we demonstrate crystallisation on the surface of a liquid due to cooling from Guyer et al. [2009]. Before anything else let us set up the grid and the function space. We use the default DoF storage available in DUNE-FEM (this can be changed for example to istl, eigen or petsc).

```python
import dune.fem as fem
from dune.grid import Marker, cartesianDomain
import dune.create as create

order = 1
dimDomain = 2  # we are solving this in 2D
dimRange = 2  # we have a system with two unknowns
domain = cartesianDomain([4, 4], [8, 8], [3, 3])
grid = create.view("adaptive", grid="ALUConform",
constructor=domain, dimgrid=dimDomain)
space = create.space("lagrange", grid, dimrange=dimRange,
order=order, storage="fem")
```

We want to solve the following system of equations of variables $\phi$ (phase field) and $T$ (temperature field)

$$
\sigma \frac{\partial \phi}{\partial t} = \nabla \cdot D(\phi) \nabla \phi + m(\phi, T),
$$

$$
\frac{\partial T}{\partial t} = 2.25 \Delta T + \frac{\partial \phi}{\partial t},
$$

where $\sigma$ is a constant, $m(\phi, T)$ is given by

$$
m(\phi, T) = \phi (1 - \phi) (1 - \frac{1}{2} \frac{\kappa_1}{\pi} \arctan(\kappa_2 T)),
$$

(here $\kappa_1$ and $\kappa_2$ are constants) and $D(\phi)$ is a matrix representing anisotropic diffusion given by

$$
D(\phi) = \alpha^2 (1 + c\beta) \begin{pmatrix}
1 + c\beta & -c\frac{\beta}{\partial \psi} \\
-c\frac{\beta}{\partial \psi} & 1 + c\beta
\end{pmatrix},
$$

and where $\beta = \frac{1 - \Phi^2}{1 + \Phi^2}$, $\Phi = \tan \left( \frac{N}{2} \psi \right)$, $\psi = \theta + \arctan \left( \frac{\partial \phi/\partial y}{\partial \phi/\partial x} \right)$ and $\alpha, c, \theta$ and $N$ are
Let us first set up the parameters for the problem.

\[
\begin{align*}
\alpha &= 0.015 \\
\sigma &= 3.0 \times 10^{-4} \\
\kappa_1 &= 0.9 \\
\kappa_2 &= 20. \\
c &= 0.02 \\
N &= 6.
\end{align*}
\]

As we will be discretising in time, we define the unknown data as \( u_n = (\phi_n, T_n)^T \), with given data (from the previous time step) as \( u_{n-1} = (\phi_{n-1}, T_{n-1})^T \) and test function as \( v = (v_0, v_1)^T \).

```python
from ufl import TestFunction, TrialFunction, Constant
from dune.ufl import NamedConstant

u = TrialFunction(space)
v = TestFunction(space)
dt = NamedConstant(space, "dt")  # time step
```

We define the initial data and create a function from it. We use this value to set up our solution \( u_n \) and previous solution \( u_{n-1} \).

```python
def initial(x):
    r = (x - [6, 6]).two_norm
    return [0 if r > 0.3 else 1, -0.5]

initial_gf = create.function("global", grid, "initial",
                          order+1, initial)
u_h = space.interpolate(initial_gf, name="solution")
u_h_n = u_h.copy()  # previous solution
```

To obtain the numerical scheme, we begin by multiplying the first equation by \( v_0 \) and the second by \( v_1 \) and integrate by parts to obtain

\[
\int \sigma \frac{\partial \phi}{\partial t} v_0 \, dx = \int (-D(\phi) \nabla \phi) \cdot \nabla v_0 + m(\phi, T)v_0) \, dx,
\]

\[
\int \frac{\partial T}{\partial t} v_1 \, dx = \int \left( -2.25 \nabla T \cdot \nabla v_1 + \frac{\partial \phi}{\partial t} v_1 \right) \, dx.
\]

We then discretise the time derivatives via \( \frac{\partial \phi}{\partial t} = (\phi_n - \phi_{n-1})/\Delta t \) and \( \frac{\partial T}{\partial t} = (\ldots) \).
\( (T_n - T_{n-1})/\Delta t \). For the other terms we discretize implicitly (i.e. using \( \phi_n \) and \( T_n \)), with the exception of \( D(\phi_{n-1}) \). This ultimately results in

\[
\int (\phi_n - \phi_{n-1}) v_0 \, dx = \int \frac{\Delta t}{\sigma} (-D(\phi_{n-1}) \nabla \phi_n) \cdot \nabla v_0 + m(\phi_n, T_n) v_0) \, dx,
\]

\[
\int (T_n - T_{n-1}) v_1 \, dx = \int (-2.25 \nabla T_n \cdot \nabla v_1 + (\phi_n - \phi_{n-1}) v_1) \, dx.
\]

To finally get the desired equation, we add both equations together and rewrite using vector notation.

\[
\int \left( \frac{\Delta t}{\sigma} (D(\phi_{n-1}) \nabla \phi_n) \cdot \nabla v_0 + 2\Delta t 2.25 \nabla T_n \cdot \nabla v_1 + u_n \cdot v - s \cdot v \right) \, dx \\
= \int (u_{n-1} \cdot v - \phi_{n-1} v_1) \, dx,
\]

where

\[
s = \left( \frac{\Delta t}{\sigma} m(\phi_n, T_n), \phi_n \right)^T.
\]

Let us put this into code. First we put in the right hand side which only contains explicit data.

```python
from ufl import inner, dx
a_ex = (inner(u_h_n, v) - inner(u_h_n[0], v[1])) * dx
```

For the left hand side we have the spatial derivatives and the implicit parts.

```python
from ufl import pi, atan, atan_2, tan, grad, as_vector, inner, dot
psi = pi/8.0 + atan_2 (grad(u_h_n[0])[1], (grad(u_h_n[0])[0]))
Phi = tan (N / 2.0 * psi)
beta = (1.0 - Phi*Phi) / (1.0 + Phi*Phi)

dbeta_dPhi = -2.0 * N * Phi / (1.0 + Phi*Phi)
fac = 1.0 + c * beta
diag = fac * fac
offdiag = -fac * c * dbeta_dPhi
d0 = as_vector([diag, offdiag])
d1 = as_vector([-offdiag, diag])
m = u[0] * (1.0 - u[0]) * (u[0] - 0.5 - kappa1/pi*atan(kappa2*u[1]))
s = as_vector([dt / sigma * m, u[0]])

alpha = alpha*alpha*dt / sigma * (inner(dot(d0, grad(u[0]))),
```
We set up the scheme with some parameters.

```python
solverParameters = {
    "fem.solver.newton.tolerance": 1e-5,
    "fem.solver.newton.linabstol": 1e-8,
    "fem.solver.newton.linreduction": 1e-8,
    "fem.solver.newton.verbose": 0,
    "fem.solver.newton.linear.verbose": 0
}
scheme = create.scheme("galerkin", a_im == a_ex, space,
    solver="gmres", parameters=solverParameters)
```

We set up the adaptive method. We start with a marking strategy based on the value of the gradient of the phase field variable.

```python
def mark(element):
    u_h_local = u_h.localFunction(element)
    grad = u_h_local.jacobian(element.geometry.
        referenceElement.center)
    if grad[0].infinity_norm > 1.2:
        return Marker.refine if element.level < maxLevel \
        else Marker.keep
    else:
        return Marker.coarsen
```

We do the initial refinement of the grid.

```python
maxLevel = 11
hgrid = grid.hierarchicalGrid
hgrid.globalRefine(6)
for i in range(0, maxLevel):
    hgrid.mark(mark)
    fem.adapt(hgrid, [u_h])
    fem.loadBalance(hgrid, [u_h])
    u_h.interpolate(initial_gf)
```
Let us start by plotting the initial state of the material, which is just a small circle in the centre.

```python
from dune.fem.plotting import plotComponents as plotComponents
import matplotlib.pyplot as pyplot
from dune.fem.function import levelFunction, partitionFunction
import matplotlib
vtk = grid.sequencedVTK("crystal", pointdata=[u_h],
    celldata=[levelFunction(grid), partitionFunction(grid)])

matplotlib.rcParams.update({‘font.size’: 10})
matplotlib.rcParams[‘figure.figsize’] = [10, 5]
plotComponents(u_h, cmap=pyplot.cm.rainbow, show=[0])
```

Figure 2.15: The initial adapted grid and phase field

We set \( dt \) and the initial time \( t = 0. \)

```python
scheme.model.dt = 0.0005
t = 0.0
```

Finally we set up the time loop and solve the problem - each time this cell is run the simulation will progress to the given endTime and then the result is shown. The simulation can be progressed further by rerunning the cell while increasing the endTime.

endTime = 0.05
while t < endTime:
    u_h.n.assign(u_h)
scheme.solve(target=u_h)
    print(t, grid.size(0), end="\r")
t += scheme.model.dt
hgrid.mark(mark)
fem.adapt(hgrid, [u_h])
fem.loadBalance(hgrid, [u_h])
print()

plotComponents(u_h, cmap=pyplot.cm.rainbow)

Figure 2.16: The grid, phase field and temperature after the final timestep

2.5 Moving Meshes

In this section we will consider an example where the grid itself changes over time subject to PDEs. Specifically we refer to a geometric evolution equation, which describes the motion of a hypersurface by prescribing its velocity geometrically. In DUNE-FEMPY, it is possible to accomplish this through the following process.

1. Create an interpolated function that describes the initial surface, i.e.

   ```python
   positions = space.interpolate(lambda x: some_function(x),
                                name="position")
   ```

2. Create a surface from positions using this function
3. Create the scheme that describes the surface evolution and solve it in the usual way.

4. Update the surface using the computed solution.

With this process the surface (and by extension the mesh) can be changed over time. We will now demonstrate this in a mean curvature flow example.

### 2.5.1 Mean Curvature Flow

Mean curvature flow is a specific example of a geometric evolution equation where the evolution is governed by the mean curvature $H$. One real-life example of this is in how soap films change over time, although it can also be applied to image processing (e.g. Malladi and Sethian [1996]). Assume we can define a reference surface $\Gamma_0$ such that we can write the evolving surface $\Gamma(t)$ in the form

$$\Gamma(t) = X(t, \Gamma_0).$$

Then we can say $\Gamma$ moves by mean curvature if $X = X(t, x)$ satisfies for $x \in \Gamma_0$,

$$\frac{\partial}{\partial t} X = -H(X)n(X), \quad (2.9)$$

where $H$ is the mean curvature of $\Gamma(t)$ and $n(X)$ is its outward pointing normal.

For the following we will use the tangential gradient operator (or surface gradient) defined by

$$\nabla_\Gamma u = \nabla u - (\nabla u \cdot n)n.$$  

We note that $\nabla_\Gamma$ is the orthogonal projection of $\nabla$ onto the tangent space of $\Gamma$.

Now we will solve (2.9) using a finite element approach based on the following
time discrete approximation from [Deckelnick et al., 2005, Eqn 4.16].

\[ \int_{\Gamma^n} (U^{n+1} - \text{id}) \cdot \varphi + \Delta t \int_{\Gamma^n} \nabla_{\Gamma^n} U^{n+1} : \nabla_{\Gamma^n} \varphi = 0. \]

Here \(U^n\) parametrizes \(\Gamma(t^{n+1})\) over \(\Gamma^n := \Gamma(t^n)\), \(I\) is the identity matrix and \(\Delta t\) is the time step. Finally we apply a \(\theta\)-scheme to arrive at the following form.

\[ \int_{\Gamma^n} (U^{n+1} - \text{id}) \cdot \varphi + \Delta t \int_{\Gamma^n} (\theta \nabla_{\Gamma^n} U^{n+1} + (1 - \theta) I) : \nabla_{\Gamma^n} \varphi = 0. \]

where \(\theta \in [0, 1]\) is a discretization parameter.

We begin by setting up reference domain \(\Gamma_0 (\text{grid})\), and the space on \(\Gamma_0\) that describes \(\Gamma(t) (\text{space})\). From this we interpolate the non-spherical initial surface.
positions, and, then reconstruct space for the discrete solution on $\Gamma(t)$.

```python
grid = create.grid("ALUConform", "sphere.dgf",
    dimgrid=2, dimworld=3)
space = create.space("lagrange", grid,
    dimrange=grid.dimWorld, order=order)
positions = space.interpolate(lambda x: x * (1 + 0.5*math.sin(2*
    math.pi*x[0]*x[1])*math.cos(math.pi*
    x[2])), name="position")
surface = create.view("geometry", positions)
space = create.space("lagrange", surface,
    dimrange=surface.dimWorld, order=order)
solution = space.interpolate(lambda x: x, name="solution")
```

We set up the theta scheme with $\theta = 0.5$.

```python
theta = 0.5  # Crank-Nicholson
t = 0
end_time = 0.05
scheme.model.dt = 0.005
fig = pyplot.figure(figsize=(10, 10))
plot(solution, figure=(fig, 131+count%3), colorbar=False,
    gridLines="", triplot=True)
```

Now we solve the scheme in time. We first set up the initial time variables, then we plot the initial figure’s mesh, and finally we begin the loop, updating positions on each step and plotting the results side-by-side.
while t < end_time:
    scheme.solve(target=solution)
    t += scheme.model.dt
    count += 1
    positions.dofVector.assign(solution.dofVector)
    if count % 4 == 0:
        plot(solution, figure=(fig, 131+count%3), colorbar=False,
             triplot=True)
    pyplot.show()
pyplot.close('all')

Figure 2.17: The plot of the surface at three different timesteps

In case we start with a spherical initial surface, i.e., $\Gamma(0) = R_0 S^2$, the solution to the mean curvature flow equation is easy to compute:

$$
\Gamma(t) = R(t) S^2
$$

$$
R(t) = \sqrt{R_0^2 - 4t}
$$

We can use this to check that our implementation is correct. To do so we first define a function that computes the averaged radius of the surface.

def calcRadius(surface):
    # compute $R = \int_x |x| / \int_x 1$
    R = 0
    vol = 0
    for e in surface.elements:
        rule = geometry.quadratureRule(e.type, 4)
        R += rule.int(e.type, lambda x: x, 1)
        vol += rule.int(e.type, lambda x: 1)
    R /= vol

def calcRadius(surface):
    # compute $R = \int_x |x| / \int_x 1$
    R = 0
    vol = 0
    for e in surface.elements:
        rule = geometry.quadratureRule(e.type, 4)
        R += rule.int(e.type, lambda x: x, 1)
        vol += rule.int(e.type, lambda x: 1)
    R /= vol

56
for p in rule:
    geo = e.geometry
    weight = geo.volume * p.weight
    R += geo.toGlobal(p.position).two_norm * weight
    vol += weight
return R/vol

Now we test the convergence rate by solving over a loop, and calculating the error in terms of the difference between the above analytical solution and our calculated one. We plot this in a figure.

end_time = 0.1
scheme.model.dt = 0.02

import numpy as np
pyplot.figure()
pyplot.gca().set_xlim([0, end_time])
pyplot.gca().set_ylabel("error")
pyplot.gca().set_xlabel("time")

number_of_loops = 3
errors = np.zeros(number_of_loops)
totalIterations = np.zeros(number_of_loops, np.dtype(np.uint32))
gridSizes = np.zeros(number_of_loops, np.dtype(np.uint32))
for i in range(number_of_loops):
    positions.interpolate(lambda x: x * (R0/x.two_norm))
    solution.interpolate(lambda x: x)
    t = 0.
    R = calcRadius(surface)
    Rexact = math.sqrt(R0**2 - 4.*t)
    x = np.array([t])
    y = np.array([R - Rexact])
    iterations = 0
    while t < end_time:
        solution, info = scheme.solve(target=solution)
        # move the surface
        positions.dofVector.assign(solution.dofVector)
        # store some information about the solution process

7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
iterations += int( info["linear_iterations"] )

t += scheme.model.dt

R = calcRadius( surface )

Rexact = math.sqrt(R0*R0-4.*t)

x = np.append(x, [t])
y = np.append(y, [R - Rexact])

pyplot.plot(x, y, label='i = ' + str(i) if t >= end_time \
            else '')

pyplot.legend()

display.clear_output(wait=True)
display.display(pyplot.gcf())

errors[i] = abs(R - Rexact)
totalIterations[i] = iterations
gridSizes[i] = grid.size(2)

if i < number_of_loops - 1:
    grid.hierarchicalGrid.globalRefine(1)
scheme.model.dt /= 2.

Figure 2.18: Comparison of the error over time for varying levels of refinement

The estimated orders of convergence (EOCs) are calculated as shown.

eocs = np.log(errors[0:][:number_of_loops-1] / errors[1:]) / math.log(2.)

print(eocs)
Finally we organise this information into a table using pandas.

```python
try:
    import pandas as pd
    keys = {'size': gridSizes, 'error': errors,
            'eoc': np.insert(eocs, 0, None),
            'iterations': totalIterations}
    table = pd.DataFrame(keys, index=range(number_of_loops),
                          columns=['size', 'error', 'eoc',
                                   'iterations'])
    print(table)
except ImportError:
    print("pandas could not be used to show table with results")
```

<table>
<thead>
<tr>
<th>size</th>
<th>error</th>
<th>eoc</th>
<th>iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>318</td>
<td>0.001060</td>
<td>NaN</td>
</tr>
<tr>
<td>1</td>
<td>854</td>
<td>0.000599</td>
<td>0.823679</td>
</tr>
<tr>
<td>2</td>
<td>2065</td>
<td>0.000273</td>
<td>1.131173</td>
</tr>
</tbody>
</table>

### 2.6 Partitioned Grids

As another application of grid techniques, we look at a problem where we want to divide the grid into three regions. We do this using an 'adaptive' grid that allows for grid filters to be applied. We note that another way of creating multi-domain grids in DUNE is described in M"uthing and Bastian [2012] (though DUNE-FEMPY bindings are not yet available).

#### 2.6.1 Li-ion Battery Problem

In this example we provide an implementation of a Li-ion battery model described in Popov et al. [2011]. The aim is to model how the concentration of Lithium (Li) ions and the electric potential vary over time in a battery as it discharges.
Li-ion batteries are among the most popular types of rechargeable batteries available, having widespread use in phones, laptops and other portable devices. Due to the desire to improve their capacities, charge times and overall lifetime, they have been studied extensively (see e.g. Efendiev et al. [2013], Taralov et al. [2012], Taralov [2015] and Latz et al. [2011]).

In particular this model considers smaller scale behaviour of the system, by looking at an individual cell split into three parts, as described below.

We consider the following PDE system.

\[
\frac{\partial c}{\partial t} - \nabla \cdot (A(u) \nabla u) = 0, \quad \mbox{in } \Omega_a, \Omega_e \mbox{ and } \Omega_c, \\
- \nabla \cdot (B(u) \nabla u) = 0, \quad \mbox{in } \Omega_a, \Omega_e \mbox{ and } \Omega_c,
\]

where \( u = (c, \phi) \), i.e. the concentration and electric potential, \( \nabla u \) is the Jacobian, and \( A(u), B(u) \) are defined as

\[
A(u) = \left( D_e + \frac{RT t_+ \kappa}{F^2 c}, \frac{t_+ \kappa}{F} \right), \\
B(u) = \left( \frac{RT \kappa}{F c}, \kappa \right),
\]

where \( F = 96485 \ \text{C mol}^{-1} \) is the Faraday constant, \( R = 8.314 \ \text{J mol}^{-1} \ \text{K}^{-1} \) is the gas constant and \( T = 300 \ \text{K} \). \( D_e, \kappa \) and \( t_+ \) are constants that depend on the domain and are given as

<table>
<thead>
<tr>
<th>Domain</th>
<th>( D_e )</th>
<th>( \kappa )</th>
<th>( t_+ )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anode</td>
<td>3.9x10^{-10}</td>
<td>1.0</td>
<td>0</td>
</tr>
<tr>
<td>Electrolyte</td>
<td>7.5x10^{-7}</td>
<td>0.002</td>
<td>0.2</td>
</tr>
<tr>
<td>Cathode</td>
<td>1.0x10^{-9}</td>
<td>0.038</td>
<td>0</td>
</tr>
</tbody>
</table>

The domain is given by a rectangle \( \Omega \) split into three parts. \( \Omega_a \) on left (anode), \( \Omega_e \) in middle (electrolyte), \( \Omega_c \) on right (cathode). The inner boundaries are \( \Gamma_a = \bar{\Omega}_a \cap \bar{\Omega}_e, \)
\( \Gamma_c = \bar{\Omega}_e \cap \bar{\Omega}_c \). The outer boundary is \( \Gamma_{out} \). Note that \( u \) is considered discontinuous across the inner boundaries, therefore we denote \( u \) separately as \( u_a, u_e \) and \( u_c \).
each respective domain.

Figure 2.19: The domain, a cell split into three parts

For the outer boundary $\Gamma_{out}$ we set the Neumann conditions (no flux),

\[
(A(u)\nabla u) \cdot n = 0, \\
(B(u)\nabla u) \cdot n = 0,
\]

where $n$ is the unit normal pointing out the domain. Additionally on the inner boundaries $\Gamma_a$ we set the Neumann conditions,

\[
(A(u_a)\nabla u_a) \cdot n = -(A(u_e)\nabla u_e) \cdot n = N(u_a, u_e), \\
(B(u_a)\nabla u_a) \cdot n = -(B(u_e)\nabla u_e) \cdot n = J(u_a, u_e),
\]

(note that the negative sign on the $\Omega_e$ side accounts for the fact the normal is inverted) and similarly for $\Gamma_c$, where $N$ and $J$ are defined on $\Gamma_a$ (and equivalently $\Gamma_c$) as follows.

\[
J(u_a, u_e) = k \left( \frac{c_e}{c^0_e} \right)^{\alpha_a} \left( \frac{c_a}{c^0_a} \right)^{\alpha_a} \left( 1 - \frac{c_a}{c_{a,max}} \right)^{\alpha_c} \left( \exp \left( \frac{\alpha_a F}{RT} \eta_a \right) - \exp \left( -\frac{\alpha_c F}{RT} \eta_a \right) \right), \\
N(u_a, u_e) = \frac{J(u_a, u_e)}{F},
\]

61
where \( \eta_a = \phi_a - \phi_e - U_{a,0} \) (and correspondingly \( \eta_c = \phi_c - \phi_e - U_{c,0} \)) and \( \alpha_a + \alpha_c = 1 \) are anodic and cathodic weightings respectively. Additionally \( c^0 \) (the initial condition for \( c \)), \( c_{\text{max}} \) and \( U_0 \) are defined on each domain by

<table>
<thead>
<tr>
<th>Domain</th>
<th>( c^0 )</th>
<th>( c_{\text{max}} )</th>
<th>( U_0 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anode</td>
<td>0.002639</td>
<td>0.02639</td>
<td>0</td>
</tr>
<tr>
<td>Electrolyte</td>
<td>0.001</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Cathode</td>
<td>0.020574</td>
<td>0.02286</td>
<td>0.001</td>
</tr>
</tbody>
</table>

We will now write the weak form of the above PDE. Introducing a test function \( v = (v_0, v_1) \) and integrating over the domain, we get,

\[
\int_{\Omega} \frac{\partial c}{\partial t} v_0 - \nabla \cdot (A(u) \nabla u) v_0 \, dV = 0,
\]

\[
- \int_{\Omega} \nabla \cdot (B(u) \nabla u) v_1 \, dV = 0.
\]

We then apply an implicit time discretisation, denoting the data from the previous time step by \( u^0 = (c^0, \phi^0) \), the new time step by \( u^1 = (c^1, \phi^1) \) and the size of the time step by \( \Delta t \). This gives us the following.

\[
\int_{\Omega} c^0 v_0 \, dV = \int_{\Omega} c^1 v_0 - \Delta t \nabla \cdot (A(u^1) \nabla u^1) v_0 \, dV,
\]

\[
- \int_{\Omega} \nabla \cdot (B(u^1) \nabla u^1) v_1 \, dV = 0.
\]

We now apply Green’s identity to get rid of the divergence terms. Note that for the following, we will separate the PDE into 3 equations for each part of the domain, since the boundaries are different in each case. For \( \Omega_a \), we get the following.

\[
\int_{\Omega_a} c^0_a v_0 \, dV = \int_{\Omega_a} c^1_a v_0 + \Delta t (A(u^1_a) \nabla u^1_a) \cdot \nabla v_0 \, dV - \int_{\Gamma_a} \Delta t (A(u^1_a) \nabla u^1_a) \cdot \mathbf{n} v_0 \, dS,
\]

\[
\int_{\Omega_a} (B(u^1_a) \nabla u^1_a) \cdot \nabla v_1 \, dV - \int_{\Gamma_a} \Delta t (B(u^1_a) \nabla u^1_a) \cdot \mathbf{n} v_1 \, dS = 0.
\]
For $\Omega_e$,
\[
\int_{\Omega_e} c^e_0 v_0 \, dV = \int_{\Omega_e} c^e_1 v_0 + \Delta t(A(u^e_1) \nabla u^e_1) \cdot \nabla v_0 \, dV - \int_{\Gamma_e} \Delta t(A(u^1) \nabla u^1) \cdot \mathbf{n}_0 \, dS,
\]
\[
\int_{\Omega_e} (B(u^e_1) \nabla u^e_1) \cdot \nabla v_1 \, dV - \int_{\Gamma_e} \Delta t(B(u^1) \nabla u^1) \cdot \mathbf{n}_1 \, dS = 0.
\]
And for $\Omega_e$,
\[
\int_{\Omega_e} c^e_0 v_0 \, dV = \int_{\Omega_e} c^e_1 v_0 + \Delta t(A(u^e_1) \nabla u^e_1) \cdot \nabla v_0 \, dV - \int_{\Gamma_e} \Delta t(A(u^1) \nabla u^1) \cdot \mathbf{n}_0 \, dS,
\]
\[
\int_{\Omega_e} (B(u^e_1) \nabla u^e_1) \cdot \nabla v_1 \, dV - \int_{\Gamma_e} \Delta t(B(u^1) \nabla u^1) \cdot \mathbf{n}_1 \, dS = 0.
\]
Finally we apply our boundary conditions for the integrands with $dS$. For $\Omega_a$,
\[
\int_{\Omega_a} c^a_0 v_0 \, dV = \int_{\Omega_a} c^a_1 v_0 + \Delta t(A(u^a_1) \nabla u^a_1) \cdot \nabla v_0 \, dV - \int_{\Gamma_a} \Delta t N(u^a_1, u^a_1) v_0 \, dS,
\]
\[
\int_{\Omega_a} (B(u^a_1) \nabla u^a_1) \cdot \nabla v_1 \, dV - \int_{\Gamma_a} J(u^a_1, u^a_1) v_1 \, dS = 0.
\]
For $\Omega_a$,
\[
\int_{\Omega_a} c^a_0 v_0 \, dV = \int_{\Omega_a} c^a_1 v_0 + \Delta t(A(u^a_1) \nabla u^a_1) \cdot \nabla v_0 \, dV
\]
\[+ \int_{\Gamma_a} \Delta t N(u^a_1, u^a_1) v_0 \, dS + \int_{\Gamma_e} \Delta t N(u^e_1, u^e_1) v_0 \, dS,
\]
\[
\int_{\Omega_a} (B(u^a_1) \nabla u^a_1) \cdot \nabla v_1 \, dV + \int_{\Gamma_a} J(u^a_1, u^a_1) v_1 \, dS + \int_{\Gamma_e} J(u^e_1, u^e_1) v_1 \, dS = 0.
\]
And for $\Omega_a$,
\[
\int_{\Omega_a} c^a_0 v_0 \, dV = \int_{\Omega_a} c^a_1 v_0 + \Delta t(A(u^a_1) \nabla u^a_1) \cdot \nabla v_0 \, dV - \Delta t \int_{\Gamma_e} N(u^e_1, u^e_1) v_0 \, dS,
\]
\[
\int_{\Omega_a} (B(u^a_1) \nabla u^a_1) \cdot \nabla v_1 \, dV - \int_{\Gamma_e} J(u^e_1, u^e_1) v_1 \, dS = 0.
\]

We will now look at an implementation of the above in DUNE-FEMPY. First, let us import the necessary Python modules.

```python
1 import math
2 from ufl import *
```
import dune.ufl
import dune.fem

import dune.create as create
from dune.fem.view import filteredGridView

Let us define the parameters for the problem.

dune.fem.parameter.append("parameter")

# general parameters
dimDomain = 2
dimRange = 2
order = 1  # order of FE space
numRefines = 1  # number of refinements of initial grid
timeStep = 50  # size of timeStep
maxIter = 20  # max number of solver iterations

# problem parameters (from battery paper)
R = 8.314
T = 300
F = 96485
t_plus = [0, 0.2, 0]
kappa = [1.0, 0.002, 0.038]
D_e = [3.9e-10, 7.5e-7, e-9]
c_init = [0.002639, 0.001, 0.020574]  # initial value for c
phi_init = [0, 0, 0]  # initial value for potential
c_max = [0.02639, None, 0.02286]
U_0 = [0, None, 0.001]
alpha_a = 0.5
alpha_c = 1 - alpha_a

In the usual way we define the variables in UFL.

# define u^1 = (c^1, phi^1) and v = (v_0, v_1)
space = dune.ufl.Space(dimDomain, dimRange)
u = TrialFunction(space)
v = TestFunction(space)
Let us define the PDE and boundary conditions, starting with variables that we might want to modify.

```python
# define A_1 and A_2 in PDE (for the Id, a = 0, e = 1, c = 2)
def A1(Id):
    return D_e[Id] + R*T/(F**2)*t_plus[Id]**2*kappa[Id]/u[0]
def A2(Id):
    return kappa[Id]*t_plus[Id]/F

# define B_1 and B_2 in PDE
def B1(Id):
    return R*T/F*t_plus[Id]*kappa[Id]/u[0]
def B2(Id):
    return kappa[Id]

# define Neumann boundary term J for inner boundaries
def J(uElec, uSolid):
    J1 = (uElec[0]/c_init[1])**alpha_a
    J2 = (uSolid[0]/c_init[0])**alpha_a
    J3 = (1 - uSolid[0]/c_max[0])**alpha_c
    J4 = (exp(alpha_a*F/(R*T)*(uSolid[1] - uElec[1] - U_0[0])) -
         exp(-alpha_c*F/(R*T)*(uSolid[1] - uElec[1] - U_0[0])))
    return J1*J2*J3*J4

# define dirichlet conditions on the left and right boundaries
tmp_a = 0.000951 # these should be removed
tmp_c = 0.018454#
diric_a = as_vector([tmp_a, 2.5e-8])
diric_c = as_vector([tmp_c, 1.9e-2])
```
We continue defining the PDE and boundary conditions, this time with parts of the framework that should mostly remain the same.

```python
# define J, N in Omega_a and Omega_c
J_s = J(u_e, u)
N_s = J_s/F

# define J, N in Omega_e on Gamma_a and Gamma_c respectively
J_ea = J(u, u_a)
N_ea = J_ea/F
J_ec = J(u, u_c)
N_ec = J_ec/F

# define the bilinear form's explicit part using B1, B2
def eq_ex(Id):
    ex = inner(un[0], v[0])*dx
    ex += inner(B1(Id)*grad(u[0])
                + B2(Id)*grad(u[1]), grad(v[1]))*dx
    return ex

# define the implicit part using A1, A2
def eq_im(Id):
    im = (inner(u[0], v[0]))*dx
    im += dt*inner(A1(Id)*grad(u[0])
                   + A2(Id)*grad(u[1]), grad(v[0]))*dx
    return im

# let's combine the bilinear forms with the BCs in each domain
a_ex = eq_ex(0) - J_s*v[1]*ds(4)

# same for Omega_e
e_ex = eq_ex(1) + J_ec*v[1]*ds(5)
e_im = eq_im(1) + dt*N_ea*v[0]*ds(3) + dt*N_ec*v[0]*ds(5)

# and Omega_c
c_ex = eq_ex(2) - J_s*v[1]*ds(4)
c_im = eq_im(2) - dt*N_s*v[0]*ds(4)
```

The remaining code involves setting up the FEM and is mostly independent of the problem parameters, thus should not need to be changed.
Let us construct the three separate grids using a grid filter. We label the three domains by 3, 4 and 5 to correspond with the inner BCs defined above.

```python
def filter(e):
    if e.geometry.center[0] <= 0.2:
        return 3
    elif 0.2 <= e.geometry.center[0] <= 0.8:
        return 4
    elif 0.8 <= e.geometry.center[0]:
        return 5

unitcube = 'unitcube-' + str(dimDomain) + 'd. dgf'
grid = create.view("adaptive", create.grid("ALUCube", unitcube, 
dimgrid=dimDomain))
grid.hierarchicalGrid.globalRefine(numRefines)
anode = filteredGridView(grid, filter, 3)
electrolyte = filteredGridView(grid, filter, 4)
cathode = filteredGridView(grid, filter, 5)
```

We construct the FE spaces and the solutions.

```python
space_a = create.space("Lagrange", anode, dimrange=dimRange, 
    order=order)
space_e = create.space("Lagrange", electrolyte, dimrange=dimRange, 
    order=order)
space_c = create.space("Lagrange", cathode, dimrange=dimRange, 
    order=order)

solution_a = space_a.interpolate(lambda x: [c_init[0], 
    phi_init[0]], name="solution_a")
solution_a_n = solution_a.copy()
solution_a_n.assign( solution_a )
solution_e = space_e.interpolate(lambda x: [c_init[1], 
    phi_init[1]], name="solution_e")
solution_e_n = solution_e.copy()
solution_e_n.assign( solution_e )
solution_c = space_c.interpolate(lambda x: [c_init[2], 
    phi_init[2]], name="solution_c")
solution_c_n = solution_c.copy()
```
We construct the models and schemes.

```python
# omega_a
model_a = create.model("split", anode, a_ex == a_im,
                       dirichlet={6: diric_a},
                       coefficients={u_e: solution_e_n, un: solution_a_n})
model_a.setConstant(dt, timeStep)
scheme_a = create.scheme("galerkin", model_a, space_a)

# omega_e
model_e = create.model("split", electrolyte, e_ex == e_im,
                       coefficients={u_a: solution_a_n, u_c: solution_c_n,
                                     un: solution_e_n})
model_e.setConstant(dt, timeStep)
scheme_e = create.scheme("galerkin", model_e, space_e)

# omega_c
model_c = create.model("split", cathode, c_ex == c_im,
                       dirichlet={7: diric_c},
                       coefficients={u_e: solution_e_n, un: solution_c_n})
model_c.setConstant(dt, timeStep)
scheme_c = create.scheme("galerkin", model_c, space_c)
```

We define the method for plotting the solution. We do this using matplotlib and by plotting each solution to its own domain. We also calculate a global_max and global_min of all solutions to create the colour plot.

```python
from numpy import amin, amax, linspace
import matplotlib
from matplotlib import pyplot
from IPython import display
matplotlib.rcParams.update({'font.size': 10})
matplotlib.rcParams['figure.figsize'] = [10, 5]

def matplot(grid, solution, sol2, sol3, a=False):
    triangulation = grid.triangulation()
    for p in range(2):
```
```python
import matplotlib.pyplot as plt

# Setup subplot
pyplot.subplot(121 + p)
plt.gca().set_aspect('equal')
plt.gca().locator_params(tight=True, nbins=4)
data = solution.pointData()
data2 = sol2.pointData()
data3 = sol3.pointData()
global_min = min(amin(data[:,p]), amin(data2[:,p]),
                   amin(data3[:,p])) - 1e-4
global_max = max(amax(data[:,p]), amax(data2[:,p]),
                  amax(data3[:,p])) + 1e-4
if global_min != global_max:
    levels = linspace(global_min, global_max, 256)
    plt.tricontourf(triangulation, data[:,p],
                    cmap=plt.cm.rainbow, levels=levels)
else:
    plt.tricontourf(triangulation, data[:,p],
                    cmap=plt.cm.rainbow)
if a == True:
    plt.colorbar(shrink=0.725)

Finally we start the solving process over a loop. We plot the initial solution
and the result after 20 steps, and we save each step to a paraview file.
```

```python
anode.writeVTK("battery_anode_", pointdata=[solution_a], number=0)
electrolyte.writeVTK("battery_electrolyte_", pointdata=[solution_e],
                      number=0)
cathode.writeVTK("battery_cathode_", pointdata=[solution_c],
                  number=0)
```

```python
for i in range(1, 20):
    scheme_a.solve(target=solution_a)
    scheme_e.solve(target=solution_e)
    scheme_c.solve(target=solution_c)
solution_a_n.assign(solution_a)
```

69
solution_e_n.assign(solution_e)
solution_c_n.assign(solution_c)
anode.writeVTK("battery_anode_",
    pointdata=[solution_a], number=i)
electrolyte.writeVTK("battery_electrolye_",
    pointdata=[solution_e], number=i)
cathode.writeVTK("battery_cathode_",
    pointdata=[solution_c], number=i)
matplot(anode, solution_a, solution_e, solution_c, a=True)
matplot(electrolyte, solution_e, solution_a, solution_c)
matplot(cathode, solution_c, solution_a, solution_e)
display.display(pyplot.gcf())

Figure 2.20: The initial plot of $c$ and $\phi$

Figure 2.21: The plot after the final timestep
2.7 Translating Python Code to C++

Having looked at some of the functionality available for solving various FEM problems, we now shift our focus towards more in-depth features that concern efficiency and C++ development. Here we consider the idea of moving sections of Python code over to C++ for efficiency. A key aspect of the design of DUNE-FEMPY has been about keeping the structure of the C++ code in the Python code’s design, to the point where translating between the two is relatively painless. In particular this allows for rapid prototyping of methods in Python with its relative ease of use, after which code can be ported to C++ for efficiency if necessary in large-scale computation.

Here we will demonstrate this translation process, and additionally provide comparisons for the difference in efficiency timewise. We will examine the function used for calculating the averaged radius of a surface used in the mean curvature flow example from section 2.5.

Code Listing 2.31: A Pythonic function for calculating the radius of a surface

```python
def calcRadius(surface):
    # compute R = int_x |x| / int_x 1
    R = 0
    vol = 0
    for e in surface.elements:
        rule = geometry.quadratureRule(e.type, 4)
        for p in rule:
            geo = e.geometry
            weight = geo.volume * p.weight
            R += geo.toGlobal(p.position).two_norm * weight
            vol += weight
    return R/vol
```

As a relatively simple example, this code is not particularly slow in Python, however the existence of callbacks inside a looped statement are not insignificant. Now let us look at a C++ translation of the above code.

Code Listing 2.32: The C++ version of the calcRadius function
```cpp
#include <dune/geometry/quadraturerules.hh>

template< class Surface >
double calcRadius( const Surface &surface )
{
  double R = 0.;
  double vol = 0.;
  for( const auto &entity : elements( surface ) )
  {
    const auto &rule = Dune::QuadratureRules<double, 2>::rule(entity.type(), 4);
    for ( const auto &p : rule )
    {
      const auto geo = entity.geometry();
      const double weight = geo.volume() * p.weight();
      R += geo.global(p.position()).two_norm() * weight;
      vol += weight;
    }
  }
  return R/vol;
}
```

We note that we take advantage of C++11 features such as `auto` and range based for loops to keep a similar structure to the Python code.

Supposing we save the above as `radius hh`, we can then call it in a Python script and use it like a regular function as follows.

```
from dune.generator import algorithm
calcRadius = algorithm.load('calcRadius', 'radius hh', surface)
```

Doing this, we can quite easily swap between the two versions and compare the runtime of the solve method. Specifically, we test the runtime of the mean curvature flow example with the original Python version of `calcRadius` and compare it to the runtime with the above substitution. We show these results for a relatively large number of elements, as shown in figure 2.22.
What we see is that the C++ version is roughly 18% faster. On a small scale this is not a significant change, but it could be potentially worth it for a particularly long-running simulation. Naturally the more of the code that is written in C++, the faster it will be overall, though whether it is justifiable to do this depends on a case-by-case basis.

2.8 Virtualization

One final topic we want to discuss is virtualization, by which we mean the use of virtual classes and functions in C++ to abstractly represent objects such as grids and spaces.

In the development of the Python bindings for DUNE-PYTHON, the decision was made to avoid introducing such a virtual layer when exporting classes from C++ to Python or vice versa; the reason being that it would introduce additional code maintenance and more importantly perhaps lead to loss in performance when a C++ object is passed through Python back into C++. In this case code optimization steps like inlining or loop unrolling could not be utilized to their full potential.
Take as an example a discrete function which is constructed using \texttt{df = space.interpolate([0], name="df")}. The call to \texttt{interpolate} goes back to the corresponding function in DUNE-FEM and returns an instance of the discrete function. To store the solution to a PDE problem in \texttt{df}, the \texttt{solve} method on a \texttt{scheme} is called. While executing \texttt{scheme.solve(target=df)} the discrete function instance is passed back to another DUNE-FEM function. If \texttt{df} were virtualized (i.e. type erased) in either of the two steps, i.e. when passed to or from Python, then the \texttt{solve} method could not work with the same efficiency as when used in a pure C++ environment. The number of degrees of freedom, local structure, etc. of the discrete function would only be known as dynamic properties, making code optimization by the compiler or the DUNE developer implementing the \texttt{solve} method more difficult or even impossible. Note that virtualizing the discrete function for example, would almost certainly also require virtualization of the underlying discrete function spaces (with mapper and basis function set), and the underlying grid view (with its iterators). The cumulative effect of this would be quite severe on performance.

To avoid this issue, no type erasure is carried out when an object is passed into Python. So in the above example the call to \texttt{interpolate} returns an object which still contains the full type information of the underlying DUNE-FEM function. This approach leads to compilation overhead the first time a new type of discrete function is used since a new Python module needs to be generated. But this overhead occurs only the first time the discrete function is used during the development of a project and is thus negligible. Since no type erasure has occurred, any DUNE object can now be passed back to it. The \texttt{solve} method on the scheme is exported in such a way that the target argument has to be of the same discrete function type that was defined by the \texttt{storage} argument provided during the space construction. Consequently a scheme over a given space (e.g. a Lagrange space of a fixed order using an \texttt{istl} storage) will only accept one type of discrete function as \texttt{target} argument for its solve method. As described before the advantage of this is that the full static type information is available at the cost of more compilation requirements when changes (e.g. to the storage back end) are made.
There are a few exceptions to the above rule, where Python objects passed as arguments to C++ functions undergo type erasure if their type does not match the exact type of the arguments of that function.

An example is the `__call__` method on an `operator`. When calling `op(arg, dest)`, the destination parameter (`dest`) has to be of the correct discrete function type, but for the argument parameter (`arg`) it can make sense to allow for a wide range of grid functions, e.g. an exact solution given by a UFL expression or a different type of discrete function. In many cases the implementation of the operator does not require the argument to even be discrete since only the evaluation of `arg` at quadrature points is required; in this case any grid function is a valid argument. On the C++ side the operator call is simply implemented as a template method on the operator class with the signature

```cpp
1 template <class GF> Operator::operator()(const GF &arg, typename Operator::DiscreteFunction &dest);
```

We note that it is not possible to export a template method to Python without fixing all of its arguments. Since an optimized version of such a method is often implemented for the case that `arg` is of the same type as `dest`, the default method that will always be exported to Python has `GF=Operator::DiscreteFunction`. In addition, a second version is exported where `GF=VirtualizedGridFunction<...>`, which is a type erased implementation of a grid function. Any grid function exported to Python (e.g. UFL expressions, discrete function etc.) will implicitly convert to a `VirtualizedGridFunction` so that `op(arg, dest)` can be used in Python even in the case that `arg` is not of the same type as `dest`. Optimal code is still produced in the case where both parameters are of the same type.

A second use of type erasure where objects are passed back to C++ occurs when an `operator` or `scheme` is constructed from a given model. Since the development of a new model can involve repeated changes being made to it (e.g. its underlying UFL form) we aimed to avoid the situation of each change requiring a recompilation of the `operator` or `scheme`. To this end the model is virtualized when it is passed to the constructor of the `operator` or `scheme` class. Consequently, these
classes only depend on some type information like the underlying type of the grid view and the range dimension of the model but not on the actual details of the weak form. The consequence of this approach is that evaluating some part of the form introduces a virtual function call.
Chapter 3

Nonvariational PDEs

3.1 Definition and Notation

We start this chapter by concretely defining our problem for the linear case and providing some notation for the following sections.

Let the computational domain for our finite element method be \( \Omega \subset \mathbb{R}^d \).

Then let us once again state the problem in general terms. For \( u \in W := H^2(\Omega) \cap H^1_0(\Omega) \), we would like to solve

\[
\begin{align*}
-A : D^2 u &= f, & \text{in } \Omega, \\
    u &= 0, & \text{on } \partial \Omega.
\end{align*}
\]

(3.1)

Here \( D^2 u \) is the Hessian of \( u \), \( f \in L^2 \) is a real-valued prescribed function, \( M : N = \sum_{i,j} M_{ij} N_{ij} \) is the Frobenius inner product and \( A(x) \in \text{Sym}(\mathbb{R}^{d \times d}) \cap C^0(\Omega^{d \times d}) \) is a coefficient matrix that is elliptic in the following sense.

**Definition 3.1.1 (Ellipticity of Symmetric Matrices).** We say \( A \) is elliptic on \( \text{Sym}(\mathbb{R}^{d \times d}) \) if, for each \( M \in \mathbb{R}^{d \times d} \), there exist \( \Lambda \geq \lambda > 0 \) such that

\[
\lambda \sup_{|\xi|=1} |N\xi| \leq A(M + N) - A(M) \leq \Lambda \sup_{|\xi|=1} |N\xi|, \quad \forall N \in \text{Sym}(\mathbb{R}^{d \times d}).
\]

Additionally for the next theorem we will use the following definition.
**Definition 3.1.2** (Hölder Domain). A domain \( \Omega \subset \mathbb{R}^d \) is \( C^{k,\alpha} \), if each point on the boundary \( \partial \Omega \) has a neighbourhood in which \( \partial \Omega \) can be represented by a function in the Hölder space \( C^{k,\alpha} \), after a change of coordinates.

Then we have the following theorem for the existence of a strong solution from [Gilbarg and Trudinger, 2015, Thm 9.15].

**Theorem 3.1.1** (Existence of a strong solution to (3.1)). Let \( \Omega \subset \mathbb{R}^d \) be a \( C^{1,1} \) domain. Let \( A \in \text{Sym}(\mathbb{R}^{d\times d}) \cap C^0(\Omega)^{d\times d} \) be an elliptic matrix and \( f \in L^2(\Omega) \). Then the equation

\[
-A : D^2 u = f, \quad \text{in } \Omega, \\
u = 0, \quad \text{on } \partial \Omega.
\]

has a unique solution \( u \in H^2(\Omega) \cap H^1_0(\Omega) \). There also exists a constant \( C \) independent of \( u \) such that

\[
\|u\|_2 \leq C\|f\|.
\]

where \( \| \cdot \|_k \) denotes the \( H_k(\Omega) \) norm.

Now in future sections we will be working in the discrete case of this problem, therefore to streamline the analysis, let us define the notation we will be using here.

Regarding the domain, let \( \mathcal{T} \) denote a triangulation of \( \Omega \), and \( \mathcal{E} \) denote the edges of the elements (with \( \mathcal{E}_0 := \mathcal{E} \setminus \partial \Omega \)). For measuring the refinement of the mesh, we define \( h = \max_{K \in \mathcal{T}} h_K \) to be the mesh size (where \( h_K \) is the diameter of element \( K \)).

In the future we will be working in a conforming finite element space \( V_h \), for which the discretized problem is defined. Let \( P^k(\mathcal{T}) \) denote the space of piecewise polynomials of degree \( k \) over \( \mathcal{T} \), i.e.

\[
P^k(\mathcal{T}) = \{ \phi : \phi|_K \in P^k(K) \}.
\]

Then we let \( V_h = C^0(\Omega) \cap P^k(\mathcal{T}) \). We remark that this space is the typical choice used for Lagrange finite elements.
Now as a lot of the upcoming methods use discontinuous Galerkin (DG) techniques, we shall also define some of these concepts here.

**Definition 3.1.3 (Jumps and Averages).** Consider an edge $e \in \mathcal{E}$ (if $e \in \mathcal{E}_0$ then let it be between two elements $K_1$ and $K_2$). We define the **jump** and **average** of $v \in L^2(\Omega)$ on $e$ respectively as

\[
\llbracket v \rrbracket = \begin{cases} 
v|_{K_1} n_{K_1} + v|_{K_2} n_{K_2}, & e \in \mathcal{E}_0, \\
v \cdot n, & e \subset \partial \Omega, \end{cases}
\]

\[
\{v\} = \begin{cases} \frac{1}{2} (v|_{K_1} + v|_{K_2}), & e \in \mathcal{E}_0, \\
v, & e \subset \partial \Omega, \end{cases}
\]

where $n_K$ is the outward pointing normal to $K$. Additionally for a vector $v \in L^2(\Omega)^d$ we have the following natural extensions to the definitions.

\[
\llbracket v \rrbracket = \begin{cases} \n abla \cdot (v|_{K_1} n_{K_1} + v|_{K_2} n_{K_2}), & e \in \mathcal{E}_0, \\
v \cdot n, & e \subset \partial \Omega, \end{cases}
\]

\[
\{v\} = \begin{cases} \frac{1}{2} (v|_{K_1} + v|_{K_2}), & e \in \mathcal{E}_0, \\
v, & e \subset \partial \Omega, \end{cases}
\]

And we have a similar version of the jump for the outer product,

\[
\llbracket v \rrbracket = \begin{cases} \n abla \times (v|_{K_1} n_{K_1} + v|_{K_2} n_{K_2}), & e \in \mathcal{E}_0, \\
v \times n, & e \subset \partial \Omega. \end{cases}
\]

Finally we note there exists an alternate form of the jump sometimes used in the
literature, which we define now.

\[ \|v\|_0 = \begin{cases} v|_{K_1} - v|_{K_2}, & e \in \mathcal{E}_0, \\ v, & e \subset \partial \Omega. \end{cases} \]

**Remark.** We note that the \( \| \cdot \| \) definition returns a vector when applied to a scalar, and a scalar when applied to a vector, whilst the \( \| \cdot \|_0 \) definition always returns a variable of the same dimension. Thus it is important to keep this distinction in mind.

In the discontinuous Galerkin case, note that we use a discontinuous version of \( V_h \) for our space, which we denote by \( V_{DG} \).

One more concept we would like to mention used in some nonvariational approaches (e.g. Dedner and Pryer [2013] and Wang and Wang) is to discretize the Hessian as follows.

**Definition 3.1.4 (Finite Element Hessian).** We define the finite element Hessian \( H[u] \) to be a unique element of \( V_{DG}^{d \times d} \) such that for all \( \varphi \in V_h \),

\[ \int_\Omega H[u] \varphi \, dx = -\int_\Omega \nabla_h u \otimes \nabla_h \varphi \, dx + \int_\mathcal{E} \|u\| \otimes \{\nabla_h \varphi\} + \|\varphi\| \otimes \{\nabla_h u\} \, ds, \quad (3.2) \]

where \( \nabla_h = (D_h)^T \) is the elementwise gradient. In section 3.5 we will return to this concept and provide a full derivation.

### 3.2 Existing Methods

Now that we have provided a background for the mathematical concepts of (3.1) in section 3.1, we start our analysis of nonvariational problems by considering the methods from the literature that have been developed to solve this kind of problem numerically.

Before we go into the specifics of these approaches, let us briefly state the
variational method we use in the case when \( A \) is assumed to be smooth. We introduce this for the purpose of a benchmark to compare nonvariational methods to in later sections.

**Example 3.2.1 (Variational).** We begin by rewriting \( A : D^2 u \) into divergence form.

\[
A : D^2 u = \nabla \cdot (A \nabla u) - (\nabla \cdot A) \nabla u.
\]

Substituting this form into (3.1), multiplying by \( v \) and integrating by parts, we obtain the following bilinear form.

\[
\int_\Omega (A \nabla u \cdot \nabla v - (\nabla \cdot A) \cdot \nabla uv) \, dx - \int_{\partial \Omega} A \nabla u \cdot n v \, ds = \int_\Omega f v \, dx.
\]

Lastly we incorporate the boundary condition \( u = 0 \) by adding the following term which is a weak implementation of Dirichlet boundary conditions.\(^1\)

\[
\beta h^{-1} \int_{\partial \Omega} u v \, ds, \quad \beta > 0,
\]

where \( \beta > 0 \) is a constant. This results in the following bilinear form for the variational approach.

\[
\int_\Omega (A \nabla u \cdot \nabla v + (\nabla \cdot A) \cdot \nabla uv) \, dx + \int_{\partial \Omega} (\beta h^{-1} u v - A \nabla u \cdot n v) \, ds = \int_\Omega f v \, dx,
\]

With the variational method stated, we shall now consider the nonvariational approaches.

**Example 3.2.2 (Pryer).** Let us consider the numerical method used in Lakkis and Pryer [2010], which is equivalent to finding \( u_h \in V_h \) such that

\[
- \int_\Omega A : \hat{H}[u_h] \varphi_h \, dx = \int_\Omega f \varphi_h \, dx, \quad \forall \varphi_h \in V_h,
\]

\(^1\)This is an alternative to a strong Dirichlet BC implementation which in the numerical implementation requires one to manually set columns in the system matrix to zero. As we will later formulate the problem as a saddle point problem, for comparison reasons it is easier to use the same penalty term for all methods to enforce boundary conditions.
where $\hat{H}[u_h] \in [V_h]^d \times d$ satisfies

$$
\int_{\Omega} \hat{H}[u_h] \varphi_h \, dx = - \int_{\Omega} \nabla u_h \otimes \nabla \varphi_h \, dx + \int_{\partial \Omega} \nabla u_h \otimes n \varphi_h \, ds.
$$

We note that, assuming strong treatment of the Dirichlet boundary conditions, this form of $\hat{H}[u]$ is equivalent to (3.2) in the case of Lagrange finite elements (i.e. continuity across elements is assumed so $\|v\| = 0$ on $E_0$).

A-priori error estimates are not formulated in this paper, and instead quantitative results are the focus\(^2\). In terms of convergence rates, they show (for a sufficiently smooth solution and) for $P^k$ elements that $\|u - u_h\| = O(h^{k+1})$ and $|u - u_h|_1 = O(h^k)$. We note that these are the usual results observed in FEMs for variational problems, and among the upcoming examples, we will see that convergence rates of this order are also typical for nonvariational problems of the form (3.1).

**Example 3.2.3 (NVDG).** A discontinuous Galerkin version of the above method was later derived in Dedner and Pryer [2013].

$$
- \int_{\Omega} A : H[u_h] \varphi_h \, dx + \int_{E} \sigma h^{-1} [u_h] \cdot [\varphi_h] \, ds = \int_{\Omega} f \varphi_h \, dx, \quad \forall \varphi_h \in V_h. \tag{3.6}
$$

where $H[u]$ is defined as in (3.2).

In comparison to (3.5), the DG formulation leads to a slightly more complex form of the finite element Hessian, and there is the addition of a stabilization term.

The method is implemented in DUNE, and numerically the same convergence rates are observed as in example 3.2.2. Additionally, they prove the analytical result that for sufficiently smooth $A$ and $u$ that

$$
\|u - u_h\|_{DG,1} \leq C \left( h^k |u|_{k+1} + h^{k+1} |u|_{k+3} \right),
$$

\(^2\)The results were computed in MATLAB with a GMRES (generalized minimal residual method) used for the linear solver.
where \( k \) is the polynomial order and \( \|uh\|_{DG,1} \) is the broken norm defined by

\[
\|uh\|_{DG,1}^2 := \|\nabla hu_h\|^2 + h^{-1}\|u_h\|_E^2.
\]

We note that in theory the smoothness conditions limit the usability of this result, although they still show numerically that optimal convergence holds even for a nondifferentiable operator.

Given that this form of the method can be considered an advancement to example 3.2.2, we will focus mostly on this version for comparative purposes.

**Example 3.2.4 (Feng).** We now consider a method that takes a similar form to the above, but without the finite element Hessian. In Feng et al. [2015] they used the following method.

\[
- \int_{\Omega} A : D^2_h uh \phi_h \, dx + \sum_{e \in \mathcal{E}} \int_{e} [A\nabla u_h] \phi_h \, ds = \int_{\Omega} f \phi_h \, dx, \quad \forall \phi_h \in V_h.
\]

(3.7)

where \( D^2_h \) is a piecewise defined Hessian. We also note that the stabilization term is different to the above case.

For sufficiently smooth \( u \), they obtain the following a-priori error bound.

\[
\|u - uh\|_{W^{2,p}(\Omega)} \leq h^{k-1}\|u\|_{W^{k+1,p}(\Omega)},
\]

i.e. in the case \( p = 2 \),

\[
\|u - uh\|_2 \leq h^{k-1}\|u\|_{k+1}.
\]

(3.8)

This result is roughly equivalent to the estimate for 3.2.2\(^3\), and furthermore they verify numerically that \( |u - uh|_1 = \mathcal{O}(h^k) \) and \( \|D^2_h (u - uh)\|_{L^2(\Omega)} = \mathcal{O}(h^{k-1}) \).

**Example 3.2.5 (Mu).** Recently in Mu and Ye [2017], they introduced a method that takes a different approach.

\[
\int_{\Omega} (A : D^2_h uh)(A : D^2_h \phi_h) \, dx + s(u_h, \phi_h) = \int_{\Omega} f A : D^2_h \phi_h \, dx, \quad \forall \phi_h \in V_h.
\]

(3.9)

\(^3\)Since moving from the \( H^1 \) norm to the \( H^2 \) norm is roughly the same as adding \( h^{-1} \) to the estimate.
where \( s(\cdot, \cdot) \) is a stabilization term defined by

\[
s(u_h, \varphi_h) = \int_{E} h^{-3}[u_h][\varphi_h]_0 \, ds + \int_{E_0} h^{-1}[\nabla u_h]_0 \cdot [\nabla \varphi_h]_0 \, ds.
\]

This term has been added to enforce smoothness and continuity across elements.

We note that the principal difference to the above methods is the change from \( \varphi_h \) to \( A: D^2 \varphi_h \), which gives the problem symmetry, but results in a higher order approximation. This leads to the benefit of higher regularity, however the fourth order nature of the problem could lead to higher computational costs.

In terms of convergence results, an equivalent approximation to (3.8) is proved, and this is verified for the \( k = 2 \) case in the \( L^2, H^1 \) and \( H^2 \) norms.

Now for the derivation within this paper, we consider a method that is similar to example 3.2.5, but takes a more general form. In section 3.3 we will fully introduce this idea and explain the differences to the above methods.

### 3.3 Minimization Method

In the following we look at a derivation for a new method which comes from minimizing the problem. We will first give an outline of the continuous formulation of the problem, before moving to the discrete case which will be the basis for our numerical method. Specifically the saddle point formulation in section 3.3.1 is used in the computations.

First of all, in order to set up the problem in general terms, let \( V \) be a Hilbert space with inner product \( a(\cdot, \cdot) : V \times V \to \mathbb{R} \) (i.e. a symmetric, positive definite, bilinear form). We wish to consider two different cases for this method. In the first case we will choose \( V = L^2(\Omega) \) and \( a(v, w) = \int_{\Omega} vw \, dx \) and in the second we will choose \( V = H^1(\Omega) \) and \( a(v, w) = \int_{\Omega} \nabla v \cdot \nabla w \, dx + \beta h^{-1} \int_{\partial\Omega} vw \, ds \), where \( \beta > 0 \).

Now the Riesz representation theorem says that for every \( w \in V^* \), there exists a unique \( u \in V \) such that,

\[
a(u, v) = \langle w, v \rangle, \quad \forall v \in V.
\]
where $\langle \cdot , \cdot \rangle : V^* \times V \to \mathbb{R}$ is the dual pairing. From this we can define an invertible projection operator $N : V^* \to V$ which has the property

$$a(Nw, v) = \langle w, v \rangle, \quad \forall v \in V,$$  \hspace{1cm} (3.10)

We can additionally define a norm using $N$ and $a$ as

$$\|w\|^2_N := a(Nw, Nw), \quad \forall w \in V^*.$$

We note that this norm is defined in the dual of $V$, however in the case of $V = L^2$, they are equivalent.

Using the above definitions, we can reformulate (3.1) as a minimization problem as follows.

**Definition 3.3.1** (Continuous Minimization Formulation). Let $u \in H^2(T) \cap H_0^1(\Omega) \subset V$ such that for $J : H^2(T) \cap H_0^1(\Omega) \to \mathbb{R}^+$

$$J(u) := \frac{1}{2}\|A : D_h^2 u + f\|^2_N \to \min,$$  \hspace{1cm} (3.12)

Then we call (3.12) the continuous minimization formulation of (3.1).

Here $D_h^2 u \in L^2(\Omega)$ is a piecewise approximation to the Hessian, i.e. $D_h^2 v|_K = D^2 v|_K$ for all $K \in T$, which we will use for the remainder of this method.

**Remark.** We note that it is necessary to have $u \in H^2(T) \cap H_0^1(\Omega)$ for the continuous case, however for the discrete case we will instead be working with $u \in V_h$.

Now by formulating the Euler-Lagrange equation of the above, by taking the functional derivative, we get a variational version of (3.12).

**Definition 3.3.2** (Continuous Euler-Lagrange Formulation). Let $u \in W$ such that

$$a(N(A : D_h^2 u), N(A : D_h^2 \varphi)) = l(\varphi), \quad \forall \varphi \in W.$$  \hspace{1cm} (3.13)
where the right-hand side is defined by

\[ l(\varphi) = -a(Nf, N(A : D_0^2\varphi)). \]

Then (3.13) is the **continuous Euler-Lagrange formulation** of (3.12).

We note that this formulation is written in a general way so that it can take different forms depending on the choices of \( a(\cdot, \cdot) \) (and by extension \( V \)). These different cases will be considered later on.

Now let us consider a discrete version of the problem. Note that the discrete version keeps virtually the same structure, with the only changes being the use of \( V_h, N_h, \) and \( u_h \), which are discrete versions of the above.

In particular \( N_h \) is a standard Galerkin approximation to \( N \), and takes a similar form to (3.10).

**Definition 3.3.3** (\( N_h \) projection). Recall \( V_h = C^0(\Omega) \cap P^k(T) \). We define \( N_h : V^* \to V_h \) by

\[ a(N_h v_h, \varphi_h) = (v_h, \varphi_h)_{L^2}, \quad \forall \varphi_h \in V_h. \] (3.14)

First, we note that we make a standard assumption related to the difference between our non-discrete and discrete projection operators \( N \) and \( N_h \).

**Assumption 3.3.1.** For \( N v \in H^k \), \( k \in \mathbb{N} \) and \( 0 \leq m \leq k \)

\[ \| (N - N_h)v \|_m \leq Ch^{k-m}\| Nv \|_k. \] (3.15)

We can also derive the following bound on the discrete projector by the non-discrete version.

**Lemma 3.3.1** (\( N_h \) bound). Let \( \|v\|_a := a(v, v) \) for \( v \in V \). Then

\[ \| N_h v \|_a \leq C \| N v \|_a, \quad \forall v \in V^*. \] (3.16)
Proof.
\[ \|N_h v\|^2_a = a(N_h v, N_h v), \]
\[ = a(N v, N_h v), \quad \text{(due to Galerkin orthogonality of } N_h) \]
\[ \leq C\|N v\|_a \|N_h v\|_a, \quad \text{(by boundedness of } a(\cdot, \cdot)) \]

Thus we divide through by \( \|N_h v\|_a \) to get \( \|N_h v\|_a \leq C\|N v\|_a \).

Having defined these discretized concepts, let us now consider the minimization formulation which we will use as the basis for our finite element method.

**Definition 3.3.4 (Euler-Lagrange Formulation).** Let \( v_h \in V_h \) such that
\[ a(N_h(A : D^2_h u_h), N_h(A : D^2_h \varphi_h)) + s(u_h, \varphi_h) = l_h(\varphi_h), \quad \forall \varphi_h \in V_h, \quad (3.17) \]

where we have added a stabilization term \( s(\cdot, \cdot) \), defined by
\[ s(v, w) := \int E \beta_1 h^9 \| \nabla v \| \cdot \| \nabla w \| ds + \beta_2 h^9 \int \Omega A : D^2_h v A : D^2_h w dx + \beta_3 h^r \int_{\partial \Omega} vw ds, \]

where \( \beta_1, \beta_2, \beta_3 > 0 \) are parameters, \( h \) is the grid size, \( p, q, r \in \mathbb{Z} \) and \( l_h(\cdot) \) is
\[ l_h(v) := -a(N_h f, N_h(A : D^2_h v)) - \beta_2 h^9 \int \Omega f A : D^2_h v dx. \]

Then (3.17) is the discrete **Euler-Lagrange formulation** of (3.1).

For convenience we denote the left hand side of (3.17) by
\[ b_h(v, w) := a(N_h(A : D^2_h v), N_h(A : D^2_h w)) + s(v, w). \quad (3.18) \]

**Remark.** The values of \( p, q, r \) will vary depending on the choice of \( a(\cdot, \cdot) \), as this will change the \( h \)-scaling of the first term in \( b_h \), thus we leave them unspecified.

It is quite easy to show Galerkin orthogonality for this problem, which shows that it is consistent with the original one.

**Lemma 3.3.2 (Galerkin Orthogonality).** Let \( u \in W = H^2(\Omega) \cap H^1_0(\Omega) \) be the
solution to (3.1), and \( b_h(\cdot, \cdot) \) be defined as in (3.18). Then we have

\[
b_h(u, v) = l_h(v), \quad \forall v \in V_h.
\] (3.19)

Consequently,

\[
b_h(u - u_h, v) = 0, \quad \forall v \in V_h.
\]

**Proof.**

\[
b_h(u, v) - l_h(v),
\]

\[
= a(N_h(A : D^2_hu_h), N_h(A : D^2_hv)) + \int_{\partial \Omega} uv \, ds + a(N_h f, N_h A : D^2_hv) + \beta_2 h^q (A : D^2_h u_h,A : D^2_h v)_L^2
\]

\[
+ \beta_3 h^r \int_{\partial \Omega} u v \, ds + a(N_h f, N_h A : D^2_h f) + \beta_2 h^q (A : D^2_h f,A : D^2_h v)_L^2
\]

\[
= (A : D^2_h u + f, N_h(A : D^2_h v))_L^2 + \beta_2 h^q (A : D^2_h u + f, A : D^2_h v)_L^2
\]

\[
+ \beta_3 h^r \int_{\partial \Omega} u v \, ds,
\]

(using (3.14) and \( \|\nabla u\| = 0 \)),

\[
= 0,
\]

where we have used the fact \(-A : D^2_h u = f\) in \(\Omega\) and \(u = 0\) on \(\partial \Omega\).

We can also prove the existence of a unique solution fairly easily due to the choice of stabilization term.

**Lemma 3.3.3 (Existence and Uniqueness).** Assume that the original equation (3.1) has a unique solution \(u_h \in W\). Then for \(\beta_1, \beta_2, \beta_3 > 0\), the approximation (3.17) has a unique solution \(u_h \in V_h\).

**Proof.** This follows by taking \(f = 0\) and showing the solution \(u_h\) must be zero.

Setting \(f = 0\) in (3.17) and taking \(\varphi_h = u_h\) gives

\[
0 = a(N_h(A : D^2_hu_h),N_h(A : D^2_hu_h)) + s(u_h,u_h),
\]

\[
= \|N_h(A : D^2_hu_h)\|^2_a + \beta_1 h^p \|\nabla u_h\|^2_\mathcal{E} + \beta_2 h^q \|A : D^2_h u_h\|^2_0 + \beta_3 h^r \|u_h\|_{\partial \Omega}^2.
\]

where \(\| \cdot \|_\mathcal{E}\) denotes the \(L^2\) norm over the skeleton of the grid and \(\| \cdot \|_{\partial \Omega}\) denotes the \(L^2\) norm over the boundary.

88
Since every term in this expression is $\geq 0$, they must each be zero. Thus we have $\|\nabla u_h\| = 0$ on $\mathcal{E}$, which means $u_h$ is smooth across elements, so $u_h \in C^1(\Omega)$. We already had $u_h \in C^0(\Omega) \cap P^k(\mathcal{T})$, so $u_h \in C^1(\Omega) \cap P^k(\mathcal{T})$, which by extension means $u_h \in H^2(\Omega)$. Additionally since $u_h|_{\partial\Omega} = 0$, we have that $u_h \in W$. Next since $A : D^2_h u_h = 0$ (and $u_h|_{\partial\Omega} = 0$), $u_h$ is a piecewise solution to (3.1) when $f = 0$. Since at the start we assumed the solution to (3.1) was unique, and $u_h \in W$, this means $u_h = 0$. □

Let us now return to the two specific examples of the method, which come from selecting appropriate choices for $V$ and $a(\cdot, \cdot)$.

**Example 3.3.1 (L^2 Minimization).** Let $V = L^2(\Omega)$ and $a(v, w) = \int_{\Omega} vw \, dx$ (thus $\| \cdot \|_{L^2}$ is simply the $L^2$ norm). In this case $\mathcal{N}_h$ defined in (3.14) is the discrete $L^2$ projection $P_h : L^2(\Omega) \to V_h$ defined by

$$\int_{\Omega} P_h v \varphi_h \, dx = \int_{\Omega} w \varphi_h \, dx, \quad \forall \varphi_h \in V_h. \quad (3.20)$$

So the discrete problem becomes

$$\int_{\Omega} P_h(A : D^2_h u_h) P_h(A : D^2_h \varphi_h) \, dx + s(u_h, \varphi_h) = l_h(\varphi_h), \quad \forall \varphi \in V_h, \quad (3.21)$$

where we choose $p = -1, q = 0, r = -3$ in $s(u_h, \varphi_h)$.

**Remark 1.** The formulation in (3.21) is similar to the one presented in Mu and Ye [2017], with the discrete projection $P_h$ and a different penalty term being the main differences.

**Remark 2.** The choice of $p, q$ and $r$ comes from examining how each term scales in magnitude with regards to the grid size $h$. For the main component,

$$\int_{\Omega} P_h(A : D^2_h u_h) P_h(A : D^2_h \varphi_h) \, dx$$

we can state that the $D^2_h u_h$ and $D^2_h \varphi_h$ each provide approximately $O(h^{-2})$ scaling, whilst the integral $dx$ itself provides $O(h^2)$ scaling, giving $O(h^{-2} h^{-2} h^2) = O(h^{-2})$.
Applying similar logic to the $s(u_h, \varphi_h)$ terms, we find that

$$
\beta_1 h^p \int_\mathcal{E} \left\| \nabla v \right\|_0 \cdot \left\| \nabla w \right\|_0 \, ds \approx \mathcal{O}(h^p h^{-2} h) = \mathcal{O}(h^{p-1})
$$

$$
\beta_2 h^q \int_\Omega A : D_h^2 v, A : D_h^2 w \, dx \approx \mathcal{O}(h^q h^{-4} h^2) = \mathcal{O}(h^{q-2})
$$

$$
\beta_3 h^r \int_{\partial\Omega} v w \, ds \approx \mathcal{O}(h^r h^0 h) = \mathcal{O}(h^{r+1})
$$

In order to match the original $\mathcal{O}(h^{-2})$, we thus choose $p = -1, q = 0, r = -3$.

**Example 3.3.2 ($H^{-1}$ Minimization).** Let $V = H^1(\Omega)$ and $a(v, w) = \int_\Omega \nabla v \cdot \nabla w \, dx + \beta_3 h^{-1} \int_{\partial\Omega} v w \, ds$ (in which case $\|\cdot\|_a^2 = |\cdot|_1^2 + h^{-1} \|\cdot\|_{\partial\Omega}^2$ where $|\cdot|_1$ is the $H^1$ seminorm).

In this case, $N_h$ is the (discrete) Ritz projection $N_h : H^{-1}(\Omega) \rightarrow V_h$ defined by

$$
\int_{\Omega} \nabla N_h w \nabla \varphi_h \, dx + \frac{\beta_3}{h} \int_{\partial\Omega} N_h w \varphi_h \, ds = \int_{\Omega} w \varphi_h \, dx, \quad \forall \varphi_h \in V_h. \quad (3.22)
$$

This gives us the following approximation.

$$
\int_{\Omega} \nabla N_h (A : D_h^2 u_h) \cdot \nabla N_h (A : D_h^2 \varphi_h) \, dx + \frac{\beta_3}{h} \int_{\partial\Omega} N_h (A : D_h^2 u_h) N_h (A : D_h^2 \varphi_h) \, ds + s(u_h, \varphi_h) = l_h(\varphi_h), \quad \forall \varphi_h \in V_h, \quad (3.23)
$$

where we choose $p = 1, q = 2, r = -1$ in $s(u_h, \varphi_h)$ using the same logic as in example 3.3.1, remark 2.

**Remark.** In this case the original problem is equivalent to the minimization of $A : D_h^2 u_h - f$ in the $H^{-1}$ norm.

**Remark 2.** This form for $a(\cdot, \cdot)$ comes from Nitsche’s method (see Freund and Stenberg [1995]) for solving Poisson’s equation. The second term is added to weakly impose boundary conditions so that $a(\cdot, \cdot)$ is invertible (the alternative would be working in $V = H^1_0$ and not using the term). We note that there exist additional terms $\int_{\partial\Omega} \nabla v \cdot m w \, ds - \int_{\partial\Omega} \nabla w \cdot m v \, ds$ in the full method that can be added for consistency, however as they do not appear to affect our numerical results we do not use them to simplify the presentation.
3.3.1 Saddle Point Formulation

One potential weakness to the forms (3.21) and (3.23) is that they can be tricky to implement numerically. For one, \( P_h \) and \( N_h \) require extra effort to implement. Another problem is the \( D_h^2 \varphi \) term present in these equations, since in later sections (specifically we refer to example 3.5.1) we plan to replace \( D_h^2 \) with \( H[\cdot] \), and \( H[\varphi] \) lacks an implementation in DUNE-FEMPY.

Thus we look here to rewrite the approximation into the form of a saddle point problem. This is the form we will use later in computational results, and we can also show directly the appeal of example 3.3.2 in terms of the complexity of the problem. To obtain this form, we introduce an additional variable.

\[
\sigma = -N_h(A : D_h^2 u_h + f).
\]

(3.24)

Taking \( a(\cdot, \varphi_h) \) of both sides of (3.24), and substituting \( \sigma \) into (3.17), we get two equations,

\[
a(\sigma, \varphi_h) = -a(N_h(A : D_h^2 u_h + f), \varphi_h),
\]

\[
a(\sigma, N_h(A : D_h^2 \varphi_h)) - s(u_h, \varphi_h) = -\beta_2 h^q(f, A : D_h^2 \varphi_h).
\]

Recalling property (3.14), i.e. \( a(N_h w, v) = (w, v)_{L^2} \), and rearranging the first equation, this can be rewritten as

\[
a(\sigma, \varphi_h) + (A : D_h^2 u_h, \varphi_h) = -(f, \varphi_h),
\]

\[
(\sigma, A : D_h^2 \varphi_h) - s(u_h, \varphi_h) = -\beta_2 h^q(f, A : D_h^2 \varphi_h).
\]

Letting \( b(v, w) := (A : D_h^2 v, w) \), we can think of the above in terms of bilinear forms instead.

\[
a(\sigma, \varphi_h) + b(u_h, \varphi_h) = -(f, \varphi_h),
\]

\[
b(\varphi_h, \sigma) - s(u_h, \varphi_h) = -\beta_2 h^q b(\varphi_h, f).
\]
We can then rewrite this system in matrix-vector form as

\[
\begin{pmatrix}
M & B \\
B^T & -S
\end{pmatrix}
\begin{pmatrix}
\sigma \\
u
\end{pmatrix}
=egin{pmatrix}
f \\
g
\end{pmatrix}
\]  
(3.25)

where here we can think of \(B\) as representing \(b(\cdot, \varphi_h)\), \(S\) as representing \(s(\cdot, \varphi_h)\) and \(M\) as \(a(\cdot, \varphi_h)\). The Schur complement of this is then the following.

\[
(B^T M^{-1} B + S) u = -B^T M^{-1} f - g. 
\]  
(3.26)

Thus we can see the differences between the two examples in terms of the order of the complexity of the problem. In the case \(a(\cdot, \cdot)\) is the \(L^2\) inner product, \(M\) is approximately the identity, i.e. of order 0. On the other hand if \(a(v, w) = \int_\Omega \nabla v \cdot \nabla w \, dx + \beta_3 h^{-1} \int_{\partial \Omega} vw \, ds\), then \(M\) becomes the discrete Laplace operator, so \(M^{-1}\) is of order -2.

From this we can see that the primary advantage of the \(H^{-1}\) approach in comparison to the \(L^2\) version is that the order of the method is \(O(h^2 h^{-2} h^2) = O(h^2)\), in comparison to \(O(h^4)\).

### 3.4 Error Analysis

#### 3.4.1 Error Bound for \(H^{-1}\) Method

As stated previously, analysis of a method similar to example 3.3.1 has already been carried out in Mu and Ye [2017]. Thus for the rest of this chapter, we will focus on the discrete problem for the \(V = H^1\) case. Let us state this formulation again.

**Example 3.4.1 (\(H^{-1}\) Minimization).** Since we are just dealing with a single example now, let us take \(p = 1, q = 2, r = -1\) in \(s(u, v)\). This results in the discrete
problem of finding $u_h \in V_h$ such that

$$
\int_{\Omega} \nabla N_h(A : D_h^2 u_h) \cdot \nabla N_h(A : D_h^2 \varphi_h) \, dx \\
+ \frac{\beta_3}{h} \int_{\partial \Omega} N_h(A : D_h^2 u_h) N_h(A : D_h^2 \varphi_h) \, ds + s(u_h, \varphi_h) = l_h(\varphi_h), \quad \forall \varphi_h \in V_h,
$$

where

$$
s(v, w) = \int_{\mathcal{E}} \beta_1 h \| \nabla v \| \cdot \| \nabla w \| \, ds + \beta_2 h^2 \int_{\Omega} A : D_h^2 v A : D_h^2 w \, dx + \frac{\beta_3}{h} \int_{\partial \Omega} vw \, ds,
$$

and

$$
l_h(v) = -a(N_h f, N_h(A : D_h^2 v)) - \beta_2 h^2 \int_{\Omega} f A : D_h^2 v \, dx.
$$

We also note that in this example $N$ acts as the inverse Laplace operator, and we make the assumption that it has elliptic regularity, i.e. for any $v \in V_h$, we have

$$
|Nv|_{H^2} \leq \|v\|_{L^2},
$$

where $|\cdot|_{H^2}$ denotes the $H^2$ seminorm. Now for the following error analysis it will be convenient to introduce a bilinear form using $N$ instead of $N_h$ as follows.

$$
b(v, w) = a(N(A : D_h^2 v), N(A : D_h^2 w)) + s(v, w).
$$

Note that this is equivalent to the LHS of the Euler-Lagrange formulation (3.17), save for the $N$ replacing $N_h$. For the choice of $a(\cdot, \cdot)$ given in the $H^1$ case, this becomes

$$
b(v, w) = \int_{\Omega} \nabla N(A : D_h^2 v) \cdot \nabla N(A : D_h^2 w) \, dx + \frac{\beta_3}{h} \int_{\partial \Omega} N(A : D_h^2 v) N(A : D_h^2 w) \, ds \\
+ \int_{\mathcal{E}} \beta_1 h \| \nabla v \| \cdot \| \nabla w \| \, ds + \beta_2 h^2 \int_{\Omega} A : D_h^2 v A : D_h^2 w \, dx + \frac{\beta_3}{h} \int_{\partial \Omega} vw \, ds.
$$

From this we define the norm $\|v\|_b^2 = b(v, v)$, $\forall v \in V_h$. Explicitly,

$$
\|v\|_b^2 = \|A : D_h^2 v\|_{\mathcal{N}}^2 + \beta_1 h \|\nabla v\|_E^2 + \beta_2 h^2 \|A : D_h^2 v\|_{0,0}^2 + \beta_3 h^{-1} \|v\|_{\partial \Omega}^2.
$$
Recall $\| \cdot \|_E$ denotes the $L^2$ norm over the skeleton of the grid and $\| \cdot \|_{\partial \Omega}$ denotes the $L^2$ norm over the boundary.

The primary reason for defining this norm is that we can use it to show convergence results by proving that the LHS of (3.17) is coercive and bounded with respect to it. Let us show this now.

**Lemma 3.4.1 (Coercivity of $b_h(v, w)$).** For $\beta_2$ large enough, we have that for all $v \in V_h$,

$$b_h(v, v) \geq C\|v\|_E^2.$$  

**Proof.** By definition from (3.18),

$$b_h(v, v) := a(N_h(A : D_h^2 v), N_h(A : D_h^2 v)) + s(v, v).$$

Let us consider these terms separately. First we start with the $a(\cdot, \cdot)$ component.

$$a(N_h(A : D_h^2 v), N_h(A : D_h^2 v)),$$

$$= a((N_h - N)(A : D_h^2 v), N_h(A : D_h^2 v))$$

$$+ a(N(A : D_h^2 v), N_h(A : D_h^2 v)),$$

$$= (N_h - N)(A : D_h^2 v), A : D_h^2 v)_{L^2} \quad \text{(using (3.14))}$$

$$+ a(N(A : D_h^2 v), N(A : D_h^2 v)), \quad \text{(using G.O. of $N_h$)}$$

$$\geq -\| (N_h - N)(A : D_h^2 v) \|_0 \| A : D_h^2 v \|_0 \quad \text{(using Cauchy-Schwarz)}$$

$$+ \| A : D_h^2 v \|_N^2,$$

$$\geq -Ch^2\| N(A : D_h^2 v) \|_2 \| A : D_h^2 v \|_0 \quad \text{(using assumption 3.3.1, i.e.)}$$

$$+ \| A : D_h^2 v \|_N^2, \quad \| (N_h - N)g \|_0 \leq Ch^2\| Ng \|_2.)$$

$$\geq -Ch^2\| A : D_h^2 v \|_N^2 + \| A : D_h^2 v \|_N^2. \quad \text{(using (3.27))}$$

We now add in the penalty term (which helps account for the negative term). We
get
\begin{align*}
b_h(v, v) \geq & \| A : D_h^2 v \|^2_N - C h^2 \| A : D_h^2 v \|_0^2 \\
& + \beta_1 h \| \nabla v \|_2^2 + \beta_2 h^2 \| A : D_h^2 v \|_0^2 + \beta_3 h^{-1} \| v \|_{B_0}^2 \\
= & \| A : D_h^2 v \|^2_N + (\beta_2 - C) h^2 \| A : D_h^2 v \|_0^2 + \beta_1 h \| \nabla v \|_2^2 + \beta_3 h^{-1} \| v \|_{B_0}^2.
\end{align*}

From here we can see that by taking \( \beta_2 > C \), the above expression can be bounded from below by (3.28) multiplied by a constant. \( \square \)

**Lemma 3.4.2** (Boundedness of \( b_h(v, w) \)). For all \( v, w \in V \),
\[ b_h(v, w) \leq C \| v \|_b \| w \|_b. \]

**Proof.**
\begin{align*}
b_h(v, w) \\
= & a(N_h(A : D_h^2 v), N_h(A : D_h^2 w)) + \int_\mathcal{E} \beta_1 h \| \nabla v \| \cdot \| \nabla w \| \ ds \\
& + \beta_2 h^2 \int_\Omega A : D_h^2 v A : D_h^2 w \ dx + \beta_3 h^{-1} \int_{\partial \Omega} v w \ ds, \\
\leq & C_1 \| N_h(A : D_h^2 v) \|_a \| N_h(A : D_h^2 w) \|_a + \beta_1 h \| \nabla v \|_\mathcal{E} \| \nabla w \|_\mathcal{E} \tag{*} \\
& + \beta_2 h^2 \| A : D_h^2 v \|_0 \| A : D_h^2 w \|_0 + \beta_3 h^{-1} \| v \|_{\partial \Omega} \| w \|_{\partial \Omega}, \\
\leq & C_2 \| N(A : D_h^2 v) \|_a \| N(A : D_h^2 w) \|_a + \beta_1 h \| \nabla v \|_\mathcal{E} \| \nabla w \|_\mathcal{E} \tag{**} \\
& + \beta_2 h^2 \| A : D_h^2 v \|_0 \| A : D_h^2 w \|_0 + \beta_3 h^{-1} \| v \|_{\partial \Omega} \| w \|_{\partial \Omega}, \\
\leq & C_2 \| v \|_b \| w \|_b + \| v \|_b \| v \|_b + \| v \|_b \| w \|_b + \| v \|_b \| w \|_b, \\
\leq & C_3 \| v \|_b \| w \|_b. \quad \square
\end{align*}

(In step (*) we have used the boundedness of \( a(\cdot, \cdot) \) and of the Cauchy-Schwarz inequality, and in step (***) we have used lemma 3.3.1.)

Now we will look to create a bound for the difference between the original solution \( u \) and discrete solution \( u_h \) for this method.

For the upcoming lemma we will make use of the standard result with regards
to the Lagrange interpolation operator $I_h$ that is similar to assumption 3.3.1.

**Lemma 3.4.3** (Lagrange Interpolation Estimate). Suppose $v \in H^r(\Omega)$ and let $I_h : H^r(\Omega) \to V_h$ where $V_h = C^0(\Omega) \cap P_k(\mathcal{T})$. Then for $0 \leq m \leq p$ and $p = \min\{k+1, r\}$,

$$\|v - I_h v\|_m \leq Ch^{p-m}\|v\|_p.$$  

**Proof.** See e.g. [Ciarlet and Raviart, 1972, Thm 5].

**Remark.** Naturally in the case where $v$ is arbitrarily smooth, this becomes

$$\|v - I_h v\|_m \leq Ch^{k+1-m}\|v\|_{k+1}.$$ 

Then we have the following bound on the difference between the solution and its interpolation in the $\| \cdot \|_b$ norm.

**Lemma 3.4.4** ($\| \cdot \|_b$ Interpolation Estimate). Let $I_h$ be the Lagrange interpolation operator defined in lemma 3.4.3. Then for $V_h = C^0(\Omega) \cap P_k(\mathcal{T})$, where $k \in \mathbb{N}$, $k \geq 2$ and $v \in H^{k+1}(\Omega)$ we have

$$\|v - I_h v\|_b \leq C h^{k-1}\|v\|_{k+1}. \quad (3.29)$$

**Proof.** Recall that the $\| \cdot \|_b$ is defined by

$$\|v\|_b^2 = \|A : D_h^2 v\|_{K}^2 + \beta_1 h \|\|\nabla v\|\|_2^2 + \beta_2 h^2\|A : D_h^2 v\|_0^2 + \beta_3 h^{-1} \|v\|_{\partial \Omega}^2. \quad (3.28)$$
For the first term, we have

\[
\|A : D_h^2 v\|_{N}^2 = a(\mathcal{N}(A : D_h^2 v), \mathcal{N}(A : D_h^2 v))
\]

\[
= (A : D_h^2 v, \mathcal{N}(A : D_h^2 v))_{L^2} \quad \text{(using (3.10))}
\]

\[
\leq \|A : D_h^2 v\|_0 \cdot \|\mathcal{N}(A : D_h^2 v)\|_0 \quad \text{(using Cauchy-Schwarz inequality)}
\]

\[
\leq C \|A : D_h^2 v\|_0 \cdot |\mathcal{N}(A : D_h^2 v)|_1 \quad \text{(using Poincaré inequality)}
\]

\[
\leq C \|A : D_h^2 v\|_0 \cdot \|\mathcal{N}(A : D_h^2 v)\|_a
\]

\[
= C \|A : D_h^2 v\|_0 \cdot \|A : D_h^2 v\|_{\mathcal{N}}
\]

Thus dividing through by \(\|A : D_h^2 v\|_{\mathcal{N}}\) we get

\[
\|A : D_h^2 v\|_{\mathcal{N}} \leq C \|A : D_h^2 v\|_0.
\]

We also note that for the last term, after substituting \(v - I_h v\), we can simply use the trace theorem, i.e.

\[
\frac{\beta_1}{h} \|v - I_h v\|_{\partial\Omega} \leq \frac{C}{h} \|v - I_h v\|_1
\]

\[
\leq C h^{k-1} \|v - I_h v\|_{k+1} \quad \text{(using (3.4.3))}
\]

Then the result follows by using [Mu and Ye, 2017, p308, Lemma 3], i.e.

\[
\|A : D_h^2 (v - I_h v)\|_0 + h^{-1} \|\nabla (v - I_h v)\|_{E_0} + h^{-1} \|\nabla (v - I_h v)\|_{E} \leq C h^{k-1} \|v\|_{k+1},
\]

where we recall \(E_0 = E \setminus \partial\Omega\). Putting \(v - I_h v\) into \(\|v\|_b\), and noting that \(h\) is small, and we can see that it is bounded by the above expression times a constant. \(\square\)

With these results, we can now prove the following error estimate.

**Theorem 3.4.1.** Let \(u\) be the solution to (3.1) and let \(u_h \in V_h\) be the solution to
the approximation (3.23). Then we have

\[ \| u - u_h \|_b \leq C h^{k-1} \| u \|_{k+1}. \]

**Proof.** Via the triangle inequality, we have

\[ \| u - u_h \|_b \leq \| u - I_h u \|_b + \| I_h u - u_h \|_b. \]

For the first term we can apply lemma 3.4.4, i.e.

\[ \| u - I_h u \|_b \leq h^{k-1} \| u \|_{k+1}. \]

For the second, we have

\[ \| I_h u - u_h \|_b^2 \leq C b_h(I_h u - u_h, I_h u - u_h) \]

\[ \leq C(b_h(u - u_h, I_h u - u_h) - b_h(u - I_h u, I_h u - u_h)) \quad \text{(coercivity of } b_h(\cdot, \cdot)) \]

\[ \leq C(b_h(u, I_h u - u_h) - I_h(u, u) - b_h(u - I_h u, I_h u - u_h)) \quad \text{(since } b_h(u_h, v) = l_h(v)) \]

\[ \leq C(b_h(u - I_h u, I_h u - u_h)) \quad \text{(lemma 3.3.2)} \]

\[ \leq C \| u - I_h u \|_b \| I_h u - u_h \|_b \quad \text{(boundedness of } b(\cdot, \cdot)) \]

\[ \leq C h^{k-1} \| u \|_{k+1} \| I_h u - u_h \|_b \quad \text{(lemma 3.4.4)} \]

Dividing through by \( \| I_h u - u_h \|_b \), we get

\[ \| I_h u - u_h \|_b \leq C h^{k-1} \| u \|_{k+1}. \]

Adding the two together gives us

\[ \| u - u_h \|_b \leq C h^{k-1} \| u \|_{k+1}. \]

This concludes the proof. \( \square \)
Remark. Although this result is consistent with other results in the literature (e.g. [Mu and Ye, 2017, Thm 1]), we remark that it in practice the convergence rate is greater than indicated, which we demonstrate in section 3.4.2.

3.4.2 Numerical Demonstration of $H^{-1}$ Interpolation Error

Recall the bound for the interpolation error given in lemma 3.4.4

$$\|v - I_h v\|_b \leq C h^{k-1} \|v\|_{k+1},$$

which ultimately leads to the error estimate in theorem 3.4.1. In particular for the case of $k = 1$, this leads to linear basis functions and no scaling with the grid size $h$. This is clearly accurate if one considers that for linear basis functions of the form $I_h v = ax, D^2(I_h v) = 0$, thus

$$\|v - I_h v\|_b := \|A : D_h^2(v - I_h v)\|_{N}^2 + \beta_1 h \|\nabla(v - I_h v)\|_{E}^2 + \beta_2 h^2 \|A : D_h^2(v - I_h v)\|_{0}^2 + \beta_3 h^{-1} \|v - I_h v\|_{\partial \Omega}$$

which does not converge as $h$ decreases.

On the other hand for $k = 2$ we have quadratic functions and a factor of $h$. Now whilst the estimate may be accurate for the linear case, numerical experiments we have done would indicate that this result might be suboptimal for the $H^{-1}$ minimization method, even if it is still accurate for the $L^2$ minimization method. We demonstrate the actual computational difference between the two methods here.

To start off with, we examine the simple case where $A = I$ in (3.1), which results in $-\Delta u = f$, giving us Poisson’s equation. For our experiments we suppose we have a smooth exact solution $u = \sin(2\pi x) \sin(2\pi y)$, and that we use a second order interpolation $I_h v$.

In that case the error for the $L^2$ method we want to examine is $\|\Delta(u - I_h v)\|_0$. For the $H^{-1}$ method, we want to compute $\|\nabla N\Delta(u - I_h v)\|_0$, which we can approximate via $\|\nabla N_h \Delta(u - I_h u)\|_0$. We can then calculate $N_h \Delta v$ numerically by
using (3.14) and substituting in $\Delta v$. That is to say we solve for $\xi$ (in a $P_3$ space),

$$(\nabla \xi, \nabla w) = (\Delta (u - I_h u), w).$$

Additionally we can in this simple case note that since $N_h$ approximates the inverse Laplace operator, $\|\nabla N_h \Delta (u - I_h u)\| \approx \|\nabla (u - I_h u)\|$, so we consider this too for comparison.

To compare the convergence of the error we will compute the estimated order of convergence (EOC) using the formula.

$$EOC = \frac{\log(\text{error}_{\text{new}}/\text{error}_{\text{old}})}{\log(h_{\text{new}}/h_{\text{old}})}.$$ 

Calculating the above in DUNE-FEMPY and varying the grid size, we get the results in table 3.1 (letting $e_h = u - I_h u$). From this we are able to show that the kind

<table>
<thead>
<tr>
<th>Elements</th>
<th>$|\Delta e_h|$</th>
<th>EOC</th>
<th>$|\nabla N_h(\Delta e_h)|$</th>
<th>EOC</th>
<th>$|\nabla e_h|$</th>
<th>EOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>20.88</td>
<td>-</td>
<td>1.483</td>
<td>-</td>
<td>0.984</td>
<td>-</td>
</tr>
<tr>
<td>128</td>
<td>10.82</td>
<td>0.942</td>
<td>0.435</td>
<td>1.768</td>
<td>0.264</td>
<td>1.899</td>
</tr>
<tr>
<td>512</td>
<td>5.462</td>
<td>0.985</td>
<td>0.113</td>
<td>1.940</td>
<td>0.0672</td>
<td>1.975</td>
</tr>
<tr>
<td>2048</td>
<td>2.738</td>
<td>0.997</td>
<td>0.0287</td>
<td>1.985</td>
<td>0.0169</td>
<td>1.994</td>
</tr>
</tbody>
</table>

of optimal interpolation estimate we would expect from the $H^{-1}$ case would have $O(h^2)$ for 2nd order interpolation, in comparison to the $O(h)$ for $L^2$. Furthermore, the similar convergence rate when compared to the $\|\nabla e_h\|$ case shows that it retains the same convergence as a typical Laplace scheme.

A comparison of the plots for the $L^2$ and $H^{-1}$ interpolation errors for 2048 elements is given in figures 3.1 and 3.2. To confirm that this still holds for non constant $A(x)$, we perform the same experiment for

$$A = \frac{16}{9} \begin{pmatrix} x^{2/3} & -x^{1/3}y^{1/3} \\ -x^{1/3}y^{1/3} & y^{2/3} \end{pmatrix}.$$ 

This gives us the following results in table 3.2. So once again we recover the same
convergence rate.

We note that these are the optimal results given a smooth enough solution. In particular the theory specifies that for $k + 1 = 3$, that $H^3$ functions or above are necessary for convergence.

Suppose we instead use the exact solution $u = x^{4/3}y^{4/3}$, (used in [Feng et al., 2015, Test 3]) for which $u \in W^{2,p}(\Omega) \cap W^{1,\infty}(\Omega)$, where $p < 3/2$. Then we get the results in table 3.3. We note that we do not obtain optimal convergence in such a case, which is consistent with the results from said paper (i.e. $|u - u_h|_1 = O(h^{5/6})$).

Table 3.3: Interpolation error for a non-smooth solution with different norms

<table>
<thead>
<tr>
<th>Elements</th>
<th>$|A : D^2 e_h|$</th>
<th>EOC</th>
<th>$|\nabla N_h A : D^2 e_h|$</th>
<th>EOC</th>
<th>$|\nabla e_h|_4$</th>
<th>EOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.290</td>
<td>-</td>
<td>0.0135</td>
<td>-</td>
<td>0.0249</td>
<td>-</td>
</tr>
<tr>
<td>128</td>
<td>0.207</td>
<td>0.483</td>
<td>6.509e-3</td>
<td>1.048</td>
<td>0.0140</td>
<td>0.833</td>
</tr>
<tr>
<td>512</td>
<td>0.147</td>
<td>0.492</td>
<td>2.820e-3</td>
<td>1.207</td>
<td>7.860e-3</td>
<td>0.833</td>
</tr>
<tr>
<td>2048</td>
<td>0.104</td>
<td>0.496</td>
<td>1.136e-3</td>
<td>1.312</td>
<td>4.411e-3</td>
<td>0.833</td>
</tr>
</tbody>
</table>
3.5 Finite Element Hessian

3.5.1 Derivation of FEH

In the previous section in (3.30) we demonstrated that our method may not show convergence for linear basis functions, since it uses a piecewise approximation to the Hessian. Additionally there remains the problem of there being no clear way to make use of the piecewise Hessian in the nonlinear case, since it lacks a nonlinear form.

One way to avoid these problems is to make use of the finite element Hessian (FEH), first derived in (3.2). For instance due to the fact that this formulation only uses first derivatives, it means linear basis functions can be used. Thus in this section we will properly introduce this concept, which we will later use in application to methods 3.3.1 and 3.3.2.

The FEH was first considered in Aguilera and Morin [2008], and was later used in a nonvariational context in Lakkis and Pryer [2010]. The principal idea is to rewrite the Hessian using a variational approach, allowing us to express it using lower derivatives. Additionally we follow the derivation used in Dedner and Pryer [2013] that considers it in a discontinuous Galerkin setting. This allows for a more efficient computation of the Hessian from a numerical point of view compared to a continuous formulation\(^5\).

Let \( v \in H^2(\Omega) \), and let \( n : \partial \Omega \rightarrow \mathbb{R}^d \) be the outward pointing normal of \( \Omega \). Then the Hessian \( D^2 v \), satisfies the following identity.

\[
\int_\Omega D^2 v \varphi \, dx = - \int_\Omega \nabla v \otimes \nabla \varphi \, dx + \int_{\partial \Omega} \nabla v \otimes n \varphi \, ds, \quad \forall \varphi \in H^1(\Omega). \tag{3.31}
\]

If \( v \in H^1(\Omega) \), the left hand side of (3.31) can still be expressed as a dual pair. In this case we have

\[
\langle D^2 v, \varphi \rangle = - \int_\Omega \nabla v \otimes \nabla \varphi \, dx + \int_{\partial \Omega} \nabla v \otimes n \varphi \, ds, \quad \forall \varphi \in H^1(\Omega).
\]

\(^5\)One reason for this is that the mass matrix can be locally constructed in a DG setting, meaning that it can be inverted much more easily.
We note that we will define $H[v]$ in such a way that it is defined elementwise. With that in mind we consider the following results for elementwise integration that can be obtained via the identities in definition 3.1.3.

**Proposition 3.5.1.** For a vector valued function $p \in H^1(\Omega)$ and scalar valued function $\varphi \in H^1(\Omega)$ we have

$$
\sum_{K \in T} \int_K \nabla \cdot p \varphi \, dx = \sum_{K \in T} \left( - \int_K p \cdot \nabla_h \varphi \, dx + \int_{\partial K} p \cdot n_K \varphi \, ds \right), \quad (3.32)
$$

where $\nabla_h = (D_h)^T$ is the elementwise gradient. Furthermore if $p \in L^2(\mathcal{E})^d$ and $\varphi \in L^2(\mathcal{E})$, the following identity holds

$$
\sum_{K \in T} \int_{\partial K} p \cdot n_K \varphi \, ds = \int_{\mathcal{E}_0} \llbracket p \rrbracket \llbracket \varphi \rrbracket \, ds + \int_{\mathcal{E}} \llbracket \varphi \rrbracket \cdot \{ p \} \, ds = \int_{\mathcal{E}} \llbracket p \varphi \rrbracket \, ds, \quad (3.33)
$$

An equivalent tensor formulation of (3.32)–(3.33) is

$$
\sum_{K \in T} \int_K D_h p \varphi \, dx = \sum_{K \in T} \left( - \int_K p \otimes \nabla_h \varphi \, dx + \int_{\partial K} p \otimes n_K \varphi \, ds \right), \quad (3.34)
$$

where the last term is given by

$$
\sum_{K \in T} \int_{\partial K} p \otimes n_K \varphi \, ds = \int_{\mathcal{E}_0} \llbracket p \rrbracket \otimes \{ \varphi \} \, ds + \int_{\mathcal{E}} \llbracket \varphi \rrbracket \otimes \{ p \} \, ds = \int_{\mathcal{E}} \llbracket p \varphi \rrbracket \otimes \, ds.
$$

By using equations (3.31) and (3.34), we formulate the following definition for the finite element Hessian in its most general form.

**Definition 3.5.1** (Generalized Finite Element Hessian). Let $u \in H^2(\mathcal{T})$, let $\hat{U} : H^1(\mathcal{T}) \to L^2(\mathcal{E})$ be a linear form and $\hat{p} : H^2(\mathcal{T}) \times H^1(\mathcal{T})^d \to L^2(\mathcal{E})^d$ a bilinear form representing approximations to $u$ and $\nabla u$ respectively over the skeleton of the triangulation. Then we define the **generalised finite element Hessian** $H[u]$ as
the solution of
\[
\int_K H[u] \varphi \, dx = - \int_K p \otimes \nabla_h \varphi \, dx + \int_{\partial K} \hat{p} \otimes n \varphi \, ds, \quad \forall \varphi \in H^1(T) \cap V_h, \quad (3.35)
\]
\[
\int_K \hat{p} \otimes \psi \, dx = - \int_K u D_h \psi \, dx + \int_{\partial K} \psi \otimes n \hat{U}_K \, ds, \quad \forall \psi \in (H^1(T))^d. \quad (3.36)
\]

**Theorem 3.5.1.** Let \( u \in H^2(T) \) and let \( \hat{U} \) and \( \hat{p} \) be defined as in Definition 3.5.1. Then the generalised finite element Hessian \( H[u] \) is given for each \( \varphi \in V_h \) as
\[
\int_\Omega H[u] \varphi \, dx = - \int_\Omega \nabla_h u \otimes \nabla_h \varphi \, dx + \int_E \{ \varphi \} \otimes \{ \hat{p} \} \otimes ds \quad (3.37)
\]

**Proof.** Note that in view of definition 3.1.3, for \( v \in L^2(E)^d \) and \( w \in L^2(E)^d \) we have the following identity
\[
\sum_{K \in T} \int_{\partial K} v \otimes n w \, ds = \int_E \{ w \} \otimes \{ v \} \, ds + \int_{E_0} \{ w \} \otimes \{ v \} \, ds. \quad (3.38)
\]
Then summing (3.35) over \( K \in T \) and making use of identity (3.38) we see
\[
\int_\Omega H[u] \varphi \, dx = - \int_\Omega \nabla_h u \otimes \nabla_h \varphi \, dx + \int_E \{ \varphi \} \otimes \{ \hat{p} \} \otimes ds \quad (3.39)
\]
Using the same argument for (3.36)
\[
\int_\Omega p \otimes \psi \, dx = - \int_\Omega u D_h \psi \, dx + \int_E \{ \psi \} \otimes \{ \hat{U} \} \otimes ds. \quad (3.40)
\]
Note that, again making use of (3.38) we have for each \( \psi \in H^1(T)^d \) and \( v \in H^1(T) \)
that
\[
\int_{\Omega} \psi \otimes \nabla_h v \, dx = - \int_{\Omega} D_h \psi v \, dx + \int_{\mathcal{E}} \{\psi\} \otimes \{v\} \, ds + \int_{\mathcal{E}_0} \{\psi\} \otimes \{v\} \, ds. \tag{3.41}
\]
Taking \( v = u \) in (3.41) and substituting into (3.40) we see
\[
\int_{\Omega} p \otimes \psi \, dx = \int_{\Omega} \psi \otimes \nabla_h u \, dx + \int_{\mathcal{E}} \hat{U} - u \otimes \{\psi\} \, ds + \int_{\mathcal{E}_0} \{\hat{U} - u\} \otimes \{\psi\} \, ds. \tag{3.42}
\]
Now choosing \( \psi = \nabla_h \varphi \) and substituting (3.42) into (3.39) we arrive at the finite element Hessian given by (3.37).

We can now present the definition previously stated in (3.2).

**Definition 3.5.2** (Finite Element Hessian). In the case where the fluxes in Definition 3.5.1 are chosen to be
\[
\hat{U} = \begin{cases} 
\{u\}, & \text{over } \mathcal{E}, \\
0, & \text{on } \partial \Omega
\end{cases},
\]
\[
\hat{p} = \{\nabla_h u\}, \text{ on } \mathcal{E} \cup \partial \Omega,
\]
then the **finite element Hessian** \( H[u] \) is a unique element of \( V_h^{d \times d} \) such that for all \( \varphi \in V_h \),
\[
\int_{\Omega} H[u] \varphi \, dx = - \int_{\Omega} \nabla_h u \otimes \nabla_h \varphi \, dx + \int_{\mathcal{E}} \{\nabla_h \varphi\} \otimes \{\nabla_h u\} \, ds \tag{3.2}
\]

**Remark.** Whilst this discrete form of the Hessian may be more complicated than its original form, we can show that when used to write the Laplace equation, it corresponds to a second order finite difference stencil.

To demonstrate this remark, let us consider an elementary problem in 1D, with linear basis functions for \( u \) and constant basis functions for \( H \). Since we are in 1D, let the domain be the interval \([0, 1]\), with the grid constructed via \( N \) evenly spaced vertices \( 0 \leq i \leq N \), with intervals of length \( h \). We will denote our current element by \( K_{i+\frac{1}{2}} \), which has vertices \( i \) and \( i + 1 \). This lets us rewrite the equation
elementwise as

$$\int_{K_{i+\frac{1}{2}}} H[u] \varphi \, dx = \int_{K_{i+\frac{1}{2}}} u' \varphi' \, dx + \int_{K_{i+\frac{1}{2}}} \{\varphi\}_{i+1} + \int_{K_{i+\frac{1}{2}}} \{\varphi'\}_i$$

$$+ \{\varphi\}_{i+1} + \{\varphi'\}_i.$$  

Since $\varphi$ is a basis function for $H$ and is constant, $\varphi' = 0$ and $\varphi(x) = \frac{1}{h}$. Thus $\int_{K_{i+\frac{1}{2}}} H[u] \varphi \, dx = H[u]_{i+\frac{1}{2}} (x h^{-1})^{i+1} = H[u]_{i+\frac{1}{2}}$ This lets us simplify the above to

$$H[u]_{i+\frac{1}{2}} = \frac{1}{h} \{u'\}_{i+1} - \frac{1}{h} \{u'\}_i$$

$$= \frac{1}{2h} \left( \frac{u_{i+2} - u_{i+1}}{h} + \frac{u_{i+1} - u_i}{h} \right) - \frac{1}{2h} \left( \frac{u_{i+1} - u_i}{h} + \frac{u_i - u_{i-1}}{h} \right)$$

$$= \frac{1}{2h^2} (u_{i+2} - u_{i+1} - u_i + u_{i-1})$$

Finally we consider this applied to the Laplace equation for $u$, by testing $H[u]$ with $v_i$ (a basis function for $u$). Since $v_i$ has support only on $K_{i-\frac{1}{2}}$ and $K_{i+\frac{1}{2}}$, this gives us the following.

$$\int_{\Omega} H[u] v_i \, dx = H[u]_{i-\frac{1}{2}} \int_{K_{i-\frac{1}{2}}} v_i \, dx + H[u]_{i+\frac{1}{2}} \int_{K_{i+\frac{1}{2}}} v_i \, dx$$

$$= \frac{1}{2} H[u]_{i-\frac{1}{2}} + \frac{1}{2} H[u]_{i+\frac{1}{2}}$$

$$= \frac{1}{4h^2} (u_{i+2} - 2u_i + u_{i-2})$$

As claimed, this indeed corresponds to a second order FD stencil. It should be noted however that due to the $\pm 2$ spaced grid points, such a stencil would rarely be used.

### 3.5.2 Numerical Implementation of FEH

Now that we have derived the finite element Hessian in 3.2, let us consider the implementation of this in a finite element scheme. In terms of incorporating it into our existing methods, for the $L^2$ and $H^1$ minimization methods, we simply replace the piecewise Hessian with the finite element Hessian and omit all penalty terms except the boundary term from $s(u, v)$. For instance for the $H^{-1}$ example (originally
formulated in example 3.3.2), we use the following method.

**Example 3.5.1** 
\( (H^{-1} \text{ minimization with FEH}) \). Find \( u_h \in V_h \) such that

\[
\int_{\Omega} \nabla N_h(u_h) \cdot \nabla N_h(A : H[u_h] \varphi_h) \, dx + \frac{\beta}{h} \int_{\partial \Omega} N_h(A : H[u_h]) N_h(A : H[\varphi_h]) \, ds + s(u_h, \varphi_h) = l_h(\varphi_h), \quad \forall \varphi_h \in V_h,
\]

where

\[
s(v, w) = \frac{\beta}{h} \int_{\partial \Omega} vw \, ds,
\]

and

\[
l_h(v) = -a(N_h f, N_h(A : H[v])).
\]

For the \( L^2 \) version we use exactly the same procedure, so we will skip it for brevity.

Now to present an idea of the numerical implementation that is used to compute the FEH, let us consider the formulation in terms of finite element basis functions. Let \( u \in V_h \) be a function on a fixed element \( K \in T \). It will be advantageous for us to assume that \( H[u] \in W_h \) which is potentially different to \( V_h \). Thus we express \( V_h \) and \( W_h \) on each element \( K \) in terms of basis functions as

\[
V_h = \text{span}\{\phi^K_{\mu} : \mu = 1, \ldots, R\}, \quad W_h = \text{span}\{\psi^K_{\nu} : \nu = 1, \ldots, S\}.
\]

We will also compute each entry of the Hessian separately as \( H_{ij} \), since this reflects what is done numerically. With that in mind, we can express (3.2) elementwise and entrywise as follows.

\[
\int_K H^K_{ij}[u] \Psi^K \, dx = - \int_K \partial_i u \partial_j \Psi^K \, dx + \frac{1}{2} \sum_{N \in N^K} \int_{e_N} \left( (\partial_i u|_K + \partial_i u|_N) n^K_i \partial_j \Psi^K + (u|_K - u|_N) n^K_i \partial_j \Psi^K \right) \, ds,
\]

where we denote the set of neighbours of \( K \) by \( N^K \), and \( e_N \) as the corresponding
edge. For convenience we denote the above expression by a functional \( l^K_{ij} \), i.e.

\[
l^K_{ij}(u, \Psi^K) := \int_K H^K_{ij}[u] \Psi^K \, dx. \tag{3.44}
\]

From this we can define the vector \( l^K_{ij}(u) \), which contains the degrees of freedom of \( l^K_{ij} \), by

\[
l^K_{ij}(u) = (l^K_{ij}(u, \Psi^K))_{\nu=1,...,S}.
\]

We use the natural notation that entry \( \nu \) of \( l^K_{ij}(u) \) is \( l^K_{ij\nu}(u) \). In addition we define the mass matrix in the usual way by \( M_K = (\int_K \Psi^K_{\nu} \Psi^K_\lambda)_{\nu\lambda=1,...,S} \). Thus we obtain the degrees of freedom of \( H^K_{ij}[u] \) (which we denote by \( H^K_{ij}[u] \)) by

\[
H^K_{ij}[u] = M_K^{-1} l^K_{ij}(u).
\]

Consequently we can recover the original \( H^K_{ij}[u] \) by

\[
H^K_{ij}[u] = M_K^{-1} l^K_{ij}(u) \cdot \Psi^K. \tag{3.45}
\]

where \( \Psi^K = (\Psi^K_\nu)_{\nu=1,...,S} \). This motivates the following algorithm.

Algorithm 3.5.1. To compute \( H^K_{ij}[u] \), we do the following.

### 3.6 Numerical Implementation in Dune-Fempy

Having considered various numerical methods for tackling nonvariational problems and looking at them analytically, let us now implement these methods and compare the results. In particular, we would like to compare the aforementioned approaches from the literature, i.e. examples 3.2.3, 3.2.4 and 3.2.5, and the new methods, examples 3.3.1 and 3.3.2. Additionally we will implement the two minimization methods with the finite element Hessian from section 3.5 in place of the piecewise Hessian, for further comparison.

The structure of this section is as follows. First we will describe the details of the implementation and the methods we are testing in section 3.6.1. Then in
for $\nu = 1$ to $S$ do

\[ l_{ij\nu}^K(u) = -\int_K \partial_i u|_K \partial_j \Psi^K_\nu \]

forall $N \in N^K$ do

\[ l_{ij\nu}^K(u) + = \frac{1}{2} \sum_{N \in N^K} \int_{\epsilon_N} \left( (\partial_i u|_K + \partial_i u|_N)n_j^K \Psi^K_\nu + (u|_K - u|_N)n_i^K \partial_j \Psi^K_\nu \right) \]

end

end

$H^K_{ij}[u] = M_K^{-1} l_{ij}^K(u)$

for $\nu = 1$ to $S$ do

\[ H^K_{ij}[u] + = H^K_{ij\nu}[u] \Psi^K_\nu \]

end

section 3.6.2 we will examine the effectiveness of each method in terms of errors and error convergence by applying them to three different problem cases. Lastly in section 3.6.3 we will look at efficiency in terms of iterations, condition numbers and time taken.

3.6.1 Numerical Setup

PDE and General Setup

We will conduct the tests using DUNE-FEMPY\(^6\), by writing the methods in variational form using Unified Form Language. As previously, we will look to solve equations of the form:

\[-A : D^2 u = f \quad \text{in} \quad \Omega,\]
\[u = 0 \quad \text{in} \quad \partial \Omega,\]

where the choice of $A$ is the main variable. For the prescribed exact solution, we will use the smooth function $u = \sin(2\pi x) \sin(2\pi y)$ unless otherwise stated, and we calculate the RHS by substituting $u$ into $-(A : D_h^2 v, w)$, as demonstrated in the

\(^6\)In terms of the specs, the desktop for the simulations has an Intel\textsuperscript{\textregistered} Xeon\textsuperscript{\textregistered} CPU E5-2650 with 10 cores and 198Gb of RAM.
following DUNE-FEMPY code.

Code Listing 3.1: The right-hand side calculation

```python
def rhs(A, exact):
    
b = -inner(A, grad(grad(exact[0])))*v[0]*dx
    return b
```

For all examples we will carry out all the computations on the square domain, \( \Omega = [0, 1]^2 \), i.e.

Code Listing 3.2: The initial grid for the test examples

```python
grid = create.grid('ALUSimplex', cartesianDomain([0, 0], [1, 1], [4, 4]))
```

The initial grid here is 4 squares in each direction, and the class 'ALUSimplex' specifies triangular elements. This choice of domain and exact solution means the Dirichlet boundary condition is consistently \( u = 0 \) on \( \partial \Omega \). Furthermore unless otherwise specified we will use Lagrange basis functions of order 2.

For the case of the minimization method, we use the matrix-vector form (3.25), i.e.

\[
\begin{pmatrix}
M & B \\
B^T & -S
\end{pmatrix}
\begin{pmatrix}
\sigma \\
u
\end{pmatrix}
= -
\begin{pmatrix}
f \\
g
\end{pmatrix}
\]

Recall \( B \) is the system matrix for \( b(v, w) = (A : D^2_h v, w) \), \( M \) is the system matrix for either \( a(v, w) = (v, w) \) in the \( V = L^2 \) case or \( a(v, w) = (\nabla v, \nabla w) + \beta_3 h^{-1}(v, w)_{\Omega} \) in the \( V = H^1 \) case, and \( S \) is the system matrix for the stabilization term \( s(v, w) = \beta_1 h^p \int_\Omega \| \nabla v \| \cdot \| w \| ds + \beta_2 h^q(A : D^2_h v, A : D^2_h w) + \beta_3 h^r \int_{\partial \Omega} v w ds \).

We note that for the numerical results we add into \( b(v, w) \) the extra term, \(- \int_\Omega A \nabla w \cdot \{ v \} ds \). This is a DG term used for consistency, which comes from applying the DG integration by parts formula to the variational formulation (3.4). Specifically we have the following formula from [Feng et al., 2015, 2.23].

\[
\int_\Omega \tau \cdot \nabla v dx = -\int_\Omega (\nabla \cdot \tau) v dx + \int_\Omega (\| \tau \| \{ v \} + \{ \tau \} : \{ v \}) ds + \int_{\partial \Omega} \tau \cdot n v ds,
\]
where \(v\) is any piecewise scalar function and \(\tau\) is a vector function. In the case where \(\tau = A \nabla u_h\) and \(v = \varphi_h \in V_h\), i.e. \(\|\varphi_h\| = 0\), this becomes

\[
\int_\Omega A \nabla u_h \cdot \nabla \varphi_h \, dx = - \int_\Omega A : D_h^2 u_h \varphi_h \, dx + \int_{\partial \Omega} \|A \nabla u_h\| \{\varphi_h\} \, ds, \tag{3.46}
\]

Now recall the variational form, that is

\[
\int_\Omega (A \nabla u_h \cdot \nabla \varphi_h + (\nabla \cdot A) \cdot \nabla u_h \varphi_h) \, dx \\
+ \int_{\partial \Omega} (\beta_h^{-1} u_h \varphi_h - A \nabla u_h \cdot n \varphi_h) \, ds = \int_\Omega f \varphi_h \, dx. \tag{3.4}
\]

We substitute (3.46) into the first term and remove the \(\nabla \cdot A\) term that is unsuited for nonvariational problems (and the boundary terms that are already accounted for in the minimization formulation’s penalty term) and we get

\[- \int_\Omega A : D_h^2 u_h \varphi_h \, dx + \sum_{e \in \mathcal{E}} \int_e \|A \nabla u_h\| \{\varphi_h\} \, ds = \int_\Omega f \varphi_h \, dx.\]

Thus we use the LHS of this equation for \(b(v, w)\).\(^7\)

Now let us consider the implementation of this in DUNE-FEMPY. We can treat the above as one system of equations and use a standard linear solver (e.g. the conjugate gradient method) to calculate the solution. We assemble each component as a different scheme, e.g. to assemble \(B\) for the minimization method we have

```
Code Listing 3.3: Assembling B

1 a = inner(A, grad(grad(u[0]))) * v[0] * dx \\
2 - jump(A * grad(u[0]), n) * avg(v[0]) * dS \\
3 b = -rhs(A, exact) \\
4 scheme = create.scheme("galerkin", a == b, space, solver=solver) \\
5 B = scheme.assemble(uh).as_numpy
```

We then assemble the remaining components in a similar way and construct the system matrix. As an example, for the \(L^2\) minimization method we do the following.

\(^7\)We note that the analysis from sections 3.3 and 3.4.1 do not account for this term, and we have added it for the numerical results simply because it appears to improve the convergence rate.
Code Listing 3.4: Assembling the system matrix for the $V = L^2$ case

```python
BT = B.transpose(copy=True)

# assembling M
mass_model = inner(u, v)*dx == 0
mainScheme = create.scheme("h1", mass_model, space)
M = mainScheme.assemble(uh).as_numpy.tocsc()

# assembling S
s = 1/he * inner(jump(grad(u[0])), jump(grad(v[0]))) * dS \
    + inner(A, grad(grad(u[0] - exact[0]))) * inner(A, \ 
    grad(grad(v[0]))) * dx + beta/he0**3 * inner(u, v) * ds
penalty = create.scheme("galerkin", s == 0, space, solver='cg')
S = -penalty.assemble(uh).as_numpy
from scipy.sparse import bmat
system = bmat([[M, B], [BT, S]])
```

The remaining methods are described below.

**Methods**

Here we will compare the proposed methods. First of all, we note that the weak Dirichlet boundary condition, $\beta_3 h^r \int_{\partial \Omega} vw \, ds$, has been used in every method for consistency. We choose the value of $r = -3$ for the $L^2$ minimization methods and the method from Mu and Ye [2017], and $r = -1$ for the remaining methods. This is so the term scales correctly with the grid size $h$ for each method.

For $\beta_3$, the value has been chosen based on the analysis from [Ainsworth and Rankin, 2008, Lemma 1]. Specifically we have the following simplified bound on the value of the parameter.

$$\beta_3 > k(k + 1)\rho(A)$$

where $k$ is the polynomial order, $\rho(A)$ is the largest eigenvalue of $A$ from the original problem 3.1. For instance the choice of $k = 2$, and the simplest problem (where $\rho(A) = 1$), leads to $\beta_3 = 6$.

$\beta_1$ and $\beta_2$ have been chosen more empirically, by trying different values and examining convergence rates. In the following code they are taken to be the same.
value, \( \sigma \), to avoid ambiguity with \( \beta_3 \) (\( \beta \)).

Let us now list the methods, with abbreviations (\( \text{var}, \text{l2D2}, \) etc.) for labelling purposes.

- \( \text{var} \) - Variational approach (3.4). This is implemented in the code below.

\[
\hat{\text{var}} - \text{Variational approach (3.4). This is implemented in the code below.}
\]

\[
\text{Code Listing 3.5: The scheme for the variational method}
\]
\begin{verbatim}
1 a = inner(A*grad(u[0]), grad(v[0]))*dx
2 if div(A) != ufl.as_vector([0, 0]):
3     print('non-constant A used')
4     a += inner(div(A), grad(u[0]))*v[0]*dx
5 a += beta/\he0*inner(u, v)*ds - dot(A*grad(u[0]), n)*v[0]*ds
6 b = rhs(A, exact)
7 scheme = create.scheme("galerkin", a==b, space, solver=solver)
\end{verbatim}

- \( \text{l2D2} \) - Minimization method from 3.3.1 (refer to 3.3 and 3.4 above for details).

- \( \text{h1D2} \) - Minimization method from 3.3.2. This uses the same approach as the \( \text{l2D2} \) case, except with a different \( M \) and \( S \) as follows.

\[
\hat{\text{l2D2}} - \text{Minimization method from 3.3.1 (refer to 3.3 and 3.4 above for details).}
\]

\[
\hat{\text{h1D2}} - \text{Minimization method from 3.3.2. This uses the same approach as the \text{l2D2} case, except with a different \( M \) and \( S \) as follows.}
\]

\[
\text{Code Listing 3.6: } M \text{ and } S \text{ for the } \text{h1D2} \text{ method}
\]
\begin{verbatim}
1 # assembling M
2 laplace = inner(grad(u), grad(v))*dx + beta/\he0*u[0]*v[0]*ds
3 laplace_model = create.model("integrands", grid, laplace == 0)
4 mainScheme = create.scheme("galerkin", laplace_model, space)
5 M = mainScheme.assemble(uh).as_numpy.tocsc()
6 # assembling S
7 sigma = 0.1
8 s = sigma*\he*inner(jump(grad(u[0])), jump(grad(v[0])))*dS
9     + sigma*\he^2*inner(A, grad(grad(u[0] - exact[0])))*dS
10     inner(A, grad(grad(v[0])))*dx + beta/\he0*inner(u, v)*ds
11 penalty = create.scheme("galerkin", s == 0, space, solver='cg')
12 S = -penalty.assemble(uh).as_numpy
\end{verbatim}

- \( \text{l2H} \) - Minimization method 3.3.1 with FE Hessian \( H[u] \). To implement \( H[u] \) we use an 'nv' scheme (a new implementation in the DUNE-FEMNV module

113
which was created for this thesis) which effectively maps $A : D^2 u$ to $A : H[u]$. Thus our $B$ term is almost the same, except without the added term. For $S$ we just use the weak Dirichlet condition. The same $M$ is used as in L2D2 (3.4).

Code Listing 3.7: $B$ and $S$ for the L2H method

```
# assembling $S$
1 a = inner(A, grad(grad(u[0])))\*v[0]\*dx
2 b = -rhs(A, exact)
3 scheme = create.scheme("nv", space, a==b, penalty=0,
4       solver=solver, polOrder=space.order)
5 B = scheme.assemble(uh).as_numpy

# assembling $S$
6 s = beta/he0**3 * inner(u, v) * ds
7 penalty = create.scheme(" galerkin", s == 0, space, solver='cg ')
8 S = -penalty.assemble(uh).as_numpy
```

- **h1H** - Minimization method 3.3.2 with FE Hessian $H[u]$. Here we use the $B$ from L2H (3.7) and the $M$ from H1D2 (3.6). The only change is the h-scaling on the penalty.

Code Listing 3.8: $S$ for the h1H method

```
s = beta/he0 * inner(u, v) * ds
2 penalty = create.scheme(" galerkin", s == 0, space, solver='cg ')
3 S = -penalty.assemble(uh).as_numpy
```

- **nvdg** - Example 3.2.3 from Dedner and Pryer [2013]. This is simply the $B$ from the L2H and h1H minimization methods, with the penalty terms added from within the C++ class via a penalty parameter.

- **feng** - Example 3.2.4 from Feng et al. [2015].

Code Listing 3.9: The feng method

```
a = -inner(A, grad(grad(u[0])))\*v[0]\*dx
2 b = rhs(A, exact)
3 s = jump(A*grad(u[0]), n)*avg(v[0])\*dS \```
• **mu** - Example 3.2.5 from Mu and Ye [2017]. We use $\beta = 1$ here as it reflects the choice given in the paper. We also implement the RHS by subtracting it directly from the bilinear form, since $a$ is slightly different here.

**Code Listing 3.10: The mu method**

```python
a = inner(A, grad(grad(u[0] - exact[0]))) * inner(A, 
    grad(grad(v[0]))) * dx

s = 1/\he * inner( jump(grad(u[0])), jump(grad(v[0]))) ) * dS \n    + 1/\he**3 * inner(u, v) * ds

scheme = create.scheme("galerkin", a + s == 0, space, 
    solver=solver)

B = scheme.assemble(uh).as_numpy
```

**Symmetry**

One important consideration between methods is whether they are symmetric or not. Symmetric methods allow for the use of a more efficient conjugate gradient (CG) solver over a non-symmetric solver (such as a bi-conjugate gradient (BiCGSTAB) or generalized minimal residual (GMRES) solver). In particular we note that the minimization methods, and the **mu** method have the quality of being symmetric.

**Remark.** In the case of problems with a constant $A$ such as the Laplace problem, **var** and **nvdg** can be made symmetric with the addition of the symmetrizing term $- \int_{\partial\Omega} A\nabla\varphi_h \cdot \mathbf{n} u_h \, ds$. However in general these methods will not be symmetric.

We note that in terms of linear solvers, a conjugate gradient solver is used if the system matrix is found to be symmetric (according to table 3.4), otherwise a
biconjugate gradient stabilized method is used. In both cases a tolerance of $10^{-9}$ is used and a max iteration cap of $10^6$.

**Preconditioners**

For the minimization methods in particular, preconditioners are very important for efficiency. We calculate the approximation to the inverse of the system using an incomplete LU decomposition (ILU) from the SciPy package.

We note that some methods appear to require different levels of drop tolerance in order to converge, which we detail in table 3.5. These are based on whether the methods converge for the most refined grid used. Additionally we use a fill ratio upper bound of 30. This is not optimal for all methods but does not appear to affect efficiency.

### Table 3.5: Levels of drop tolerance necessary for ILU

<table>
<thead>
<tr>
<th>Method</th>
<th>l2D2</th>
<th>l2H</th>
<th>h1D2</th>
<th>h1H</th>
<th>mu</th>
<th>feng</th>
<th>nvdg</th>
<th>var</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tolerance</td>
<td>$10^{-12}$</td>
<td>$10^{-7}$</td>
<td>$10^{-7}$</td>
<td>$10^{-7}$</td>
<td>$10^{-7}$</td>
<td>$10^{-8}$</td>
<td>$10^{-8}$</td>
<td>$10^{-5}$</td>
</tr>
</tbody>
</table>

**3.6.2 Effectiveness and Convergence Rates**

Let us now look at the results and discuss the effectiveness of the methods. For testing purposes we shall consider 3 examples, beginning with the following.

**Example 3.6.1** (Poisson’s Equation). First let us look at the simplest case, Poisson’s equation.

$$A = \begin{pmatrix} 1 & 0 \\
0 & 1 \end{pmatrix}, \quad \text{i.e. } -\Delta u = f.$$ 

This choice of $A$ inserted into (3.1) leads to Poisson’s equation, which can be easily compared to the variational version.

We choose this as our first example as a simple benchmark to make sure all methods including the variational approach are working. Additionally we want the nonvariational methods to behave as similarly as possible to the variational method.
in the case where both are applicable. The bilinear form we use for the variational method is as follows.

\[
\int_{\Omega} \nabla u \cdot \nabla v \, dx + \int_{\partial \Omega} \beta h^{-1} uv - \nabla u \cdot \mathbf{n} v \, ds = \int_{\Omega} f v \, dx
\]

We compute the solution for increasing levels of grid refinement, and consider the error \( e_h \) between the exact and computed solution in different norms. In particular we consider the norm

\[
\|e_h\|_{D^1} := \|A : D_h^2 e_h\|_{H^{-1}} \equiv \|\nabla N(A : D_h^2 e_h)\|
\]

where we approximate \( N \) by \( N_h \) in the same way as in section 3.4.2. We will also use the norm

\[
\|e_h\|_{D^2} := \|A : D_h^2 e_h\|_{L^2}
\]

For the variational case we display in table 3.6 the errors and estimated orders of convergence (EOCs). Let us first comment on the observed orders of convergence.

Table 3.6: Variational method applied to Poisson’s equation

<table>
<thead>
<tr>
<th>Grid</th>
<th>( |e_h| )</th>
<th>EOC</th>
<th>( |\nabla e_h| )</th>
<th>EOC</th>
<th>( |e_h|_{D^1} )</th>
<th>EOC</th>
<th>( |e_h|_{D^2} )</th>
<th>EOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.02858</td>
<td>-</td>
<td>0.8902</td>
<td>-</td>
<td>1.913</td>
<td>-</td>
<td>21.66</td>
<td>-</td>
</tr>
<tr>
<td>128</td>
<td>0.003732</td>
<td>2.94</td>
<td>0.252</td>
<td>1.82</td>
<td>0.6626</td>
<td>1.53</td>
<td>11.18</td>
<td>0.954</td>
</tr>
<tr>
<td>512</td>
<td>0.0004776</td>
<td>2.97</td>
<td>0.06585</td>
<td>1.94</td>
<td>0.1842</td>
<td>1.85</td>
<td>5.518</td>
<td>1.02</td>
</tr>
<tr>
<td>2048</td>
<td>6.024e-05</td>
<td>2.99</td>
<td>0.01672</td>
<td>1.98</td>
<td>0.04759</td>
<td>1.95</td>
<td>2.743</td>
<td>1.01</td>
</tr>
<tr>
<td>8192</td>
<td>7.552e-06</td>
<td>3.0</td>
<td>0.004204</td>
<td>1.99</td>
<td>0.01203</td>
<td>1.98</td>
<td>1.37</td>
<td>1.0</td>
</tr>
</tbody>
</table>

For the \( L^2 \) norm we have roughly third order convergence, and for the \( H^1 \) seminorm and \( \|A : D_h^2 e_h\|_{-1} \) (which should be approximately the same), we have second order. Lastly \( \|A : D_h^2 e_h\| \) gives us first order convergence, which is expected as it is roughly the \( H^2 \) norm. These results are as expected for polynomial degree of 2. Thus we consider them to be the target for other methods to attain.

Since the \texttt{nvdg} method is identical to the \texttt{var} method in this case, we obtain the same EOCs and errors as shown in table 3.7. Now let us look at the other methods. We shall first start with the minimization methods 3.3.1 and 3.3.2.
Table 3.7: Nonvariational (DG) method applied to Poisson’s equation

| Grid | $|e_h|$ | EOC | $|\nabla e_h|$ | EOC | $|e_h|_{D1}$ | EOC | $|e_h|_{D2}$ | EOC |
|------|--------|-----|-------------|-----|-------------|-----|-------------|-----|
| 32   | 0.02858| -   | 0.8902      | -   | 1.913       | -   | 21.66       | -   |
| 128  | 0.003732| 2.94| 0.252       | 1.82| 0.6626      | 1.53| 11.18       | 0.954|
| 512  | 0.0004776| 2.97| 0.06585    | 1.94| 0.1842      | 1.85| 5.518       | 1.02 |
| 2048 | 6.024e-05| 2.99| 0.01672    | 1.98| 0.04759     | 1.95| 2.743       | 1.01 |
| 8192 | 7.552e-06| 3.00| 0.004204   | 1.99| 0.01203     | 1.98| 1.37        | 1.0  |

For the $L^2$ version we show the results in table 3.8. The main difference we note to the variational version is that the $L^2$ error only attains second order instead of third. However the result is different for the $H^{-1}$ case (table 3.9). Here we recover the same $L^2$ EOC of 3 as we saw for the nvdg and var methods, and the other EOCs remain the same. As was demonstrated previously in section 3.4.2, this would indicate the $H^{-1}$ version of the method to be superior in terms of convergence.

Table 3.8: $L^2$ minimization method applied to Poisson’s equation

| Grid | $|e_h|$ | EOC | $|\nabla e_h|$ | EOC | $|e_h|_{D1}$ | EOC | $|e_h|_{D2}$ | EOC |
|------|--------|-----|-------------|-----|-------------|-----|-------------|-----|
| 32   | 0.1085 | -   | 1.178       | -   | 1.753       | -   | 21.03       | -   |
| 128  | 0.02736| 1.99| 0.3396      | 1.79| 0.5842      | 1.59| 10.89       | 0.95 |
| 512  | 0.005443| 2.33| 0.08213    | 2.05| 0.1552      | 1.91| 5.44        | 1.0  |
| 2048 | 0.001163| 2.23| 0.01986    | 2.05| 0.03951     | 1.97| 2.726       | 0.996|
| 8192 | 0.0002702| 2.11| 0.004862   | 2.03| 0.009947    | 1.99| 1.366       | 0.997|

Next we observe in tables 3.10 and 3.11, the results of the same method but using the finite element Hessian in place of the piecewise Hessian. Here we note in particular that in the $L^2$ case, the addition of the finite element Hessian improves the $L^2$ error’s EOC from 2 to 3. Otherwise we observe the same results as before.

Table 3.9: $H^{-1}$ minimization method applied to Poisson’s equation

| Grid | $|e_h|$ | EOC | $|\nabla e_h|$ | EOC | $|e_h|_{D1}$ | EOC | $|e_h|_{D2}$ | EOC |
|------|--------|-----|-------------|-----|-------------|-----|-------------|-----|
| 32   | 0.04075| -   | 0.8951      | -   | 1.49        | -   | 20.12       | -   |
| 128  | 0.00487| 3.06| 0.2558      | 1.81| 0.4208      | 1.82| 10.6        | 0.924|
| 512  | 0.0005604| 3.12| 0.06621    | 1.95| 0.1066      | 1.98| 5.408       | 0.971|
| 2048 | 6.687e-05| 3.07| 0.01674    | 1.98| 0.02676     | 1.99| 2.725       | 0.989|
| 8192 | 8.17e-06| 3.03| 0.004205   | 1.99| 0.006703    | 2.0 | 1.367       | 0.995|

We now collect the EOCs for all methods including the remaining ones into table 3.12, which averages the final 3 EOCs. We plot some of these results in figures 118.
### Table 3.10: $L^2$ minimization method with $H[u]$

<table>
<thead>
<tr>
<th>Grid</th>
<th>$|e_h|$</th>
<th>EOC</th>
<th>$|\nabla e_h|$</th>
<th>EOC</th>
<th>$|e_h|_{D1}$ EOC</th>
<th>$|e_h|_{D2}$ EOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.06574</td>
<td>-</td>
<td>0.9629</td>
<td>-</td>
<td>1.923</td>
<td>21.76</td>
</tr>
<tr>
<td>128</td>
<td>0.01092</td>
<td>2.59</td>
<td>0.2683</td>
<td>1.84</td>
<td>0.656</td>
<td>1.55</td>
</tr>
<tr>
<td>512</td>
<td>0.001549</td>
<td>2.82</td>
<td>0.06867</td>
<td>1.97</td>
<td>0.1825</td>
<td>1.85</td>
</tr>
<tr>
<td>2048</td>
<td>0.0002084</td>
<td>2.89</td>
<td>0.01715</td>
<td>2.0</td>
<td>0.04732</td>
<td>1.95</td>
</tr>
<tr>
<td>8192</td>
<td>2.711e-05</td>
<td>2.94</td>
<td>0.004265</td>
<td>2.01</td>
<td>0.01199</td>
<td>1.98</td>
</tr>
</tbody>
</table>

### Table 3.11: $H^{-1}$ minimization method with $H[u]$

<table>
<thead>
<tr>
<th>Grid</th>
<th>$|e_h|$</th>
<th>EOC</th>
<th>$|\nabla e_h|$</th>
<th>EOC</th>
<th>$|e_h|_{D1}$ EOC</th>
<th>$|e_h|_{D2}$ EOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.03113</td>
<td>-</td>
<td>0.8814</td>
<td>-</td>
<td>1.932</td>
<td>21.5</td>
</tr>
<tr>
<td>128</td>
<td>0.00414</td>
<td>2.91</td>
<td>0.2505</td>
<td>1.81</td>
<td>0.6731</td>
<td>1.52</td>
</tr>
<tr>
<td>512</td>
<td>0.0005269</td>
<td>2.97</td>
<td>0.06567</td>
<td>1.93</td>
<td>0.1859</td>
<td>1.86</td>
</tr>
<tr>
<td>2048</td>
<td>6.571e-05</td>
<td>3.0</td>
<td>0.0167</td>
<td>1.98</td>
<td>0.04782</td>
<td>1.96</td>
</tr>
<tr>
<td>8192</td>
<td>8.169e-06</td>
<td>3.01</td>
<td>0.004201</td>
<td>1.99</td>
<td>0.01206</td>
<td>1.99</td>
</tr>
</tbody>
</table>

### Table 3.12: Table of EOCs for the Laplace example

<table>
<thead>
<tr>
<th>Method</th>
<th>$|e_h|$ EOC</th>
<th>$|\nabla e_h|$ EOC</th>
<th>$|e_h|_{D1}$ EOC</th>
<th>$|e_h|_{D2}$ EOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>feng</td>
<td>2.963</td>
<td>1.963</td>
<td>1.91</td>
<td>1.01</td>
</tr>
<tr>
<td>h1D2</td>
<td>3.073</td>
<td>1.973</td>
<td>1.99</td>
<td>0.985</td>
</tr>
<tr>
<td>h1H</td>
<td>2.993</td>
<td>1.967</td>
<td>1.937</td>
<td>1.01</td>
</tr>
<tr>
<td>l2D2</td>
<td>2.223</td>
<td>2.043</td>
<td>1.957</td>
<td>0.9977</td>
</tr>
<tr>
<td>l2H</td>
<td>2.883</td>
<td>1.993</td>
<td>1.927</td>
<td>1.01</td>
</tr>
<tr>
<td>mu</td>
<td>1.83</td>
<td>1.88</td>
<td>1.88</td>
<td>0.937</td>
</tr>
<tr>
<td>nvgd</td>
<td>2.987</td>
<td>1.97</td>
<td>1.927</td>
<td>1.01</td>
</tr>
<tr>
<td>var</td>
<td>2.987</td>
<td>1.97</td>
<td>1.927</td>
<td>1.01</td>
</tr>
</tbody>
</table>

3.3 and 3.4, which show that for the $L^2$ error the methods are for the most part the same, with the mu and l2D2 examples performing slightly worse than the others. We also compare the errors for the $H^1$ norm in figures 3.5 and 3.6, which demonstrate that all methods obtain a second order convergence rate, with the mu method being a slight underperformer.

**Example 3.6.2** (Advection dominated (AD) problem). For our second example we introduce an “advection dominated” problem (adapted from [Lakkis and Pryer,
This equation has derivatives of the following form.

\[
\frac{\partial}{\partial x_i} (\tan^{-1}(5000(x^2 + y^2 - 1)) + 2) = \frac{10000x_i}{(5000(x^2 + y^2 - 1) + 2)^2 + 1}
\]

These derivatives become particularly large on the unit circle. Recall that the variational method (3.4) includes the derivatives of \( A \). Thus whilst this problem can still be written in such a form, and will converge for a high enough grid resolution,
the variational method is ill suited.

Let us first demonstrate this fact by presenting the results for the var method in table 3.13. As stated, the method does appear to converge on the final step.

Table 3.13: Variational method applied to the AD equation

| Grid | $|\varepsilon_h|$ | EOC | $||\nabla \varepsilon_h||$ | EOC | $||\varepsilon_h||_{D1}$ | EOC | $||\varepsilon_h||_{D2}$ | EOC |
|------|-----------------|-----|-------------------------|-----|----------------------|-----|----------------------|-----|
| 32   | 0.171           | -   | 1.452                   | -   | 1.969                | -   | 25.27               | -   |
| 128  | 0.1503          | 0.186 | 1.168                  | 0.313 | 1.226                | 0.684 | 27.0               | -0.0957 |
| 2048 | 1.302           | -0.14 | 15.2                   | -0.639 | 24.19               | -1.14 | 1300.0            | -1.55 |
| 8192 | 0.05838         | 4.48 | 4.022                   | 1.92  | 4.492               | 2.43  | 617.5              | 1.07  |

However the other steps seem to face significant problems, even diverging from the solution. For this reason we do not include it in the plots.

As before we can show all the remaining methods’ average EOCs in table 3.14. We will also plot the $L^2$ errors and EOCs for the remaining methods in figures 3.7 and 3.8.

Table 3.14: Table of EOCs for the AD problem

| Method | $|\varepsilon_h|$ EOC | $||\nabla \varepsilon_h||$ EOC | $||\varepsilon_h||_{D1}$ EOC | $||\varepsilon_h||_{D2}$ EOC |
|--------|-----------------|-------------------------|----------------------|----------------------|
| feng   | 2.443           | 1.983                   | 1.89                 | 0.977                |
| h1D2   | 2.523           | 2.02                    | 1.92                 | 0.9433               |
| h1H    | 2.603           | 1.987                   | 1.917                | 0.9813               |
| l2D2   | 2.073           | 2.037                   | 1.91                 | 0.948                |
| l2H    | 2.813           | 2.053                   | 1.913                | 0.9763               |
| mu     | 1.88            | 1.843                   | 1.833                | 0.9437               |
| nvdg   | 2.587           | 1.983                   | 1.903                | 0.98                 |
| var    | 0.4533          | -0.593                  | -0.6233              | -1.507               |

We note similar results to in the Laplace case, where the mu and l2D2 examples show a lower convergence rate, but all other methods are roughly equivalent. We note that the decline in EOC for most of the methods in the last step shown in figure 3.8 is most likely coincidental, since for instance an additional step for the h1D2 method leads to a higher EOC of 2.96. However on the whole we can say that outside of the var example, this problem is not significantly different in terms of convergence rates to the Laplace problem.
For the $H^1$ errors and EOCs in figures 3.9 and 3.10, we obtain mostly similar results to before, in that all methods appear to converge with an order of around 2.

Example 3.6.3 (Nondifferentiable (nonD) problem). Lastly we introduce a problem where $A$ is nondifferentiable due to a singularity at $(x, y) = (0.51, 0.61)$.

$$A = \begin{pmatrix} 1 & 0 \\ 0 & (x - 0.51)^2(y - 0.61)^2 + 1 \end{pmatrix}$$

(3.47)

The singularity’s location is chosen such that it does not lie directly on the mesh. This problem cannot be written in divergence form since $DA$ does not exist. Thus
only nonvariational methods are suitable.

Let us once more present the EOCs for the nonD problem in table 3.15. In

Table 3.15: Table of EOCs for the nonD example

| Method | $||\epsilon_h||$ EOC | $||\nabla \epsilon_h||$ EOC | $||\epsilon_h||_{D_1}$ EOC | $||\epsilon_h||_{D_2}$ EOC |
|--------|------------------|------------------|------------------|------------------|
| feng   | 2.783            | 1.963            | 1.91             | 1.007            |
| h1D2   | 2.793            | 1.977            | 1.977            | 0.9813           |
| h1H    | 2.753            | 1.97             | 1.933            | 1.003            |
| l2D2   | 2.453            | 2.033            | 1.957            | 0.9883           |
| l2H    | 2.88             | 1.993            | 1.923            | 1.003            |
| mu     | 1.847            | 1.913            | 1.91             | 0.9783           |
| nvdg   | 2.737            | 1.967            | 1.923            | 1.007            |
| var    | -1.013           | -0.113           | 0.8093           | 0.6153           |

figures 3.11 and 3.12 we plot the results for the $L^2$ norm. In figure 3.12 we observe a general slight decline in convergence in the final iteration, although the l2H and mu cases seems unaffected. Overall fairly similar results are obtained to the AD problem, though naturally the var case performs worse. For figures 3.13 and 3.14

![Figure 3.11: Plots of $L^2$ errors for the nonD problem](image1)

![Figure 3.12: Plots of $L^2$ EOCs for the nonD problem](image2)

which show the $H^1$ errors, we once again obtain similar results.

Overall between the methods we notice the following trends with regards to convergence rates.

- The l2D2 method appears to consistently perform better than the mu method despite the similarities between the two. This could possibly be the result of
different penalty terms.

- The $h1D2$ method seems to outperform the $l2D2$ method on all accounts, which is consistent with prior observations.

- The $l2H$ method is a clear improvement upon the $l2D2$ method, however the $h1H$ method is roughly equivalent to the $h1D2$ method for most tests.

- Overall the $feng$, $nvdg$, $h1D2$, $h1H$ and $l2H$ methods all seem to consistently perform the best in terms of convergence rates.

Other Polynomial Orders

We have considered quadratic basis functions for the previous computations, but let us now examine the results for $k = 1$ and $k = 3$ polynomials. First of all let us look at first order basis functions for the nondifferentiable example in table 3.16.

We note first of all that the optimal convergence decreases to approximately 2 for the $L2$ EOC and 1 for the $H1$ EOC. Despite our expectations, the $h1D2$ method still seems to maintain optimal convergence. On the other hand the $l2D2$ and $mu$ methods no longer converge for first order polynomials, and despite appearing to have an average EOC close to 1, the $var$ method probably does not either (the method appears to exhibit negative convergence in some steps). These results are somewhat more expected. We also note that the measures of the error which involve
Table 3.16: Table of EOCs for $k = 1$, for the nonD example

<table>
<thead>
<tr>
<th>Method</th>
<th>$|\varepsilon_h|$ EOC</th>
<th>$|\nabla \varepsilon_h|$ EOC</th>
<th>$|\varepsilon_h|_{D1}$ EOC</th>
<th>$|\varepsilon_h|_{D2}$ EOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>feng</td>
<td>1.992</td>
<td>0.9747</td>
<td>5.881e-05</td>
<td>-6.13e-05</td>
</tr>
<tr>
<td>h1D2</td>
<td>1.959</td>
<td>0.9758</td>
<td>5.881e-05</td>
<td>-6.13e-05</td>
</tr>
<tr>
<td>h1H</td>
<td>1.991</td>
<td>0.9763</td>
<td>5.881e-05</td>
<td>-6.13e-05</td>
</tr>
<tr>
<td>l2D2</td>
<td>-0.05036</td>
<td>-0.01919</td>
<td>5.881e-05</td>
<td>-6.13e-05</td>
</tr>
<tr>
<td>l2H</td>
<td>1.17</td>
<td>0.9728</td>
<td>5.881e-05</td>
<td>-6.13e-05</td>
</tr>
<tr>
<td>nvdg</td>
<td>1.987</td>
<td>0.9769</td>
<td>5.881e-05</td>
<td>-6.13e-05</td>
</tr>
<tr>
<td>var</td>
<td>0.843</td>
<td>0.9034</td>
<td>5.881e-05</td>
<td>-6.13e-05</td>
</tr>
</tbody>
</table>

$A : D^2$ no longer converge in this case (since the second derivative of a first order basis function is zero).

Now let us examine the results for $k = 3$ in table 3.17.

Table 3.17: Table of EOCs for $k = 3$, for the nonD example

<table>
<thead>
<tr>
<th>Method</th>
<th>$|\varepsilon_h|$ EOC</th>
<th>$|\nabla \varepsilon_h|$ EOC</th>
<th>$|\varepsilon_h|_{D1}$ EOC</th>
<th>$|\varepsilon_h|_{D2}$ EOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>feng</td>
<td>3.846</td>
<td>2.998</td>
<td>1.908</td>
<td>1.985</td>
</tr>
<tr>
<td>h1D2</td>
<td>3.724</td>
<td>2.963</td>
<td>2.055</td>
<td>1.987</td>
</tr>
<tr>
<td>h1H</td>
<td>3.948</td>
<td>3.008</td>
<td>1.914</td>
<td>1.984</td>
</tr>
<tr>
<td>l2D2</td>
<td>2.012</td>
<td>2.111</td>
<td>1.977</td>
<td>1.988</td>
</tr>
<tr>
<td>l2H</td>
<td>4.014</td>
<td>3.043</td>
<td>1.895</td>
<td>1.983</td>
</tr>
<tr>
<td>mu</td>
<td>3.953</td>
<td>3.014</td>
<td>2.964</td>
<td>1.988</td>
</tr>
<tr>
<td>nvdg</td>
<td>3.948</td>
<td>3.0</td>
<td>1.91</td>
<td>1.984</td>
</tr>
<tr>
<td>var</td>
<td>0.0612</td>
<td>0.07163</td>
<td>0.4753</td>
<td>-0.2388</td>
</tr>
</tbody>
</table>

Here we see as expected the EOCs mostly move up by 1 compared to the $k = 2$ case, and the results are fairly uniform in this regard. One exception to this is the l2D2 case which does not improve, though we do not have an analytical explanation as to why. Another observation is that the $\|\varepsilon_h\|_{D1}$ EOC does not improve overall, which may simply be due to the fact the polynomial order used to approximate the $H^{-1}$ norm has not changed.

### 3.6.3 Efficiency

Following the previous section which shows that many of the methods are roughly equivalent in terms of absolute errors and convergence rates, we now look towards other aspects with which to compare the methods.
Condition Numbers

One way of evaluating the efficiency of the schemes is to calculate the condition number of the system matrix and its growth after grid refinements. The condition number $C$ is defined by

$$C = \frac{\max \lambda_i}{\min \lambda_i},$$

where $\lambda_i$ are the eigenvalues of the system matrix. We compare the condition numbers of the $L^2$ and $H^{-1}$ minimization methods below. For these tests we will use Poisson’s equation (example 3.6.1) with polynomial order 2.

Table 3.18: Condition numbers for the $L^2$ minimization method

<table>
<thead>
<tr>
<th>Ele.</th>
<th>$\max \lambda_i$</th>
<th>$\min \lambda_i$</th>
<th>$C$</th>
<th>$\max \lambda_i$</th>
<th>$\min \lambda_i$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>11,788</td>
<td>0.001012</td>
<td>1.164e7</td>
<td>445.5</td>
<td>0.001037</td>
<td>429.493</td>
</tr>
<tr>
<td>128</td>
<td>51,072</td>
<td>2.530e-4</td>
<td>2.016e8</td>
<td>1782</td>
<td>2.593e-4</td>
<td>6.871e6</td>
</tr>
<tr>
<td>512</td>
<td>209,387</td>
<td>6.326e-5</td>
<td>3.310e9</td>
<td>7128</td>
<td>6.483e-5</td>
<td>1.099e8</td>
</tr>
<tr>
<td>2048</td>
<td>843,302</td>
<td>1.582e-5</td>
<td>5.332e10</td>
<td>25,513</td>
<td>1.621e-5</td>
<td>1.759e9</td>
</tr>
</tbody>
</table>

In table 3.18, we note that for the $L^2$ minimization method for both the piecewise and finite element Hessian versions, the condition number $C$ grows by roughly 16 each refinement, which is proportionate to $h^{-4}$.

Table 3.19: Condition numbers for the $H^{-1}$ minimization method

<table>
<thead>
<tr>
<th>Elements</th>
<th>$\max \lambda_i$</th>
<th>$\min \lambda_i$</th>
<th>$C$</th>
<th>$\max \lambda_i$</th>
<th>$\min \lambda_i$</th>
<th>$C$</th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>24.26</td>
<td>0.1891</td>
<td>128.3</td>
<td>16.61</td>
<td>0.1860</td>
<td>89.31</td>
</tr>
<tr>
<td>128</td>
<td>26.24</td>
<td>0.0475</td>
<td>552.8</td>
<td>17.07</td>
<td>0.04725</td>
<td>361.4</td>
</tr>
<tr>
<td>512</td>
<td>26.88</td>
<td>0.01189</td>
<td>2261</td>
<td>17.21</td>
<td>0.01188</td>
<td>1449</td>
</tr>
<tr>
<td>2048</td>
<td>27.06</td>
<td>0.002975</td>
<td>9096</td>
<td>17.25</td>
<td>0.002974</td>
<td>5798</td>
</tr>
</tbody>
</table>

In table 3.19, in contrast to the $L^2$ case, the $H^{-1}$ version has a condition number growth rate of approximately 4 or $h^{-2}$. This corresponds with the observations we made in section 3.4.2.

Now, let us compare the condition numbers of all the methods in figure 3.15. For clarity we also include lines which demonstrate how the condition number grows when proportionate to $h^{-2}$ and $h^{-4}$.

There are some notable observations to make. Firstly, we see that the feng
and nvdg appear to perform the best (obtaining almost identical condition numbers). Shortly behind this, the h1H method can be found, followed by the h1D2 method, and then the remaining methods.

The second observation is regarding the slope of the lines, which can be most easily observed via comparison to the additional dotted lines. Here we see that the methods form into two categories, with the nvdg, feng, h1H and h1D2 methods (showing growth numbers of about $h^{-2}$) comprising one, and the mu, l2H and l2D2 (with a growth of about $h^{-4}$) comprising the other.

From these results we can say that the nvdg and feng methods are ideal due to their low initial condition numbers. However we do note that they seem to scale the same way as the $H^{-1}$ method, meaning that for larger scale simulations there may not be a significant difference. These results also further show that the $L^2$ method is probably suboptimal.

**Iteration Count**

Generally speaking observing the growth in iterations made by the linear solver in solving the PDE is indicative of the efficiency of the method. In this case however
the use of preconditioning in our methods reduces this effect substantially. This can be seen in figure 3.16, where it can be seen the iterations rarely rise above 10.

Figure 3.16: Plot of the iteration count for the nonD problem

We also note that the choice of tolerance given in 3.5 directly affects the iteration count. Thus the use of different tolerances makes the above information unreliable.

Whilst this preconditioning can be turned off for comparison purposes, the minimization methods rely heavily on preconditioning and are significantly less efficient without it. Thus we defer instead to the analysis of the condition numbers from 3.6.3.

**Time Taken**

One other tool we can use to compare the different methods efficiency is the time taken to run the entire solving process. In some ways this is the most direct method for analysing the efficiency, however there is no guarantee the results will remain constant when factors such as parallel processing and different hardware are taken into account. The runtimes of the simulations used are also relatively small (not exceeding half a minute after pre-processing). We also note that a method could
take less time yet have a larger error, although our tests show that the $H^1$ error typically does not deviate by more than 5% between methods\textsuperscript{8}.

Nonetheless we compare the methods in figure 3.17 for a relative idea of the scales involved.

Figure 3.17: Plot of the total time taken for the nonD problem

We first note that the minimization methods all exhibit a longer initial step, most likely due to having to assemble additional schemes. Past this point they are more comparable, although the fastest methods are still the non-minimization methods, i.e. nvdg, mu and feng.

3.7 Nonlinear Problems

Until this point, we have looked purely at linear nonvariational PDEs taking the form $A : D^2 u = f$, as that has allowed us to most easily numerically compare and analyse different methods. Now we want to take a look at the more general case,

\[
F(D^2 u) = f(x, u, Du),
\]

\textsuperscript{8}Although we do note the $L^2$ error seems to differ by a more substantial amount, the scales are still comparable enough for efficiency comparisons to be valid.
where $F : \text{Sym}(\mathbb{R}^{d \times d}) \to \mathbb{R}$, and $f \in L^2(\Omega)$. This form is no longer restricted to linearly acting on each component of the Hessian, and allows for more complicated functions such as the Monge-Ampère equation and the Hamilton-Jacobi-Bellman equations.

In this section we will provide a primarily numerical-based look at a first step towards solving NV problems in a nonlinear setting in DUNE-FEMPY. As such we do not guarantee analytical convergence but instead focus on a practical look at whether we can solve nonlinear equations effectively. We leave the comparison of different methods for nonlinear problems as future work.

We will consider this section in two parts. First we will look at the details of the implementation, then we shall look at the implementation of the problems and their convergence rates.

### 3.7.1 Numerical Setup

Let us first of all describe the method we want to use to solve nonlinear nonvariational problems of the form $F(D^2u) = f$. To obtain the weak form of the PDE for the method, we will use a simple direct substitution of the Hessian for the finite element Hessian from section 3.5, equation (3.2). Thus we instead solve $F(H[u]) = f$.

Next we will then use Newton’s method to iteratively find the solution, i.e. we start from the well-known iterative formula,

$$0 = DF(H[u^n])(u^{n+1} - u^n) + F(H[u^n]),$$

(where $DF$ is the derivative of $F$). We then rearrange and invert $DF$ to obtain,

$$u^{n+1} = u^n - DF^{-1}(H[u^n])F(H[u^n]).$$

It then remains to choose a suitable $u^0$ (in this case we will just choose $u^0 = 0$ for the first two examples), and iterate until the residual error is sufficiently small.

We note that this method is applied automatically within the DUNE software framework, the derivative being calculated automatically from the bilinear form via
UFL differentiation. However in the case of the finite element Hessian, we must implement this manually, which we describe below.

We reuse the notation from the original numerical implementation of the FEH in section 3.5.2 by considering the FEH elementwise for $K \in T$ and entrywise for $1 \leq i, j \leq d$.

To use Newton’s method with finite elements, it is necessary to compute the components $H_{ij}^{K}[\bar{u}], H_{ij}^{K}[\varphi_{\mu}^{K}]$ and $H_{ij}^{K}[\varphi_{\mu}^{N}]$ for all $\mu \in \{1, ..., R\}$ (recall $\varphi_{\mu}$ are the basis functions for $u$). It will be convenient to define matrices derived from $l_{ij}^{K}$ (defined in (3.44)) which contain all the basis functions as follows.

$$L_{K}^{K} := (l_{ij}^{K}(\varphi_{\mu}^{K}, \Psi_{K}^{\nu}))_{\mu\nu}, \quad L_{K}^{N} := (l_{ij}^{K}(\varphi_{\mu}^{N}, \Psi_{N}^{\nu}))_{\mu\nu}.$$  

Note that if we let $\{e_{\mu}\}_{\mu=1,...,R}$ be the standard basis for $V_{h}$, we have $L_{ij}^{K} e_{\mu} = l_{ij}^{K}(\varphi_{\mu})$. We also use the notation $\bar{u} = \sum_{\mu} \bar{u}_{\mu} \varphi_{\mu}^{K} = \bar{u} \cdot \varphi_{K}$, where $\varphi_{K} = (\varphi_{\mu}^{K})_{\mu=1,...,R}$. Then we can rewrite (3.45) using the above forms, giving us equations for $H_{ij}^{K}[\varphi_{\mu}^{K}], H_{ij}^{K}[\varphi_{\mu}^{N}]$ and $H_{ij}^{K}[^{\bar{u}}]$ as follows.

$$H_{ij}^{K}[\varphi_{\mu}^{K}] = M_{K}^{-1} L_{ij}^{KK} e_{\mu} \cdot \Psi_{K},$$

$$H_{ij}^{N}[\varphi_{\mu}^{N}] = M_{K}^{-1} L_{ij}^{KN} e_{\mu} \cdot \Psi_{K},$$

$$H_{ij}^{K}[\bar{u}] = M_{K}^{-1} L_{ij}^{KK} \bar{u}^{K} \cdot \Psi_{K} + \sum_{N \in N^{K}} M_{K}^{-1} L_{ij}^{KN} \bar{u}_{N}^{N} \cdot \Psi_{K}.$$  

In algorithmic form we have the following.

**Algorithm 3.7.1.** To compute $H_{ij}^{K}[\bar{u}], H_{ij}^{K}[\varphi_{\mu}^{K}]$ and $H_{ij}^{K}[\varphi_{\mu}^{N}]$ we do the following.

### 3.7.2 Effectiveness and Convergence Rates

We start off by looking at an example of a nonlinear problem that can be written in variational form.
// Part 1 - $L^{kk}$ and $L^{kn}$ construction

for $\mu = 1$ to $R$ do
  for $\nu = 1$ to $S$ do
    $(L^{k,k})_{\mu\nu} = - \int_K \partial_i \varphi^K_{\mu} \partial_j \Psi^K_{\nu}$
    for all $N \in N^K$ do
      $(L^{k,k})_{\mu\nu} += \int_{e_N} (\partial_i \varphi^N_{\mu} n^K_{j} \Psi^K_{\nu} + \varphi^K_{\mu} n^K_{j} \partial_j \Psi^K_{\nu})$
      $(L^{k,n})_{\mu\nu} = \int_{e_N} (\partial_i \varphi^N_{\mu} n^K_{j} \Psi^K_{\nu} - \varphi^K_{\mu} n^K_{j} \partial_j \Psi^K_{\nu})$
    end
  end
end

Example 3.7.1 (p-Laplace Equation). We consider the p-Laplace equation, i.e.

$$-\nabla \cdot (|\nabla u|^{p-2} \nabla u) = f, \quad \text{in } \Omega,$$

$$u = 0, \quad \text{on } \partial \Omega,$$

where $|\nabla u|^{p-2}$ is defined as

$$|\nabla u|^{p-2} = (d + \nabla u \cdot \nabla u)^{\frac{p-2}{2}},$$

and $d = 0.001$ and $1 < p < \infty$. The nonvariational bilinear form is simply

$$\int_{\Omega} -\nabla \cdot (|\nabla u|^{p-2} \nabla u) \varphi \, dx = \int_{\Omega} f \varphi \, dx.$$

This is implemented as a scheme in DUNE-FEMPY as follows, for a chosen value of $p = 1.7$, and exact solution $u = \sin(2\pi x) \sin(2\pi y)$.

Code Listing 3.11: The p-Laplace problem

1 d = 0.001
2 p = 1.7
3 $\text{norm}_{\text{grad}u} = \text{pow}(d + \text{inner}(\text{grad}(u), \text{grad}(u)), (p-2)/2)$
// Part 2 - Obtain degrees of freedom

H_{ij}^{K[\bar{u}]} = M_{K}^{-1}L_{ij}^{KK} \bar{u}^{K}

for \( \mu = 1 \) to \( R \) do

\quad H_{ij}^{K[\varphi_{\mu}^{K}]} = M_{K}^{-1}L_{ij}^{KK} e_{\mu}

end

forall \( N \in N^{K} \) do

\quad H_{ij}^{K[\bar{u}]^{N}+} = M_{K}^{-1}L_{ij}^{KN} \bar{u}^{N}

for \( \mu = 1 \) to \( R \) do

\quad H_{ij}^{K[\varphi_{\mu}^{N}]+} = M_{K}^{-1}L_{ij}^{KN} e_{\mu}

end

end

// Part 3 - Calculate result

for \( \nu = 1 \) to \( S \) do

\quad H_{ij}^{K[\bar{u}]^{\nu}+} = H_{ij\nu}^{K[\bar{u}]\Psi_{\nu}^{K}}

for \( \mu = 1 \) to \( R \) do

\quad H_{ij}^{K[\varphi_{\mu}^{\nu}]+} = H_{ij\nu}[\varphi_{\mu}^{K}]\Psi_{\nu}^{K}

forall \( N \in N^{K} \) do

\quad H_{ij}^{K[\varphi_{\mu}^{N}]+} = H_{ij\nu}[\varphi_{\mu}^{N}]\Psi_{\nu}^{K}

end

end

4 pLaplace_u = grad(norm_grad_u*grad(u))[0, 0, 0] \+
5 + grad(norm_grad_u*grad(u))[0, 1, 1]
6 a = (-inner(pLaplace_u, v[0])) * dx
7 b = ufl.replace(a, {u: exact})
8 scheme = create.scheme("nv", space, [a==b, dirichletBC], \
Note that we have obtained the RHS by substituting the exact solution for $u$. We run our method with this form and the same numerical setup as before ($[0, 1]^2$ domain, 2nd order basis functions), and this results in the table of EOCs 3.20.

Table 3.20: Table of EOCs for the nonvariational p-Laplace

| Elements | $||e_h||$ | EOC | $||\nabla e_h||$ | EOC |
|----------|----------|-----|----------------|-----|
| 32       | 0.04266  | -   | 0.9627         | -   |
| 128      | 0.004414 | 3.273 | 0.2594         | 1.892 |
| 512      | 0.0005047 | 3.129 | 0.0668         | 1.957 |
| 2048     | 6.059e-05 | 3.058 | 0.01684        | 1.988 |
| 8192     | 8.107e-06 | 2.902 | 0.004219       | 1.997 |

Here we note that we obtain the expected convergence rates of 3 for the $L^2$ norm and 2 for the $H^1$ norm. We can also confirm the correctness of this result by comparing it to the variational form of this method, i.e.

$$\int_\Omega |\nabla u|^p - 2 \nabla u \cdot \nabla \varphi \, dx = \int_\Omega f \varphi \, dx.$$  

This gives us the following errors in table 3.21.

Table 3.21: Table of EOCs for the variational p-Laplace

| Elements | $||e_h||$ | EOC | $||\nabla e_h||$ | EOC |
|----------|----------|-----|----------------|-----|
| 32       | 0.03386  | -   | 0.9275         | -   |
| 128      | 0.003885 | 3.123 | 0.2591         | 1.84 |
| 512      | 0.0004783 | 3.022 | 0.06681        | 1.956 |
| 2048     | 5.999e-05 | 2.995 | 0.01684        | 1.988 |
| 8192     | 8.061e-06 | 2.896 | 0.004219       | 1.997 |

As expected we obtain almost the same error results for the variational version compared to the nonvariational version.

**Example 3.7.2** (Nonlinear NV Equation). As a second example we implement a...\footnote{We note that quasinorms (see e.g. Barrett and Liu [1994]) could potentially be more suitable for nonlinear problems, yet we will follow the procedure of Lakkis and Pryer [2012].}
simple nonlinear problem that can only be written in nonvariational form.

\[ \sin(\Delta u) + 2\Delta u = f, \quad \text{in } \Omega, \]
\[ u = 0, \quad \text{on } \partial\Omega. \]

In DUNE-FEMPY this corresponds to the following scheme.

Code Listing 3.12: The simple nonvariational nonlinear problem

```python
1 laplace = grad(grad(u))[0, 0, 0] + grad(grad(u))[0, 1, 1]
2 a = inner(sin(laplace) + 2*laplace, v[0])*dx
3 b = ufl.replace(a, {u: ufl.as_vector(exact)} )
4 scheme = create.scheme("nv", space, [a==b, dirichletBC],
constraints='dirichlet')
```

We once again solve the code and compute the errors in 3.22.

Table 3.22: Table of EOCs for the simple nonlinear problem

| Elements | ||\|e_h|| | EOC | ||\|\nabla e_h|| | EOC |
|----------|-----------------|--------|-----------------|--------|
| 32       | 0.0314          | -      | 0.9254          | -      |
| 128      | 0.003706        | 3.083  | 0.2588          | 1.838  |
| 512      | 0.000779        | 2.25   | 0.06711         | 1.947  |
| 2048     | 0.0002034       | 1.938  | 0.01705         | 1.977  |
| 8192     | 1.62e-05        | 3.65   | 0.004278        | 1.995  |

In this situation the convergence rate appears to be unstable but nonetheless converges with roughly the same results as before.

Example 3.7.3 (Monge-Ampère equation). Finally we move onto the Monge-Ampère equation.

\[ \det(D^2u) = f, \quad \text{in } \Omega, \]
\[ u = 0, \quad \text{on } \partial\Omega. \]

Here we will use the exact solution

\[ u = e^{2((x-0.5)^2+(y-0.5)^2)}. \]
We note that for the solving of this equation, instead of Newton’s method we instead use a specific iterative method from [Benamou et al., 2010, Def. 2.1.], i.e.

\[ u^{n+1} = \Delta^{-1} \left( \sqrt{(\Delta u^n)^2 + 2(f - \det(H[u^n]))} \right) \]  

(3.48)

We are able to implement (3.48) by taking the Laplacian to the LHS and solving weakly, as shown below.

Code Listing 3.13: Monge-Ampère iterative method

```python
1 a = laplace(u)*v[0]*dx
2 b = sqrt(laplace(uOld)**2 + 2*(f - detH(uOld)))*v[0]*dx
3 scheme0 = create.scheme("nv", space, [a==b, dirichletBC],
                        solver=solver, polOrder=space.order)
```

After applying this method and solving it, this results in the EOC table 3.23. We see an EOC of approximately 2 for the \( L^2 \) error, which we note is consistent with the results from the paper the method comes from (Benamou et al. [2010]), where they observe \( \mathcal{O}(h^2) \) convergence for a smooth solution.

| Elements | ||\( e_h \)|| ||\( \nabla e_h \)|| EOC  
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>32</td>
<td>0.0003722</td>
<td>0.01135</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>128</td>
<td>8.454e-05</td>
<td>2.139</td>
<td>0.003261</td>
<td>1.799</td>
</tr>
<tr>
<td>512</td>
<td>1.980e-05</td>
<td>2.094</td>
<td>0.0006888</td>
<td>2.243</td>
</tr>
<tr>
<td>2048</td>
<td>4.822e-06</td>
<td>2.038</td>
<td>0.0001897</td>
<td>1.860</td>
</tr>
<tr>
<td>8192</td>
<td>1.167e-06</td>
<td>2.047</td>
<td>4.220e-05</td>
<td>2.1685</td>
</tr>
</tbody>
</table>

Table 3.23: Table of EOCs for the Monge-Ampère equation
Chapter 4

Conclusion

4.1 Achieved Goals

The first goal we have achieved in this work is the design and creation of DUNE-FEMPY. Primarily we sought to design a Python interface for finite element methods (via the DUNE-Fem module) which facilitates simpler code design and rapid prototyping. However to justify the existence of the package beyond the confines of DUNE development and to show its merit in comparison to other similar Python front-end packages we have endeavored to add additional functionality and do things in a unique way. For instance we have made efforts to maintain the similarity between the Python and C++ code structure so that translation between the two for efficiency reasons is easier. We have also facilitated the writing of additional modules in C++ that do things not currently available in DUNE-FEMPY, such as the DUNE-FEMNV module for nonvariational problems. The fact that such modules can be written independently and added to the interface easily means external development is encouraged and not limited by what is merely available to the user. Finally we have added many of the features common to finite element packages such as grid adaptivity and the integration of external solvers, which we have demonstrated throughout section 2.

With regards to nonvariational PDEs, our primary goal was to implement a new method for solving this class of problems based on minimization, that attempts
to improve upon existing methods in terms of analytical results and performance. We have shown a full mathematical derivation for the method and derived results for the existence and uniqueness and the error convergence. We then showed a derivation for an improvement to the method in terms of the DG finite element Hessian. In the numerical results we were able to implement all the methods in the same setting and compare them directly. We were able to demonstrate there was in fact an improvement when considering the $H^{-1}$ version of the method compared to the $L^2$ case, although some other existing methods displayed similar results. In a preliminary look at the nonlinear case, we showed that using the finite element Hessian and Newton’s method that we can also successfully solve such equations, and that an extension to the analysis would be feasible.

### 4.2 Future Work

Principally with regards to future work, we would like to continue the nonlinear analysis of nonvariational problems. Of particular note is that the minimization method could be brought to the nonlinear case by instead considering instead,

$$\frac{1}{2} \| F(H[u]) + f \|_{X'}^2 \to \min.$$ 

However we do note that this would be a nontrivial extension considering $F$ is nonlinear, so the Euler-Lagrange equation would be different. Nonetheless this would be a necessary extension to make in order to fully apply the method to interesting NV problems.

In addition to this, it would also be desirable to consider direct applications of the Monge-Ampère equation having constructed a working example, e.g. $r$-adaptivity on the sphere (see McRae et al. [2016]). In general the consideration of other applicable nonlinear NV problems would be an ideal extension as well.

In terms of the program itself, one useful feature for simplicity and transparency reasons would be the ability to write the finite element Hessian operator directly into the Python interface. Currently the operator is created externally.
through DUNE code that is not visible to the user unless they look inside the corresponding operator file, and even then it is difficult to casually read through and modify. If this operator could be written on the Python side it would be much more transparent and easy to work with. Furthermore this functionality could also be used for other DG methods, as other lifting operators are as of yet unavailable.

Regarding DUNE-FEMPY, there naturally remain many different directions the project could be taken. In particular integration with other DUNE modules would be a high priority in order to add new features to the code and encourage continued development. In particular the DUNE-FEM-DG module (see Dedner et al. [2017]) could be added to increase the complexity of DG schemes available, whilst compatibility with DUNE-PDELAB would be desirable since it accomplishes a similar thing of offering high-level abstraction for DUNE code.
Chapter 5

Bibliography


G. Alzetta, D. Arndt, W. Bangerth, V. Boddu, B. Brands, D. Davyдов, R. Gassmoeller, T. Heister, L. Heltai, K. Kormann, M. Kronbicler, M. Maier, J-


Appendix A

Running this code

The code in this thesis can be run by using the accompanying docker repository. Docker is a software package that allows one to run programs from within a self-contained container without having to download extra software. Provided that docker has been installed (using `sudo apt install docker.io` or some equivalent), the container can be accessed on the command line in linux via

```bash
$ docker run --rm -v dune:/dune -p 127.0.0.1:8888:8888 lloydconnellan/thesis
```

The examples can then be run by opening a web browser and typing the address `http://127.0.0.1:8888`, which opens up a Jupyter notebook. The password for the login is `dune`.
Appendix B

Derivation of Forchheimer Model

The origin of this equation stems from Darcy’s law, an equation that describes flow through porous media, and is applied regularly to groundwater flow models.

\[-\nabla p = \frac{\mu}{\kappa} v,\]

where \( p, v, \mu \) and \( \kappa \) are the pressure, velocity, absolute viscosity and permeability. For situations where the Reynolds number is greater than \( \sim 10 \), inertia begins to have an effect on the system, which is accounted for in the Darcy-Forchheimer equation. In its most general form we have the following.

\[-\nabla p = \sum_{i=0}^{N} a_i |v|^{\alpha_i} v,\]

where \( a_i \) and \( \alpha_i \) are obtained empirically. Through some manipulations, we can simplify this to an equation for just the pressure \( \rho \).

\[\rho_t - \nabla \cdot (K(|\nabla \rho|)\nabla \rho) = f,\]
where the function $K : \mathbb{R}^+ \to \mathbb{R}^+$ is dependent on the $a_i$ and $a_i$ above. Adding in boundary data and initial values gives us the **boundary value problem**.

\[
\begin{align*}
\rho_t - \nabla \cdot (K(|\nabla \rho|)\nabla \rho) &= f, & \text{in } \Omega \times [0, T], \\
\rho(x, 0) &= \rho^0(x), & \text{in } \Omega, \\
K(|\nabla \rho|)\nabla \rho \cdot n &= g(x), & \text{on } \partial \Omega \times [0, T],
\end{align*}
\]

where $\rho^0$ and $g$ are initial and boundary data given. Thus the **weak form** or variational formulation follows.

\[
(\rho_t, \varphi_h) + (K(|\nabla \rho|)\nabla \rho, \nabla \varphi_h) = <g, \varphi_h> + (f, \varphi_h), \quad \varphi_h \in V_h,
\]

with $\rho(x, 0) = \rho^0(x)$. Finally it remains to discretize the equation in time. Let the time domain $I = [0, T]$ be divided into $N$ intervals $t_0 = 0 < t_1 < \cdots < t_N = T$ such that $\Delta t = t_n - t_{n-1}$ and $\rho^n = \rho(x, t_n)$. Then we have the time discretized PDE.

\[
\left( \frac{\rho^{n+1} - \rho^n}{\Delta t}, \varphi_h \right) + \left( \frac{1}{2} K(|\nabla \rho^{n+1}|)\nabla \rho^{n+1} + \frac{1}{2} K(|\nabla \rho^n|)\nabla \rho^n, \nabla \varphi_h \right) \\
= <g, \varphi_h> + (f, \varphi_h), \quad \varphi_h \in V_h,
\]

where we have used Heun’s method (i.e. half-implicit/half explicit) on the term with $K$. We can see that this is the same as equation (2.5) after replacing $\rho$ with $u$. 149
Appendix C

C++ Version of Forchheimer Example

Here we present the C++ version of the Forchheimer example from section 2.1 using DUNE-FEM, that we compare to the Python version in section 2.3.3.

```cpp
#include <config.h>

#include <iostream>
#include <complex>
#include <ctime>

#include <dune/grid/yaspgrid.hh>
#include <dune/grid/io/file/dgfparservdgysap.hh>

#include <dune/fem/grid/gridpartadapter.hh>
#include <dune/fem/space/lagrange.hh>
#include <dune/fem/function/adaptivefunction.hh>
#include <dune/fem/function/localfunction/const.hh>
#include <dune/fem/function/localfunction/bindable.hh>
#include <dune/fem/solver/kryloinversionoperators.hh>
#include <dune/fem/operator/linear/spoperator.hh>
#include <dune/fem/io/file/dataoutput.hh>
```
// include header of elliptic solver
#include <dune/fem/schemes/elliptic.hh>
#include <dune/fem/schemes/femscheme.hh>

// include generated model
#include <forchheimer/forchheimer.hh>

template <class GridPart>
struct Initial : public Dune::Fem::BindableGridFunction< GridPart, Dune::Dim<1> >
{
  typedef Dune::Fem::BindableGridFunction<GridPart, Dune::Dim<1> > Base;
  using Base::Base;
  template <class Point>
  void evaluate(const Point &xhat, typename Base::RangeType &ret) const
  {
    auto x = Base::global(xhat);
    ret[0] = 1./2.*x.two_norm2() - 1./3.*(pow(x[0],3) - pow(x[1],3)) + 1.;
  }
  unsigned int order() const { return 5; }
  std::string name() const { return "Initial"; }
};

int main ( int argc, char **argv )
try
{
  Dune::Fem::MPIManager::initialize( argc, argv );
  Dune::Fem::Parameter::append( argc, argv );
  for( int i = 1; i < argc; ++i )
    Dune::Fem::Parameter::append( argv[ i ] );
  Dune::Fem::Parameter::append( "parameter" );
  typedef Dune::YaspGrid<2> HGridType ;
const std::string gridkey =
  Dune::Fem::IOInterface::defaultGridKey(HGridType::dimension);

const std::string gridfile = Dune::Fem::Parameter::getValue<std::string>(gridkey);
HGridType& grid = *gridPtr;

auto gridView = grid.leafGridView();
Dune::FemPy::GridPartAdapter<decltype(gridView)> gridPart(gridView);

typedef Dune::Fem::FunctionSpace<double, double,
  HGridType::dimensionworld, 1 > FunctionSpaceType;
Dune::Fem::LagrangeDiscreteFunctionSpace<FunctionSpaceType,decltype(gridPart),2>
  space(gridPart);
Dune::Fem::AdaptiveDiscreteFunction<decltype(space)>
  solution("solution",space);
decltype(solution) previous(solution);

Dune::Fem::interpolate(Initial<decltype(gridPart)>(gridPart),solution);

forchheimer::Model<decltype(gridPart),typename
dclttepe(previous)::LocalFunctionType> model(
  previous.localFunction());

typedef FemScheme< DifferentiableEllipticOperator<
  Dune::Fem::SparseRowLinearOperator<decltype(solution),decltype(solution)>>
  Dune::Fem::KrylovInverseOperator<decltype(solution)>> >
  SchemeType;
SchemeType scheme(space, model);

std::tuple< decltype(solution)*>* ioTuple( &solution );
Dune::Fem::DataOutput<HGridType,decltype(ioTuple)> dataOutput(
  grid, ioTuple );
dataOutput.writeData( 0 );

Dune::Fem::GridTimeProvider<HGridType> timeProvider(grid);
double timeStep = 0.05;
model.dt() = timeStep;

auto start = std::clock();
for( timeProvider.init( timeStep ); timeProvider.time() < 1.0;
    timeProvider.next( timeStep ) )
{
    previous.assign(solution);
    model.t() = timeProvider.time();
    scheme.solve( solution );
}
std::cout << double( std::clock() - start ) / CLOCKS_PER_SEC <<
std::endl;
dataOutput.writeData( 1 );
return 0;
} catch( const Dune::Exception &exception )
{
    std::cerr << "Error: " << exception << std::endl;
    return 1;
}
Appendix D

List of Dune-Python modules

Here we list the different modules that are available to DUNE-PYTHON and DUNE-FEMLPY at the time of writing. We will divide them by component into different sections that reflect the structure we have previously introduced.

D.1 Grids

Grids by default take the following form.

```
grid = create.grid('class-name', constructor, dimgrid=None,
                   dimworld=None)
```

Where in particular we have the following arguments.

1. `class-name`: One of the strings from the table below.
2. `constructor`: Either a dgf dune grid format file, a gmesh file, or a preset object similar to what is demonstrated in section 2.1.1.
3. `dimgrid` (optional): The dimension of the grid.
4. `dimworld` (optional): The dimension of the space the grid is in.

The dimensions of the grid do not have to be passed to the constructor if they can be determined from the `constructor` argument.
The table below shows a list of possible grid implementations for which binding are available at the time of writing.

<table>
<thead>
<tr>
<th>Class</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'aluconform'</td>
<td>dune.alugrid</td>
<td>Confirming simplex grid</td>
</tr>
<tr>
<td>'alucube'</td>
<td>dune.alugrid</td>
<td>Nonconforming cube grid</td>
</tr>
<tr>
<td>'alusimplex'</td>
<td>dune.alugrid</td>
<td>Nonconforming simplex grid</td>
</tr>
<tr>
<td>'oned'</td>
<td>dune.grid</td>
<td>One-dimensional grid</td>
</tr>
<tr>
<td>'polygon'</td>
<td>dune.polygongrid</td>
<td>Polygonal grid</td>
</tr>
<tr>
<td>'ug'</td>
<td>dune.uggrid</td>
<td>Hybrid nonconforming unstructured grid</td>
</tr>
<tr>
<td>'yasp'</td>
<td>dune.grid</td>
<td>Structured grid</td>
</tr>
<tr>
<td>'spgrid'</td>
<td>dune.spgrid</td>
<td>Structured grid</td>
</tr>
</tbody>
</table>

Grids can also be constructed to provide a different grid view:

Code Listing D.2: Creating a custom grid view

```python
view = create.view('class-name', grid)
```

The first argument can be any of the ones allowed for the grid construction (in which case the same `LeafGridView` is constructor as when directly constructing the grid). In addition the following views can be constructed.

<table>
<thead>
<tr>
<th>Class</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'adaptive'</td>
<td>dune.fem</td>
<td>Adaptive grid view</td>
</tr>
<tr>
<td>'filtered'</td>
<td>dune.fem</td>
<td>Filtered grid view for separated domains. Usage: subGrid = create.view(&quot;filtered&quot;, grid, filter, domainID)</td>
</tr>
<tr>
<td>'geometry'</td>
<td>dune.fem</td>
<td>Convert a coordinate function into a grid view. Usage: geometry = create.view(&quot;geometry&quot;, function)</td>
</tr>
</tbody>
</table>

D.2 Spaces

Now we move on to spaces. By default, they take the form
Code Listing D.3: The default form for space creation

```
1 space = create.space('class-name', grid, dimrange=1, order=1,
2 storage='fem', field='double')
```

where we have the following arguments.

1. `class-name`: A string from the table below.
2. `grid`: A grid object from above.
3. `dimrange` (optional): The range dimension of the space.
4. `order` (optional): The polynomial order of the basis functions.
5. `storage` (optional): `fem` by default. Specifies the storage used for discrete functions (as shown in D.3).
6. `field` (optional): The field of the range space (`double` or `complex`).

### Table D.3: Discrete Functions

<table>
<thead>
<tr>
<th>Class</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'adaptive'</td>
<td>dune.fem</td>
<td>An adaptive function</td>
</tr>
<tr>
<td>'eigen'</td>
<td>dune.fem</td>
<td>Function from the eigen package</td>
</tr>
<tr>
<td>'fem'</td>
<td>dune.fem</td>
<td>The default DUNE-Fem function</td>
</tr>
<tr>
<td>'istl'</td>
<td>dune.fem</td>
<td>Function from DUNE-Istl</td>
</tr>
<tr>
<td>'petsc'</td>
<td>dune.fem</td>
<td>Function from the PETSc package</td>
</tr>
<tr>
<td>'petscadapt'</td>
<td>dune.fem</td>
<td>An adaptive function using PETSc</td>
</tr>
</tbody>
</table>

Available spaces are:

### Table D.4: Spaces
<table>
<thead>
<tr>
<th>Class</th>
<th>Module</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>’bdm’</td>
<td>dune.fem</td>
<td>Space for Brezzi-Douglas Marini elements</td>
</tr>
<tr>
<td>’combined’</td>
<td>dune.fem</td>
<td>Discrete function space formed from a tuple of DF spaces</td>
</tr>
<tr>
<td>’dglagrange’</td>
<td>dune.fem</td>
<td>DG space using Lagrange basis functions</td>
</tr>
<tr>
<td>’dglegendre’</td>
<td>dune.fem</td>
<td>DG space with elementwise Legendre tensor product basis function. This space allows for an additional argument: hierarchical which defaults to True. This argument determines if the basis functions are sorted hierarchically according to their polynomial order or lexicographically.</td>
</tr>
<tr>
<td>’dglegendrehp’</td>
<td>dune.fem</td>
<td>’dglegendre’ with hp-adaption</td>
</tr>
<tr>
<td>’dgonb’</td>
<td>dune.fem</td>
<td>DG space with elementwise orthonormal basis functions</td>
</tr>
<tr>
<td>’dgonbhp’</td>
<td>dune.fem</td>
<td>’dgonb’ with hp-adaptation</td>
</tr>
<tr>
<td>’finitevolume’</td>
<td>dune.fem</td>
<td>Space for the finite volume method</td>
</tr>
<tr>
<td>’lagrange’</td>
<td>dune.fem</td>
<td>Space for Lagrange elements</td>
</tr>
<tr>
<td>’p1bubble’</td>
<td>dune.fem</td>
<td>P1 space with bubble elements</td>
</tr>
<tr>
<td>’product’</td>
<td>dune.fem</td>
<td>Discrete function space formed from a tuple of DF spaces</td>
</tr>
<tr>
<td>’rannacherturek’</td>
<td>dune.fem</td>
<td>Space for Rannacher-Turek elements</td>
</tr>
</tbody>
</table>

### D.3 Grid Function

Grid functions can be constructed in a variety of ways, though the explicit way to make a grid function is

```python
create.function('class-name', grid, 'function-name', order,
```
with the following arguments.

1. *class-name*: A string from the table below.

2. grid: A grid object from D.1.

3. *function-name*: A string for the name of the created function.

4. order: The order of the function used for quadrature.

5. constructor: An object described in the table below used to construct the function.

For the below table we list the available strings for *class-name* together with the compatible argument for the constructor parameter.

**Table D.5: Grid Functions**

<table>
<thead>
<tr>
<th>Class</th>
<th>Module</th>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>'cpp'</td>
<td>dune.fem</td>
<td>A C++ string, e.g. &quot;value[ 0 ] = 2;&quot;</td>
</tr>
<tr>
<td>'global'</td>
<td>dune.fem</td>
<td>A Python function or lambda taking the global coordinate as a single argument</td>
</tr>
<tr>
<td>'levels'</td>
<td>dune.fem</td>
<td>This is a special piecewise constant grid function used for visualization which returns the level of each element</td>
</tr>
<tr>
<td>'local'</td>
<td>dune.fem</td>
<td>A Python function or lambda taking an entity and a local coordinate as arguments</td>
</tr>
<tr>
<td>'numpy'</td>
<td>dune.fem</td>
<td>A numpy expression</td>
</tr>
<tr>
<td>'partitions'</td>
<td>dune.fem</td>
<td>This is a special piecewise constant grid function used for visualization which returns the partition number for each element</td>
</tr>
<tr>
<td>'ufl'</td>
<td>dune.fem</td>
<td>A UFL expression</td>
</tr>
</tbody>
</table>

In addition *discrete* can be used to construct discrete functions. But typically these are made using the `space.interpolate(expression)` syntax, using the stor-
age type for the space by default. The arguments for the `interpolate` method are the same used to construct general grid functions given in the previous table.

## D.4 Schemes and Operators

As discussed in section 2.1.4, schemes can be constructed directly with a UFL form and contain the method for solving the PDE. Additionally as shown in section 2.3, it is also possible to create models and operators that store the operator separately. While schemes have to have identical domain and range spaces, operators can map between different spaces. We review this below.

**Code Listing D.5: The default form for scheme creation**

```python
scheme = create.scheme('class-name', equation, space,
parameters=dict, solver='solver-name')
```

1. `'classname'`: A string from the table below.

2. `equation`: A UFL equation (a == b), or a model object. In addition a tuple or list can be used here where the first entry is the equation and further arguments can provide Dirichlet boundary conditions.

3. `space`: A space object from D.2. (This is optional if the trial/test UFL functions are initialized with a DUNE-FEMPY discrete function space, as in code listing 2.9).

4. `parameters` (optional): A dictionary of DUNE parameters that can be used to specify things like the solver behaviour, e.g. `{'fem.solver.newton.tolerance':1e-3}`.

5. `solver` (optional): `'fem'` by default. Used to specify the solver used, from D.6.

The two main schemes available in DUNE-FEMPY are `galerkin` and `h1`. The distinction between them is described in section 2.3. In addition the DUNE-NVDG module provides an `nvgd` scheme.
There are a number of parameters that can be passed to the scheme to influence the solving procedure. Most importantly tolerances for the iterative solvers can be provided and verbosity can be turned on and off. In addition preconditioners can be set via the parameters. Available options depend on the storage backend used to construct the space. For the istl backend available options include: none, ssor, sor, ilu-0, ilu-n, gauss-seidel, jacobi and amg-ilu-0. They are set using the following syntax.

Code Listing D.6: How preconditioning is set via parameters

```python
from dune.fem import parameter
parameter.append({"istl.preconditioning.method": "ilu",
        "istl.preconditioning.iterations": 1,
        "istl.preconditioning.relaxation": 1.2})
```

For the petsc backend options include: none, oas, sor, jacobi, hypre, ml, ilu, icc, and lu. Which of these can actually be used will depend on the PETSc implementation, e.g. hypre requires that PETSc was build with support for the hypre package.

Finally we list the possible solving methods available that can be selected during scheme creation above. There is also the possibility of constructing an operator that can be applied using the `__call__` method and linearised using the jacobian method but does not provide a `solve` method. This is especially of interest in the case where the operator maps between different spaces. Creating an operator is similar to the scheme construction.

<table>
<thead>
<tr>
<th>Class</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>'bicgstab'</td>
<td>Biconjugate gradient method</td>
</tr>
<tr>
<td>'cg'</td>
<td>Conjugate gradient method (for symmetric problems only)</td>
</tr>
<tr>
<td>'gmres'</td>
<td>Generalized minimal residual method</td>
</tr>
<tr>
<td>'minres'</td>
<td>Minimal residual method</td>
</tr>
<tr>
<td>'suitesparse'</td>
<td>This is a tuple where the second argument determines which suitesparse solver to used</td>
</tr>
<tr>
<td>'superlu'</td>
<td>Solve method from SuperLU package</td>
</tr>
</tbody>
</table>
Code Listing D.7: The default form for operator creation

```python
scheme = create.operator('class-name', equation, domainSpace,
                         rangeSpace)
```