Differentiable abstractions for integrating data with finite element models

Reuben William Nixon-Hill

Department of Mathematics
Imperial College London

Submitted for the degree of
Doctor of Philosophy of Imperial College London
and the Diploma of Imperial College

Abstract

Mathematical models are a foundational part of how mathematicians, scientists and engineers describe and study the behaviour of systems. These models typically include and interact with data: Experimental measurements are necessary inputs to a modelled physical system to produce predictive data outputs. Within a model, data in one part of the modelled system may have a relationship with another part of the modelled system. One may even wish to combine models to study a larger system: these typically interact by producing and consuming each other's data.

Finite Element Methods (FEMs) are a popular method of computing mathematical models which involve systems of equations defined on some domain. In this work, new abstractions are introduced for (a) representing arbitrary point data as finite element functions on disconnected meshes of vertices, and for (b) interacting with that data via interpolation operations which represent point evaluations. These abstractions are consistent with the finite element method paradigm of finite element functions on meshes. The interpolation operations are differentiable, allowing inverse problems to be solved involving point data.

The abstractions are sufficiently general that they allow one to reason about more generic data, both within and external to a given finite element model. The differentiable point evaluation operation allows differentiable interpolation operations between finite element functions on arbitrary coincident meshes.

Whilst these abstractions are generally applicable across many finite element method software packages, they are demonstrated here with an implementation in Firedrake. Firedrake is an MPI-parallelised code generation system which solves variational problems using the finite element method. The implementation is parallel safe and maintains the high level of abstraction described. Demonstrated applications include point-data assimilation, point forcing, model diagnostics gathering, and model coupling. Improvements to the abstractions Firedrake uses for specifying variational problems and reasoning about finite elements are also implemented.

Acknowledgements

This work is the culmination of many years of effort, and I would like to thank the many people who have helped me along the way. Firstly my primary supervisor, Professor David Ham, for his guidance and support throughout my PhD. Thank you for enthusiatically embracing me into the world of scientific computing, for providing help when I needed it, and for supporting me through illness. I would also like to thank my secondary supervisor, Professor Colin Cotter, for his friendly help and advice. Thank you for always finding time to discuss whatever mathematical or general-life queries I had, and for your encouragement and support. Many thanks are owed to various current and former members of the Firedrake team, including but not limited to: Lawrence Mitchell, Jack Betteridge, Koki Sagiyama, Connor Ward and Dan Shapero. Thank you for your code reviews, for your help with debugging, for your patience with my questions, your support and your advice.

I had the misfortune of being diagnosed with a chronic illness during my PhD, and I would like to thank the many people who have supported me through this.

Thank you to my friends and family, in particular my wife, Miranda, for your love and support. Thank you for your patience and understanding, and for your encouragement to have another go at doing a PhD! Thank you, Samuel, my son, for your joy and your simple love. And lastly thank you to God, for your saving, self-giving love, your mercy, your grace, your guidance, and your peace.

Contents

Nomenclature

$\forall$	For All
$\in$	In
$\mathbb{C}$	Complex Numbers
$\mathbb{R}$	Real Numbers
ϕ or ψ	Global or Local Basis Function (context dependant)
$\tilde{\phi}$ or $\tilde{\psi}$	Local Basis Function (in one special instance, $\hat{\phi}$)
$\tilde{x}$	Local Coordinates (in one special instance, $\hat{x}$)
$\to$	Maps To
$f(u_3, \bullet; u_1, u_0)$	A function f with a slot in which to place an argument: it is not necessarily linear in this argument.
$f(u_3, u_2; u_1, \bullet)$	A function f with a slot in which to place an argument: it is linear in this argument.
$f(u_3, u_2; u_1, u_0)$	A function f which is linear in u_1 and u_0 but not necessarily linear in u_3 and u_2.
x	Global or Local Coordinates (context dependant)
AD	Automatic Differentiation or Algorithmic Differentiation
DSL	Domain Specific Language
FEM	Finite Element Method
FIAT	FInite element Automatic Tabulator
Field	Scalar, vector or tensor valued fields over some domain (unless explicitly stated otherwise).
FInAT	FInAT Is not A Tabulator (a finite element library)
HPC	High Performance Computing
MPI	Message Parsing Interface
P0DG	Polynomial degree 0 Discontinuous Galerkin function space

PDE	Partial Differential Equation
TSFC	Two Stage Form Compiler
UFL	Unified Form Language

Chapter 1

Introduction

1.1 Introduction

Mathematical models are a foundational part of how mathematicians, scientists and engineers describe and study the behaviour of systems. These include everything from physical systems, such as the weather and the propagation of electromagnetic waves, abstract systems such as the behaviour of financial markets, and mathematical systems such as the behaviour of particular differential equations. This work is concerned with mathematical models which can be solved with numerical methods, in particular *Finite Element Methods* (FEMs).

Finite element methods, often referred to as simply 'the finite element method', are relevant when one has a domain of interest and some scalar, vector, or tensor valued fields[1] on that domain which are related by a system of equations. Typical examples are physical systems: the behaviour of electromagnetic waves, continua such as fluids and solids and atomic particles. In particular, most of these systems can be described by differential equations, typically Partial Differential Equations (PDEs). Electromagnetic wave behaviour can be modelled by the Maxwell equations, continua by the equations of continuum mechanics (for fluids those might be the Navier-Stokes equations and their various simplifications, for solids the equations of elasticity and plasticity) and atomic particles by the various quantum mechanics equations such as the Schrödinger equation.

Such systems can also be solved with other numerical methods such as the Finite Difference Method (FDM), the Finite Volume Method (FVM), and spectral methods. These all have their own advantages and disadvantages, but we will not discuss them further here other than to mention that finite difference methods produce solutions that are defined at grid points, which are usually regularly spaced. A reasonable understanding of the finite element method is assumed throughout this text. For a good introduction, see Ham and Cotters' taught course [1] which is freely available online.

1.1.1 Abstraction

The word *abstraction* can be considered as both a process and a concept. As a process it is "separating a quality common to a number of objects/situations from other qualities" [2, p. 13]. For example, consider two situations: (a) joining the numbers 2 and 8 together to get 10 and (b) joining 6 and 3 together to get 9. If we apply abstraction to these we see that there is a common action of 'joining'. As a concept 'abstraction' is the name we give to our generalisation or *abstract concept*: in this case 'addition'. Abstractions are a foundation of mathematics: it is through the process of turning particular cases, such as the sizes of triangular areas of grass, that we reach generalisations, such as trigonometry.

There is also the related computer science concept of *levels of abstraction* where one looks at a particular operation and considers how it is performed with different levels of specificity. So returning to the addition of 2 and 8: at the highest level of abstraction we have the abstract

[1]here just referred to a *fields* unless stated otherwise

mathematics $2 + 8$ or $8 + 2$. At a lower level of abstraction we consider how those numbers are added, we might assign type information such as '2 and 8 are integers' and '+ is an integer addition operation'. Lower again we may introduce explicit instructions such as 'retrieve 64 bits stored at hexadecimal address A (the number 8) and place it into process register x, next, retrieve 64 bits stored at hexadecimal address B (the number 2) and place it into process register y, now perform an addition operation using process registers x and y and place the result (the number 10) in hexadecimal address C'. Each of these has a different level of specificity and, typically, might be described in a different computer language. The process of moving between such languages is the job of a compiler or an interpreter.

Levels of abstraction also apply to extracting abstract concepts from particular cases. In our two examples, one could come up with the high-level concept of 'addition' or the slightly lower level concept of 'addition of integers'. Which of these is appropriate depends on context: If one wants to know what operation is being performed symbolically, one can identify the + operator. If one is attempting to implement addition in a computer program, it is useful to know that there are different kinds of number (integers, floating point numbers, complex numbers) and to recognise that the rules one uses to add them are different.

Returning to finite element models of systems: these involve data. The data may be external to the system but necessary to have the model produce useful outputs, such as experimental measurements. The data may be some fields or samples from other simulations to integrate with the model at hand. Or the data may be fields within the system that have some relationship with another part of the system we are modelling. This work is concerned with generalising two key aspects of these specific examples to an appropriately high level of mathematical abstraction:

1. How to represent these data in the context of the finite element method.

2. How to describe interactions between such data and the approximations of fields used in finite element methods.

The first of these is already partly solved: field data are approximated as finite element functions, as is described in Sect. 1.2. Once we have finite element functions, the second is also solved in the form of mathematical interpolation and projection operations between these functions as is described in chapter 3. These operations are differentiable, allowing key insights to be gained into the interactions of data with models as will be described in chapters 5 and 6.

The problem is that the existing abstraction only considers finite element functions as approximations of fields, such as the solution to some PDE, typically generated within the context of a particular finite element method code base. This is unnecessarily restrictive; this work will demonstrate, in chapter 4, that we can also represent arbitrary point data as finite element functions. This will allows one to integrate such data with the pre-existing mathematical operators. This work will go on to demonstrate, in chapters 6, 7 and 8, how these can be used to directly interact with data which are defined outside the particular model we are considering.

Having identified the abstractions it is important to represent them at an appropriately high level. Domain specific languages, the Unified Form Language (UFL) and the Firedrake project, introduced in Sect. 1.3, are all concerned with doing exactly this, and it is within UFL and the Firedrake project that these abstractions will be implemented.

1.2 Fields as finite element functions

This section adapts and reuses, without further attribution, text and diagrams from Nixon-Hill et al. [3] which were created by the author.

What are here referred to as finite element methods are any method (typically *Galerkin methods*) which use finite element functions, which are described here.

Typically these are used to solve linear or nonlinear variational problems. A linear variational problem is of the form 'find u in U such that

$$a(;u,v) = L(;v)\ \forall\, v \in V.\text{'} \tag{1.2.1}$$

Note that both arguments are after a semicolon; we use this to denote linearity in those arguments.[2] The argument which we solve for all V, here v, is called a *test function*[3].

A nonlinear variational problem is 'find u in U such that

$$F(u;v) = 0\ \forall\, v \in V.\text{'} \tag{1.2.2}$$

Note that u, the term for which we seek a solution, is now before the semicolon, i.e. F is not-necessarily linear in u. F is always linear in the test function v. Concrete examples of these are considered in Sect. 1.3. The process of solving these problems themselves is not our object of study and so will not be discussed in detail.

In finite element methods the domain of interest is approximated by a set of discrete cells known as a mesh Ω. The field solution of a PDE u is then approximated as the sum of a discrete number of functions on the mesh. Each function is conveniently defined to be a *basis* or *shape* function ϕ, multiplied by some weight coefficient w. For N weight coefficients and basis functions our approximate solution

$$u(x) = \sum_{i=0}^{N-1} w_i \phi_i(x) \tag{1.2.3}$$

is called a *finite element function* or, within the context of finite element methods, simply a *function.*[4]. Given a set of basis functions $\{\phi_i(x)\}$ on a particular mesh Ω, the weights $\{w_i\}$ are allowed to vary to form a $u(x)$: these weights are referred to as Degrees of Freedom (DoFs).

[2]The term 'linear' is properly defined in Sect. 1.4.1.

[3]This is standard nomenclature in the finite element method.

[4]The terms *finite element field* or simply *field* are also used in the literature

The set of all possible weight coefficients applied to the basis functions on our mesh is called a *finite element Function Space* $\mathrm{FS}(\Omega)$. Within the context of finite element methods, this often just called a *function space* and is often referred to with a single capital letter such as U or V. Our finite element function is therefore a member of our finite element function space

$$u \in \mathrm{FS}(\Omega). \tag{1.2.4}$$

Finite element function spaces are grouped by their definitions on particular mesh cell shapes with precisely defined basis functions in each case. A popular choice are piecewise polynomials, such as the second order continuous Lagrange polynomials

$$u \in \mathrm{P2CG}(\Omega) \tag{1.2.5}$$

where P2CG stands for Polynomial degree 2 Continuous Galerkin and 'continuous' refers to continuity of the function between mesh cells. For a simple line domain $[0, 2]$ discretised into 2 cells $[0, 1]$ and $[1, 2]$ we have a total of $N = 5$ basis functions and weight coefficients (DoFs). The basis functions are shown in Fig. 1.1. Our finite element function u is given by Eq. 1.2.3: $u(x) = x^2$ has weights $w_0 = 0$, $w_1 = 0.25$, $w_2 = 1$, $w_3 = 2.25$ and $w_4 = 4$.[5]

The consistency of the functions across each cell is clear in this example. We have 3 weight coefficients and 3 basis functions which are nonzero in each cell in a repeating pattern. This is useful for performing calculations with finite element functions on meshed domains with many cells. Calculations can be done cell-by-cell as long as the weights of nonzero basis functions which are found in multiple cells (here only ϕ_2) are shared appropriately.

There are two key points here. Firstly, all finite element functions are members of finite element function spaces defined on a mesh. Secondly, our solution field is approximated with a finite element function (Eq. 1.2.3). As long as this function is continuous we know its values unambiguously. We can therefore evaluate the solution at any location so long as that location is on the mesh, a point we will return to in chapter 4, Sect. 4.3. In chapter 4, we will also return to finite element function spaces with discontinuities (usually found when using *discontinuous Galerkin* methods) in the context of point evaluations.

1.3 Firedrake, UFL and Domain Specific Languages

Firedrake [4, 5] is an automated system for solving PDEs, expressed as variational problems, using the finite element method. Firedrake is primarily written in Python, but produces highly optimised, parallelised C code which is compiled, cached and run whenever a Python script is executed. Firedrake achieves automation by representing the equations to solve at a high level of abstraction via the Unified Form Language (UFL) [6]. UFL is a Domain Specific Language

[5] All polynomials of 2nd order below are members of this finite element function space.

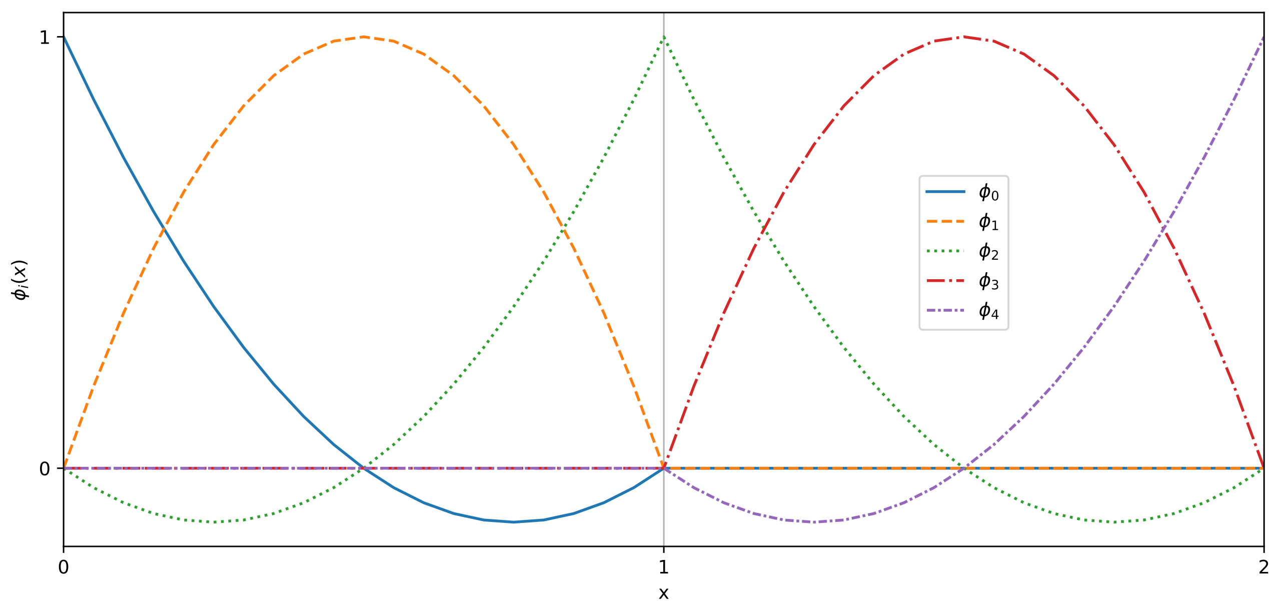


Figure 1.1: Second order Lagrange basis polynomials on an interval $[0, 2]$ meshed into two cells, $[0, 1]$ and $[1, 2]$. Only ϕ_2 is nonzero in both cells.

(DSL) for expressing symbolic mathematics in the context of finite element methods.

1.3.1 UFL and Domain Specific Languages

A DSL is a computer language which is specialised for some particular purpose; these exist in contrast to General Purpose Languages (GPLs) such as C, Python and FORTRAN. GPLs are designed, as the name implies, to be applicable in a range of different contexts. These operate with different levels of abstraction: C and FORTRAN, for example are much lower level languages than Python. The more levels of abstraction one needs to descend, the more one relies on the compiler (or, in the case of Python the interpreter) to produce efficient code. This has advantages and disadvantages: if a compiler is well written it may be better able to perform optimisations than a typical human would attempt, but tends to also require that code be written in particular styles (Python loops are famously slow compared to using other in-built language tools). Lower level languages give the programmer more fine grained control but often at the expense of understandability, complexity and verbosity.

Since a DSL is intended for a particular purpose, it has a more restricted set of operations one can perform but allows those operations to be expressed at a high level of abstraction making them easier to use [7]. In principal, if one presents fewer ways to express things, well written compilers should be able to produce high performance code.[6] Some examples of DSLs are TEX for typesetting, HTML for web pages, SQL for database queries and Make for software building [7].

[6]See for example the Two-Stage Form Compiler (TSFC) [8] which compiles UFL, expressed in Python, into high performance C code. This makes use of multiple DSLs which steadily decrease the level of abstraction, as detailed in chapter 3.

UFL is a DSL embedded within the Python GPL, and can make use of many of Python's features, but is its own, self contained language. UFL allows the symbolic mathematics of variational problems to be expressed at a very high level of abstraction. As an example, consider Poisson's equation

$$-\nabla^2 u = f. \tag{1.3.1}$$

This is the *strong formulation* of Poisson's equation. To solve this for u on some domain Ω we can express this in the *weak form* as the solution to the linear variational problem 'find u in V such that

$$\int_\Omega \nabla u \cdot \nabla v - \int_{d\Omega} \nabla u \cdot \boldsymbol{n} v \; ds = \int_\Omega f v \; dx \quad \forall\, v \in V.\text{'} \tag{1.3.2}$$

The UFL expression of this is shown in Listing 1.

```
 1  from ufl import *
 2  omega = Mesh(triangle)
 3  V = FunctionSpace(omega, FiniteElement("CG", triangle, 2))
 4  u = TrialFunction(V)
 5  v = TestFunction(V)
 6  f = Coefficient(V)
 7  n = FacetNormal(omega)
 8  a = inner(grad(u), grad(v)) * dx - inner(grad(u), n) * v * ds
 9  L = f * v * dx
10  # Now solve for u with a == L for all v in U
```

Listing 1: A UFL expression for the Poisson equation as a linear variational problem (Eq. 1.3.2).

On line 2 we symbolically define our domain Ω and its meshed equivalent[7] as a `Mesh` object, in this case with triangular cells. At this point we have specified no information about the structure or density of the mesh itself. On line 3 we define our function space V as a `FunctionSpace` object, in this case the space of second order continuous Lagrange polynomials we encountered in Sect. 1.2, though in this case on 2D triangular cells. On line 4 we define the solution u as a *trial function*, another standard term in finite element methods for the function that the linear variational problem is solved for, in the function space V. On line 5 we define the test function v, also in V.[8] On line 6 we define the forcing function f as a `Coefficient` object in V: this is the UFL name for a finite element function which has known basis function weight coefficients. On line 7 we create an outward facing normal vector `n` to the boundary of Ω as a `FacetNormal` object. On line 8 we define the left hand side of our problem, the 2-linear (bilinear) form $a(;u,v)$. [9] On line 9 we define the right hand side of our problem, the 1-linear

[7]The meshed equivalent is often referred to as Ω_h. In this work Ω is used to refer to domain or its mesh, depending on the context.

[8]A note on Galerkin methods: by specify both u and v to be in the same function space, this is a *Ritz-Galerkin* method. If they were in different spaces we would be using a *Petrov-Galerkin* method.

[9]The terms '2-linear form' and 'bilinear form' are properly defined in Sect. 1.4.4.

(linear) form $L(;v)$.

This is entirely symbolic, we have not specified an actual domain or mesh (Ω), other than to imply that it must be solved in 2D by specifying triangular cells. Similarly, we do not give values to our forcing function f. We therefore cannot use UFL for solving the problem; we need to implement some system which will attach data to these symbolic expressions and then solve them. Nevertheless, this is a flexible way of expressing the mathematics of our problem which is the intended level of abstraction. UFL has operators such as `grad` and `inner` which can be symbolically differentiated using a `derivative` operator[10], as well as domain specific features such as meshes and function spaces.

1.3.2 Firedrake

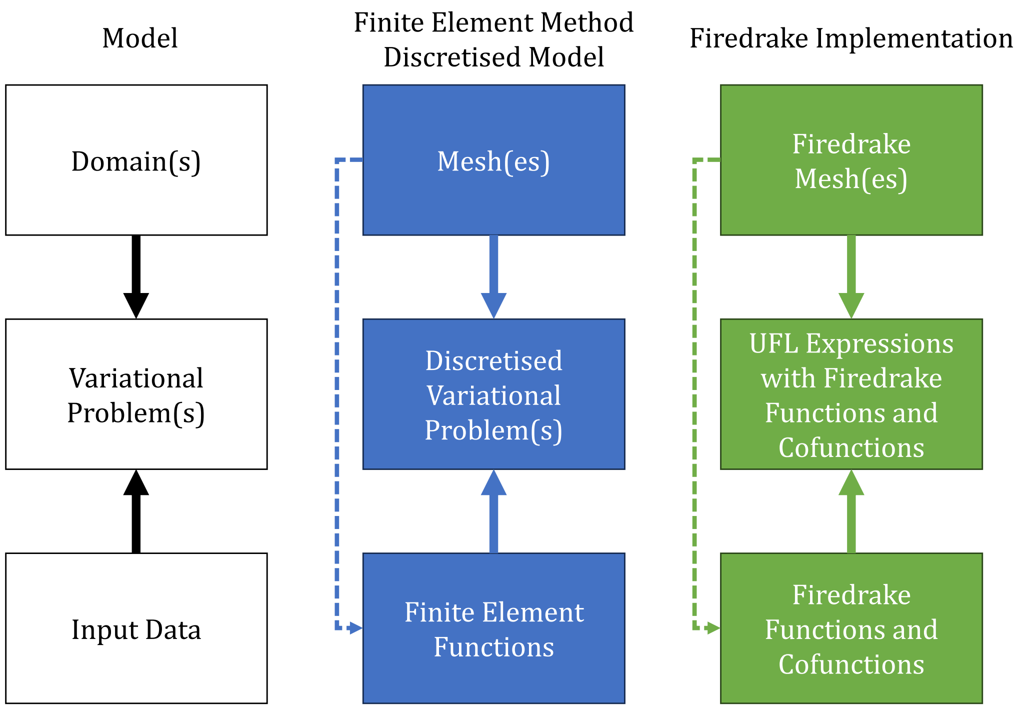

Model
Finite Element Method Discretised Model
Firedrake Implementation
Domain(s)
Variational Problem(s)
Input Data
Mesh(es)
Discretised Variational Problem(s)
Finite Element Functions
Firedrake Mesh(es)
UFL Expressions with Firedrake Functions and Cofunctions
Firedrake Functions and Cofunctions

Figure 1.2: The relationship between models on domains, finite element method discretisations, and their Firedrake implementation. Variational problems that one can solve with finite element methods exist on some domain, with some data supplied to inform them. The finite element method discretisation has these data as finite element functions on some mesh, which introduces an extra dependency. Note the similarity between Firedrake's implemented abstractions and the finite element method discretisation. Firedrake `Cofunctions` are introduced in chapter 2.

Firedrake adds concrete data to symbolic UFL expressions by introducing Python subclasses of the symbolic UFL objects. These have the same behaviour as their symbolic equivalents so can be used in place of them. Firedrake therefore maintains the high level of abstraction of UFL but allows variational problems to be solved and systems to be modelled.

[10]this performs the *Gateaux derivative*, introduced in chapter 5

```
1  from firedrake import *
2  omega = UnitSquareMesh(20, 20)
3  V = FunctionSpace(omega, family="CG", degree=2)
4  u = TrialFunction(V)
5  v = TestFunction(V)
6  f = RandomGenerator(PCG64(seed=0)).beta(V, 1.0, 2.0)
7  n = FacetNormal(omega)
8  a = inner(grad(u), grad(v)) * dx - inner(grad(u), n) * v * ds
9  L = f * v * dx
10 
11 bc = DirichletBC(V, 0, "on_boundary")
12 u_sol = Function(V)  # solution will be stored here
13 solve(a == L, u_sol, bc)
14 
15 # Equivalent nonlinear variational problem
16 u = Function(V)
17 F = (inner(grad(u), grad(v)) - f * v) * dx - inner(grad(u), n) * v * ds
18 solve(F == 0, u, bc)  # solution will now be in u
19 u_sol = u
```

Listing 2: Solving Poisson's equation (Eq. 1.3.2) with Firedrake using a randomly generated forcing function on a unit square mesh with $20 \times 20 \times 2 = 800$ triangular cells. The forcing function f and solution u are shown in Figure 1.3.

As an example, see Listing 2, which solves Poisson's equation with Firedrake. We now have a concrete Ω, a mesh of the unit square with vertices (x, y) at $(0, 0)$, $(0, 1)$, $(1, 0)$ and $(1, 1)$ made up $20 \times 20 \times 2$ triangles. The finite element function space we solve on is, as before, the space of second order continuous Lagrange polynomials. The forcing function is a randomly generated function $f(x, y)$ in V with values between 1.0 and 2.0.[11]

We can now specify a boundary condition, which we set as $u = 0$ on the boundary of Ω. [12] This gives us everything we need to solve the problem, which we do on line 13. The solution is deposited into a `Function` object: this is the data carrying subclass of the UFL `Coefficient` object in Firedrake (`f` is also a `Function`).[13] Plots of the resultant f and u are shown in Fig. 1.3. A summary of the relationship of Firedrake to the finite element method and variational problems on domains is shown in Fig. 1.2.

Firedrake can also be used to solve nonlinear variational problems of the form 'find u in U such that

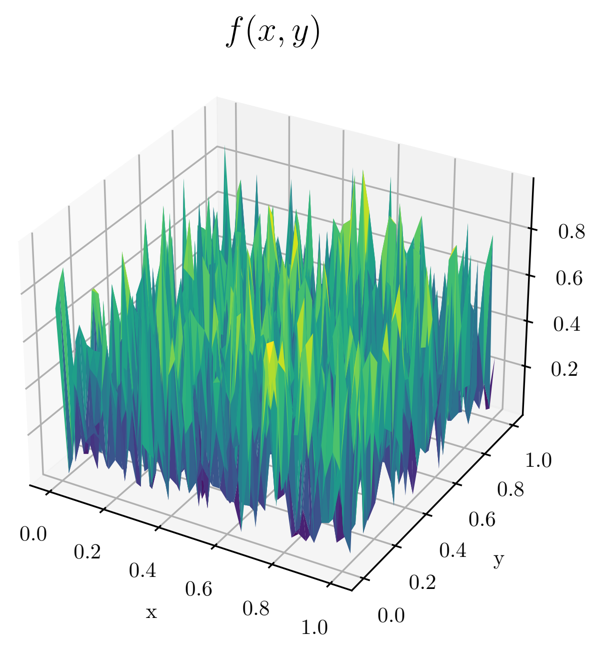
$$F(u; v) = 0 \; \forall \, v \in V.\text{'} \tag{1.3.3}$$

This is done iteratively using a Newton-type method which has one repeatedly solving a linear

[11]Note that the forcing function shown will not be mesh independent - i.e. a true solution will not be approached as the mesh is refined. This is fine for this illustrative example, but it still worth noting.

[12]This is a Dirichlet boundary condition, also known as a 'strong' boundary condition. Since we specify a value on the boundary the $\int_{d\Omega} \nabla u \cdot \boldsymbol{n} \, ds$ term goes to zero, but it is kept here for clarity.

[13]Data carrying Firedrake equivalents to UFL types are discussed further in chapter 2.



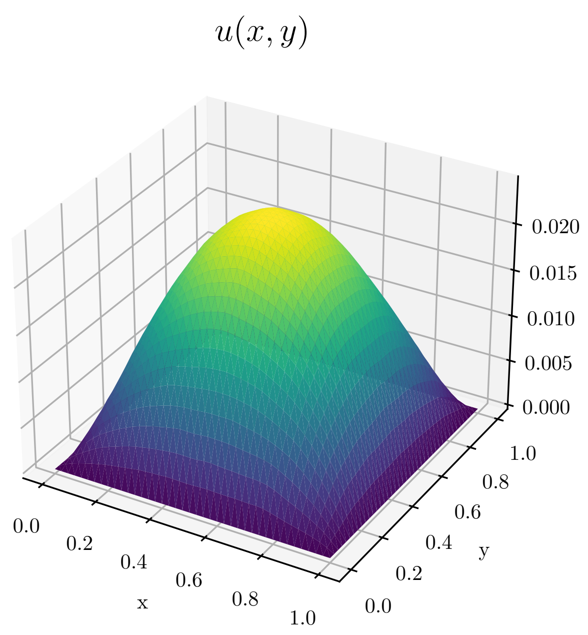


Figure 1.3: The forcing function f (left) and solution u (right) to Poisson's equation (Eq. 1.3.2), generated with the code in Listing 2.

system based on F and its first derivative[14], until F is sufficiently close to 0. For more, see chapter 9. Whilst Poisson's equation is a linear PDE, we can express it in the form of a nonlinear variational problem 'find $u \in V$ such that

$$\int_{\Omega} \nabla u \cdot \nabla v - fv \; dx - \int_{d\Omega} v \nabla u \cdot \boldsymbol{n} \; ds = 0 \quad \forall v \in V \tag{1.3.4}$$

which will converge in a single Newton iteration. This is done on lines 15 to 19 of Listing 2.

Firedrake maintains a close relationship with the Portable, Extensible Toolkit for Scientific Computation (PETSc) [9, 10]. PETSc solvers are used to solve both linear and nonlinear PDEs[15], whilst mesh topology information is stored on PETSc data structures. This will be discussed more in chapter 4.

Firedrake programs can be run in parallel using the Message Parsing Interface (MPI) [11].[16] A script that runs in serial can also be run on thousands of MPI processes with no changes to the code.[17] Parallelism occurs at various stages: meshes and function spaces on them are 'parallel decomposed', function manipulations require parallel communications, and variational problem solves make use PETSc's MPI parallelism. Much of this is discussed further in chapter 7.

Relevant parts of Firedrake's execution model are discussed more in chapter 3 and sum-

[14]taken using a Gateaux derivative, introduced in chapter 5

[15]Firedrake's solver functions are more flexible than this simple example suggests. For example, PETSc solver options can be passed from Firedrake to PETSc. See the PETSc manual [10] and Firedrake project website https://www.firedrakeproject.org/ for more information

[16]For more on MPI Gropp's 'Using MPI' [12] is a lucid introduction.

[17]It is a requirement that any features added to Firedrake must work seamlessly in serial and parallel.

marised in Fig. 1.4. In brief, one finds oneself with UFL expressions, which contain Firedrake data carrying types, that need to be executed on a meshed domain. The UFL expressions are compiled into small executables, known as 'kernels' to execute on a given mesh cell: this is the responsibility of the Two Stage Form Compiler (TSFC) [8]. Within TSFC, symbolic UFL elements are converted into concrete implementations via the Finite Element Automatic Tabulator (FIAT) [13] and FInAT[18] [14] libraries. Execution of the kernels on every cell of the mesh is handled by the PyOP2 library [15] which produces fast C code for doing this which it compiles and runs. Where one is solving a linear variational problem[19], one now has a globally defined linear system $\boldsymbol{Ax} = \boldsymbol{b}$ to solve, which is the responsibility of PETSc. Time varying problems are solved by setting up an iterative time-stepping regime when writing the Firedrake code itself.

The process of going from UFL expression with data attached, such as an expression of a linear variational problem $a(;u,v) = L(;v)$, to a concrete set of numbers, such as system of equations to solve $\boldsymbol{Ax} = \boldsymbol{b}$, is known as *assembly* and will be discussed further in chapter 2.

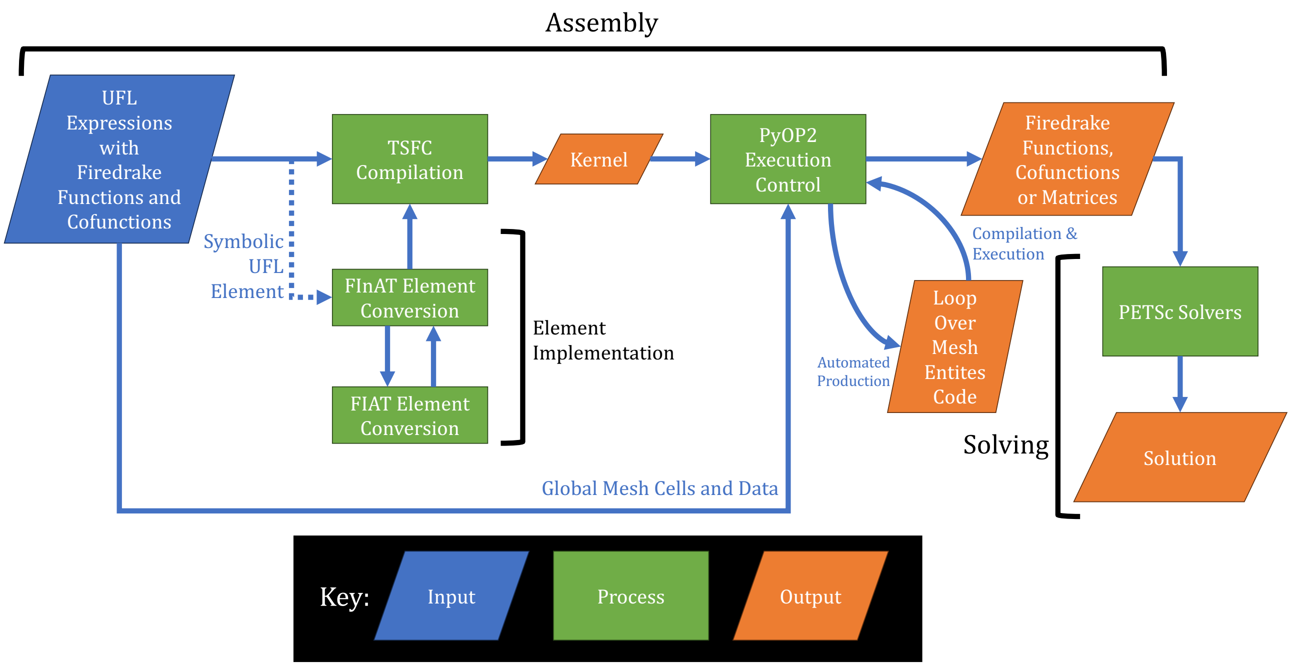


Figure 1.4: Firedrake's primary execution model, from UFL expressions with Firedrake data-carrying `Functions` and `Cofunctions` (see chapter 2) to assembled Firedrake objects which can be passed to linear solvers giving some solution.

[18]A recursive acronym 'FInAT Is not A Tabulator'

[19]by this stage, a nonlinear variational problem has been linearised by Firedrake in the Newton solver so it will look like a linear problem

1.3.3 Other Finite Element Method DSLs and Libraries

As well as Firedrake, the finite element libraries FEniCS [16, 17][20], it's recent rewrite FEniCSx, and DUNE-Fem [18] use UFL for the purpose of specifying variational problems to solve. As with Firedrake, legacy FEniCS and FEniCSx each take the approach of creating Python subclasses of the symbolic UFL objects to attach data and allow their manipulation, and implement a form compiler to generate kernels.

UFL is not the only domain specific language found in scientific computing for producing models. For example, the Devito DSL [19] provides a similar level of mathematical abstraction as UFL, but within the context of finite-difference methods.

Within the world of finite elements, the recently developed Finch DSL [20], embedded in the Julia programming language, is very similar to UFL and, within the language, contains additional specialised operations needed for finite volume methods. FREEFEM [21] is a C++ library which also contains a similar DSL. These projects both include a solver, so do not have as firm a separation of symbolics from data as found in UFL and the libraries, such as Firedrake, that use it.

Other finite element method libraries, which will be introduced as necessary in the text, do not advertise themselves as DSLs but do provide a high level of abstraction. In particular, NGSolve [22] has a Python API which bears a strong resemblance to UFL, though perhaps with more finite element method specific nomenclature.

Other open source libraries such as Elmer-FEM [23] and GOMA [24] operate at a higher level of abstraction than UFL, with finite element models of specific, predefined systems of PDEs performed. The Multiphysics Object-Oriented Simulation Environment (MOOSE) [25] allows users to define their own PDEs, but operates at a much lower level of abstraction than UFL: one must specify the kernels to use on each mesh cell from the weak form of the PDE.

There are many finite element method libraries which are application specific of which many are closed source. Examples include Abaqus FEA [26], COMSOL Multiphysics [27] and various software packages developed by ANSYS [28]. These find wide use in engineering but, due to their closed source and application specific nature, cannot be easily compared with the work in this thesis.

1.4 Hilbert Spaces, Dual Spaces, the Riesz Representation Theorem and Multilinear Forms

1.4.1 Hilbert Spaces

Hilbert spaces, named after 19th and 20th century mathematician David Hilbert, generalise Euclidean vectors ($u \in \mathbb{R}^n$ or $c \in \mathbb{C}^n$) and their associated spaces ($\mathbb{R}^n$ and $\mathbb{C}^n$) to include

[20]now known a 'legacy' FEniCS

infinite dimensions and vectors which are functions. When discussing Hilbert spaces we adopt similar notation to that found in Schwedes et al. [29]. The requirements of a Hilbert space H are that it has some inner product $\langle \bullet, \bullet \rangle_H$ (which induces a norm on its members) and that it is *complete*. The notation '$\bullet$' is used to indicate where operands are placed.

All finite element function spaces, such as piecewise polynomials, are Hilbert spaces, as are Euclidean vector spaces: the difference is which inner product is taken. For vectors u and v which are piecewise polynomial functions it is typically a Sobolev inner product, the simplest of which is the L^2 inner product over the meshed domain Ω

$$\langle u, v \rangle_{L^2} = \int_\Omega u(x)v(x)dx \tag{1.4.1}$$

whilst for n dimensional Euclidean vectors it is the l_2 inner product

$$\langle u, v \rangle_{l_2} = \sum_{i=0}^{n-1} u_i v_i. \tag{1.4.2}$$

A Hilbert space's inner product induces a *norm* (a real number which gives a notion of how 'big' a member of the space is) given, for some u in the Hilbert space, by

$$\|u\| = \sqrt{\langle u, u \rangle}. \tag{1.4.3}$$

The requirement that a Hilbert H space is 'complete' means that any Cauchy sequence of points within the space, i.e. a sequence where the points get arbitrarily close to one another (i.e. close to some point) as the sequence continues, is, as the sequence tends to having infinite members, within the space. This, crucially, means that one can take limits in the space, for example when defining calculus operations. Hilbert spaces are a subset of both *Banach spaces*, which are complete and have a norm (but not necessarily an inner product) and *inner product spaces* which have an inner product but are not necessarily complete.

1.4.2 Dual Spaces

The *topological dual space* or *continuous dual space*, here simply referred to as the *dual space*, H^* to a so-called *primal* vector space H is the space of linear continuous functions (also known as linear continuous *forms*) that map from H to the scalar members K that make up each vector:

$$H^* = H \to K. \tag{1.4.4}$$

Such functions are known as *functionals*. If H is a real vector space (such as $\mathbb{R}^n$) then the scalar members are real numbers and $K = \mathbb{R}$; if it is a complex vector space (such as $\mathbb{C}^n$) then

$K = \mathbb{C}$. We will limit ourselves to the real case, i.e.

$$H^* = H \to \mathbb{R}. \tag{1.4.5}$$

The terms 'linear' and 'continuous' require explanation. Consider some specific functional $u^* \in H^*$. We say that it is *linear* if it can be expressed as

$$u^*(;\alpha v) = \alpha u^*(;v)\ \forall\, v \in H, \alpha \in \mathbb{R}. \tag{1.4.6}$$

This linearity in αv is denoted by a semicolon before the argument. Furthermore we say that u^* is *continuous* if, for our purposes, there are no discontinuities in the value of $u^*(v)$, where $v \in H$, as we vary v (i.e. $u^* \in C^0$ and $H^* \subset C^0$)[21].[22] Functional continuity is only relevant for infinite dimensional functionals; since this work focusses on the finite dimensional case, this will not be remarked upon again.

1.4.3 The Riesz Representation Theorem

For Hilbert spaces, the *Riesz representation theorem* links given vectors in primal Hilbert spaces to *unique* continuous linear functionals in the dual space. Such is the link that, for each vector, the corresponding member of the dual space is generally referred to as a *covector*.

Consider a Hilbert space with inner product $\langle \bullet, \bullet \rangle_H$. The theorem states that, for the dual space of linear continuous functionals H^* with a given member $u^* \in H^*$ there is a unique $u \in H$ such that

$$u^*(;v) = \langle u, v \rangle_H \in \mathbb{R}\ \forall\, v \in H.[23] \tag{1.4.7}$$

The inner product in the primal space gives an inner product to use in the dual space and thus an induced norm. These are equivalent:

$$\|u\|_H = \|u^*\|_{H^*}. \tag{1.4.8}$$

Put another way, (a) for every covector there is a vector where (b) the covector functional is given by the inner product of the vector in the primal space and (c) each pair have identical magnitude. The theorem also states that the inverse is true, i.e. that for every vector there is a covector. We care about the Riesz representation theorem because it gives us a definition

[21]There are various formal definitions of continuity, see for example equation 1.87 in Schwedes et al. [29].

[22]The dual space of all linear functionals is known as the 'algebraic dual space'; we are merely concerned with the subspace of continuous linear functionals, the 'topological' or 'continuous' dual space.

[23]Note that, in mathematics, the second argument in an inner product is *antilinear*, meaning, for a scalar α, $\langle u, \alpha v \rangle_H = \langle u, v \rangle_H \bar{\alpha}$ where $\bar{\alpha}$ is the complex conjugate of α. Since we are limiting ourselves to real-valued spaces antilinearity *is* linearity. In the complex case this would define the *anti-dual space* $\bar{H}^*$, indeed UFL, and hence Firedrake, treat all functionals (i.e. forms) when running in complex mode as antilinear. This means that a form employing a test function v, which in real mode we are linear in, is treated as being antilinear in complex mode. I.e. $\langle \bullet, v \rangle = \bar{v}^T \cdot \bullet$. Dealing with the complex case is beyond the scope of this work.

for vector *adjoints*: the covector associated with a vector is known as its *adjoint* whilst the operation which finds it is called *taking the adjoint*. More on the Riesz representation theorem using similar notation, including a proof, can be found in section 1.2.1 of Schwedes et al. [29].

The *Riesz map* takes each u^* from $H^* \to H$

$$\mathcal{R}_H : H^* \to H \tag{1.4.9}$$

for a given choice of inner product. The Riesz map calculated for a given covector and inner product

$$\mathcal{R}_H(u^*) \in H \text{ for given } u^* \in H^* \tag{1.4.10}$$

is known as the *Riesz representer*.

1.4.4 Using Multilinear Forms to Find the Riesz Representer

Multilinear forms are maps which are linear in all their arguments and map to a scalar K, which is usually either $\mathbb{R}$ or $\mathbb{C}$. [24] As before, we will limit ourselves to the real case, i.e. $K = \mathbb{R}$. For k linear arguments in function spaces $\{V_i\}_0^{k-1}$ this is

$$V_{k-1} \times ... \times V_0 \to \mathbb{R} \tag{1.4.11}$$

which we refer to as a k-linear form, often referred to as simply a k-form. Since it maps to $\mathbb{R}$, a multilinear form is an example of a *functional*. Here are some examples:

1. A 2-linear form (also known as a bilinear form or 2-form) contains two arguments, for example
$$\int_\Omega u(x)v(x)dx \in \mathbb{R}\ \forall\, u \in U, v \in V. \tag{1.4.12}$$

2. a 1-linear form (also known as a linear form or 1-form) contains one argument, for example
$$\int_\Omega u(x)dx \in \mathbb{R}\ \forall\, u \in U. \tag{1.4.13}$$

3. a 0-linear form (also known as a number[25] or a 0-form) contains no arguments, for example
$$\int_\Omega 10dx \in \mathbb{R}. \tag{1.4.14}$$

Confusingly there is another kind of "form" which crops up when talking about finite elements, a *differential form*. This is a generalised approach for the expression of integrands for

[24]The general name for K in mathematics is a "field". This ought not to be confused with "field" as used here which can be considered to be synonymous with "function".

[25]more properly a particular member of the "field" K, in this case $\mathbb{R}$

integrals over any number of dimensions: line, surface, volume and higher dimension manifolds. These are used in Finite Element Exterior Calculus (FEEC) which is a complicated area and not the topic of this thesis. Unless stated otherwise, the word “form” here exclusively refers to multilinear forms.

Riesz representers can be expressed by considering our inner product as a 2-linear (bilinear) form M operating on two spaces U_1 and U_0

$$M : U_1 \times U_0 \to \mathbb{R} \text{ i.e. } M(; u_1, u_0) \in \mathbb{R}. \tag{1.4.15}$$

Note we now have two arguments after the semicolon, we are linear in both of them. For the L^2 inner product this is

$$M^{L^2}(; u_1, u_0) = \int_\Omega u_1(x) u_0(x) dx. \tag{1.4.16}$$

The argument ordering is deliberately in reverse order to match the ordering found in UFL; this is discussed in more detail in Sect. 2.4.1. We can *Curry* this form by expressing it as a function of U_1 which linearly maps to another function, this time of U_0, which then maps to $\mathbb{R}$

$$M(; u_1, u_0) = \underbrace{M_1(; u_1)}_{M_0}(; u_0) = M_0(; u_0) = \langle u_1, u_0 \rangle_H \in \mathbb{R}. \tag{1.4.17}$$

In more detail

$$M_1 : U_1 \to (\underbrace{U_0 \to \mathbb{R}}_{U_0^*}) \tag{1.4.18}$$

only evaluates u_1, leaving u_0 as a free argument such that

$$M_1(; u_1)(; \bullet) = \langle u_1, \bullet \rangle_H \in U_0^* \quad \forall\, u_1 \in U_1. \tag{1.4.19}$$

The symbol ‘$\bullet$’ indicates an empty slot to place the operand u_0. Lastly,

$$M_0 : U_0 \to \mathbb{R} \tag{1.4.20}$$

already includes the evaluation of u_1 such that

$$M_0(; u_0) = \langle u_1, u_0 \rangle_H \in \mathbb{R} \quad \forall\, u_0 \in U_0. \tag{1.4.21}$$

If $U_0 = U_1 = U$ with members u then we have

$$M_1 : U \to U^* \text{ s.t. } M_1(; u)(; \bullet) = \langle u, \bullet \rangle_H \in U^* \tag{1.4.22}$$

i.e. M_1 is the *inverse* of our Riesz map. We can then express the inverse of the Riesz representer as

$$M_1(; u)(; \bullet) = \langle u, \bullet \rangle_H = u^*(; \bullet). \tag{1.4.23}$$

The Riesz representer itself is the solution to this equation where we solve for u. Since $M_1(;u)(;\bullet) = M_0(;\bullet) = u^*(;\bullet)$, M_0 is the covector u^* to u. Note that $M_1(;u) = M_0$ is a 1-linear form: covectors are 1-linear forms.[26]

For M^{L^2} on some domain Ω we have

$$M_1^{L^2}(;u_1)(;\bullet) = \int_\Omega u_1(x) \cdot \bullet \, dx \in U_0^* \tag{1.4.24}$$

$$M_0^{L^2}(;u_0) = \int_\Omega u_1(x) u_0(x) dx \in \mathbb{R}. \tag{1.4.25}$$

Taking $U_0 = U_1 = U$ with members u shows us that $M_1^{L^2}$ is our inverse Riesz map for Hilbert spaces on a domain Ω, where the inner product is L^2. The inverse of the Riesz representer is given by

$$M_1^{L^2}(;u)(;\bullet) = M^{L^2}(;u,\bullet) = \langle u, \bullet \rangle_{L^2} = \int_\Omega u(x) \cdot \bullet \, dx = u^*(;\bullet). \tag{1.4.26}$$

In general it is much easier to express how one gets the covector from the vector than the other way around! Fortunately this is usually the operation that we want. The process of Currying and un-Currying and its relationship to the development of dual spaces in UFL (the topic of chapter 2) is discussed further in Ham [30].

The Hilbert space of Euclidean vectors $\mathbb{R}^n$ is a simple example. A typically used inner product here is l_2 so for $u^* \in (\mathbb{R}^n)^*$ we have

$$u^*(;v) = \langle u, v \rangle_{l_2} = \sum_{i=0}^{n-1} u_i v_i \in \mathbb{R} \; \forall \, v \in \mathbb{R}^n. \tag{1.4.27}$$

This sum is the same for a row-vector/vector product

$$u^*(;v) = \underbrace{u^*}_{\in 1\times\mathbb{R}^n} \underbrace{v}_{\in\mathbb{R}^n} = \begin{pmatrix} u_0 & \dots & u_{n-1} \end{pmatrix} \begin{pmatrix} v_0 \\ \vdots \\ v_{n-1} \end{pmatrix} = \sum_{i=0}^{n-1} u_i v_i = \langle u, v \rangle_{l_2} \tag{1.4.28}$$

so we can say that

$$(\mathbb{R}^n)^* = 1 \times \mathbb{R}^n. \tag{1.4.29}$$

The requirement that the norms be the same

$$\|u^*\| = \sqrt{u^*(;u)} = \sqrt{\underbrace{u^*}_{\in 1\times\mathbb{R}^n} \underbrace{u}_{\in\mathbb{R}^n}} = \sqrt{\langle \underbrace{u}_{\in\mathbb{R}^n}, \underbrace{u}_{\in\mathbb{R}^n} \rangle_{l_2}} = \|u\| \tag{1.4.30}$$

tells us that the unique covector $u^* \in (\mathbb{R}^n)^*$ is in fact the transpose of u

$$u^* = u^T \tag{1.4.31}$$

[26]This is obvious when we recall that a covector in U^* is a linear map of $U \to \mathbb{R}$.

and so the Riesz map is 'take the transpose'.

The 2-linear (bilinear) form for the inner product here is M^{l_2}

$$M_1^{l_2}(;u_1)(;\bullet) = \langle u_1, \bullet \rangle_{l_2} = \sum_{i=0}^{n-1} u_{1i}[\bullet]_i \in U_0^* \tag{1.4.32}$$

$$M_0^{l_2}(;u_0) = \sum_{i=0}^{n-1} u_{1i} u_{0i} \in \mathbb{R}. \tag{1.4.33}$$

Taking $U_0 = U_1 = U$ with members u gives $M_1^{l_2}$ as our inverse Riesz map for $\mathbb{R}^n$ whilst our inverse Riesz representer is

$$M_1^{l_2}(;u)(;\bullet) = M^{l_2}(;u,\bullet) = \langle u, \bullet \rangle_{l_2} = \sum_{i=0}^{n-1} u_i[\bullet]_i = u^*(;\bullet). \tag{1.4.34}$$

The sum here once again tells us that covectors can be represented as row vectors.

Since we have finite computational resources, we cannot perform calculations on infinite dimensional vector spaces. The inner products we perform on finite element function spaces, such as the L^2 inner product on piecewise polynomials can always be expressed as a row-vector/matrix/vector product or equivalent tensor contraction. This is because we represent our functions as coefficients of a finite number of basis functions: for a finite dimensional function space U on some meshed domain Ω where we take the L^2 inner product we have

$$\langle u, v \rangle_{L^2_U} = \int_\Omega u(x) \cdot v(x) dx = \sum_{i,j}^{\dim(U)} \int_\Omega u_i \phi_i(x) \cdot v_j \phi_j(x) dx = \sum_{i,j}^{\dim(U)} u_{1i} M^U_{ij} v_j, \tag{1.4.35}$$

$$u, v \in U, \tag{1.4.36}$$

$$M^U_{ij} = \int_\Omega \phi_i(x) \cdot \phi_j(x) dx. \tag{1.4.37}$$

This form, where we have a matrix, here M^U, between the coefficients of the basis functions is common to all inner products. M^U is necessarily square since u and v are taken from the same discretised space. The matrix defines the exact inner product we do but its elements depend on the specific discretisation we use, which in turn depends on how our domain has been meshed and which function spaces we choose. For given functions in a particular function space on a particular mesh, however, the matrix is always the same. We deliberately chose to call the forms we used earlier M for this very reason. For the L^2 inner product the matrix is used so often that it is known as the *mass matrix*. For other inner products this matrix generally goes unnamed. For an H^1 inner product it could be called a discretisation of the Helmholtz operator or a 'Helmholtz matrix' (the mass matrix minus the discretisation of the Laplacian operator)[27]

[27]There are multiple equivalent, though not equal, ways of expressing an H^1 inner product: one can, for example, multiply the discretisation of the Laplacian operator by a scaling factor and still have an H^1 inner

whilst for the l_2 inner product it is an identity matrix.

A point that will become important in our discussion of automatic differentiation (Chapter 5), is that if we change any of the mesh, the function space, or the choice of inner product, we get a different covector for each vector in the function space which depends on the particular inner product matrix. An important consequence of this for mesh refinement is discussed in Sect. 5.3. For example the inverse Riesz representer that goes with equation 1.4.35 is

$$M_1^{L^2_U}(;u)(;\bullet) = \langle u, \bullet \rangle_{L^2_U} = \sum_{ij}^{\dim(U)} u_{1i} M^U_{ij} \cdot [\bullet]_j = u^*(;\bullet) \tag{1.4.38}$$

i.e. u^* is the transpose of the coefficients of the basis functions of the primal function u multiplied by the particular mass matrix M^U

$$u^*_{1j} = \sum_{i}^{\dim(U)} u_{1i} M^U_{ij}. \tag{1.4.39}$$

This is the same for any Hilbert space where its members are represented by finite members of orthogonal basis functions: one merely needs to update the inner product matrix to reflect the inner product used. Note that since M^U_{ij} is square, for every u_{1i} we have a u^*_{1j}, i.e. the finite dimensional space of covectors U^* has the same number of dimensions as the primal space U

$$\dim(U) = \dim(U^*). \tag{1.4.40}$$

product.

Chapter 2

Dual Spaces in UFL

This chapter as a whole draws on ideas from [30].

2.1 Motivation and Existing Work

All software libraries present a layer of abstraction on top of some underlying set of routines in order to achieve a particular task, via an Application Programming Interface (API). A Domain Specific Language (DSL) takes the idea of an API and allows the user to be more expressive, using the software library in more ways than a developer could program within an API. To achieve this, DSLs need to implement a set of types and operations that can be applied to those types: for example in UFL there are `Coefficient` types which can be added together to produce a UFL `Expr` object (specifically an instance of `Sum` which is a subclass of `Expr`). This result can then be further manipulated within UFL. As long as operations on objects within the language yield objects which are also within the language, various rules can then be implemented in an attempt to avoid the valid expression of things which are uncomputable or mathematically nonsensical.

UFL is a powerful language for specifying k-linear forms which are integrals. However, prior to the start of this PhD it had no concept of dual spaces or the covector members of them despite them often appearing in the language, as is detailed in this chapter. Adding these concepts makes the language safer, since one can type check for a primal function versus a dual space covector, and, as will be detailed in chapter 9, allows new kinds of k-linear form to be represented.

Adding covectors also gives types to use with *adjoint* operators: Consider a linear operator on Hilbert spaces $A : U \to V$, with corresponding dual spaces U^* and V^*. The adjoint to A is the linear operator

$$A^* : V^* \to U^* \tag{2.1.1}$$

which satisfies

$$v^*(; A(; u)) = A^*(; v^*)(; u) \; \forall \, v^* \in V^*, u \in U. \tag{2.1.2}$$

In words: for specific u and v^*, $A^*(v^*) \in U^*$ is the covector that operates on u to give the same answer as v^* operating on $A(; u) \in V$. Adjoint operators are fundamental to operations between Hilbert spaces and therefore appear regularly in finite element methods. Their construction is considered in chapter 5, Sect. 5.4.1.

As stated before, UFL is used by Firedrake, legacy FEniCS [16, 17] its recent rewrite 'FEniCSx' and DUNE-Fem [18] for the purpose of specifying weak formulations of PDEs, so making this change has a significant potential impact.

Some finite element libraries that maintain a significant enough level of abstraction to recognise that finite element functions are vectors in Hilbert spaces already contain this information. MFEM [31, 32] explicitly has types which maintain a clear distinction, as detailed on their website[1], whilst FREEFEM [21] discusses ‘dual vectors’ but does not have an explicit type for them. Other libraries which are considered in this work, such as deal.II [33], NGSolve [22] and Finch [20] do not usually explicitly refer to dual spaces and covectors in their documentation, other than when referring to dual evaluation[2] (the topic of chapter 3). This is not surprising: for the distinction to be of significant consequence, the method of representing k-linear forms needs to be sufficiently abstracted and composable for this to matter. For finite element methods, only UFL operates at a high enough level of abstraction.

2.2 Summary of Contributions

I have been closely involved in the effort to add dual spaces to UFL, aiding both design and implementation, throughout my PhD. This work was completed by Nacime Bouziani[3] who went on the integrate the UFL changes with Firedrake. I am listed as an author on a, currently work-in-progress, paper detailing this work.

Whilst this is not my main body of work, much of the description here is needed to understand later chapters, in particular chapter 9.

2.3 Representing Covectors as Cofunctions

Recall that finite element functions are weighted sums of basis functions (Sect. 1.2)

$$u(x) = \sum_{i}^{\dim(U)} u_i \phi_i(x) \in U. \tag{2.3.1}$$

Ciarlet’s finite element formulation [34]) has us build the basis functions from members of the dual space $\phi_i^* \in U^*$ which are defined such that

$$\phi_i^*(;\phi_j) = \delta_{ij} \tag{2.3.2}$$

where δ_{ij} is the Kronecker delta. This is known as a *biorthogonality relationship.*

Thanks to this formulation, we can also represent the functional members of U^* as weighted sums, this time of the dual basis functionals

$$u^* = \sum_{j}^{\dim(U)} u_j^* \phi_j^* \in U^*. \tag{2.3.3}$$

The coefficients u_j^* of the dual basis functionals can be calculated from u_i by considering the

[1]https://mfem.org/pri-dual-vec/
[2]Finch does not mention dual evaluation
[3]Department of Mathematics, Imperial College London

j^{th} entry of the row vector in Eq. 1.4.39

$$u_j^* = \sum_i^{\dim(U)} u_i M_{ij}^U \tag{2.3.4}$$

where M_{ij}^U is the inner product matrix we have chosen for U to get U^*[4]. We usually call these covector functionals *cofunctions*.

The action of some $u^* \in U^*$ on some $v \in U$ yields the usual inner product thanks to the biorthogonality relationship

$$\begin{aligned} u^*(;v) &= \sum_{i,j}^{\text{dim(U)}} u_i^* \phi_i^*(;v_j\phi_j) && (2.3.5)\\ &= \sum_{i,j}^{\text{dim(U)}} u_i^* v_j \underbrace{\phi_i^*(;\phi_j)}_{\delta_{ij}} && (2.3.6)\\ &= \sum_{i}^{\text{dim(U)}} u_i^* v_i && (2.3.7)\\ &= \langle u, v\rangle_U. && (2.3.8)\end{aligned}$$

Note that here u^* is not represented as a row vector (as was done in Sect. 1.4.4), instead the necessary tensor contraction naturally falls out of the biorthogonality relationship. The sum is still equivalent to a row-vector/column-vector product and both tensor contraction and row-vector/column-vector product representations are used throughout this work.

2.3.1 UFL Symbolic Equivalents

UFL's type system contains two types for specifying finite element functions $u(x) = \sum_i^{\dim(U)} u_i\phi_i(x) \in U$

(i) `Coefficient` for when we know and can specify the exact list of coefficients for the primal bases and

(ii) `Argument` for when we don't know the list of coefficients and, typically, want to find them out by solving some problem.

These types, particularly `Arguments`, have been designed to go in multilinear forms (they are, in each case, subclassed from a `FormArgument` type).

We have added two new equivalent types for finite element cofunctions $u^* = \sum_j^{\dim(U)} u_j^*\phi_j^* \in U^*$

[4]i.e. u and u^* are linked by the particular Riesz map that this specific inner product matrix yields

(i) `Cofunction` for when we know and can specify the exact list of coefficients to dual basis functionals and

(ii) `Coargument` for when we don't know the list of coefficients.

Finite element functions `Coefficients` and `Arguments` in UFL are created from a finite element function space U, in UFL represented by a `FunctionSpace` class which requires a symbolic meshed domain and symbolic finite element. `Cofunctions` and `Coarguments` are created from a new `DualSpace` class representing U^*. This can be accessed via a `FunctionSpace.dual()` method. An `is_dual` helper function helps to identify whether finite element spaces and their members are primal or dual. Note that if we create a `Coefficient` with a `DualSpace` we return a `Cofunction`.

It is planned that a `Cofunction` should be able to operate on a `Coefficient` or `Argument` and produce the symbolic equivalent of the action of that cofunction on the primal function. The changes are summarised in Listing 3.

```python
from ufl import *
from ufl.duals import is_dual

omega = Mesh(triangle)
U = FunctionSpace(omega, FiniteElement("CG", triangle, 1))
u = Coefficient(U)
v = Argument(U, 0)  # Argument 0 implies a test function
# Everything below is new
U_star = U.dual()
u_star = Cofunction(U_star)
u_star = Coefficient(U_star)  # this also gives a Cofunction
assert is_dual(u_star)
v_star = Coargument(U_star, 0)
v_star = Argument(U_star, 0)  # this also gives a Coargument
assert is_dual(u_star)
```

Listing 3: A summary of the new dual space and cofunction capabilities in UFL.

2.3.2 Firedrake Data Carrying Equivalents

In Firedrake we add data-carrying ability to the `Coefficient` type by subclassing it to create a `firedrake.Function` class - at this point we also add a basis function ordering. `firedrake.Functions` are created from a `firedrake.FunctionSpace` class, which requires a meshed domain, and a finite element. The `firedrake.FunctionSpace` class then creates the global Degrees of Freedom (DoFs) over the mesh. The `firedrake.Argument` class, a subclass of `ufl.Argument`, includes the DoF ordering information from the `firedrake.FunctionSpace`.

`firedrake.Cofunction` is similarly subclassed from `ufl.Cofunction` and carries data and a dual basis functional ordering. `firedrake.Cofunction` requires a `firedrake.FiredrakeDualSpace` class to instantiate. Note that if we create a `firedrake.Function` with a `firedrake.FiredrakeDualSpace` we return a `firedrake.Cofunction`.

Both `firedrake.Function` and `firedrake.Cofunction` have a `riesz_representation` method which returns the corresponding `firedrake.Cofunction` or `firedrake.Function`, respectively, for a chosen Riesz map. The changes are summarised in Listing 4.

```python
from firedrake import *
from ufl.duals import is_dual

omega = UnitSquareMesh(20, 20)
U = FunctionSpace(omega, FiniteElement("CG", triangle, 1))
u = Function(U)
v = Argument(U, 0)  # Argument 0 implies a test function
# Everything below is new
U_star = U.dual()
u_star = Cofunction(U_star)
u_star = Function(U_star)  # this also gives a Cofunction
assert is_dual(u_star)
u_star = u.riesz_representation(riesz_map="L2")
assert is_dual(u_star)
u = u_star.riesz_representation(riesz_map="L2")
assert not (is_dual(u))
v_star = Coargument(U_star, 0)
v_star = Argument(U_star, 0)  # this also gives a Coargument
assert is_dual(v_star)
```

Listing 4: A summary of the data carrying dual space and cofunction capabilities in Firedrake, using types inherited from UFL.

2.4 UFL Forms and Assembly

2.4.1 Multilinear Forms in UFL

As the name of the language implies, the Unified Form Language (UFL) allows the expression of forms. These are important since they appear when expressing variational problems. Recall that a linear variational problem is of the form ‘find $u \in U$ such that

$$a(;u,v) = L(;v) \quad \forall\, v \in V \tag{2.4.1}$$

(Eq. 1.2.1). The left hand side is a 2-linear form and the right hand side is a 1-linear form. Of course we also need to solve nonlinear variational problems, but since these are always linearised in an iterative solver, we find ourselves solving linear variational problems in practice.

Prior to this work, UFL multilinear forms were limited to the expression of definite integrals, such as the L^2 inner product in Eq. 1.4.16 with k in Eq. 1.4.11 being at most 2 (see chapter 9 for the new extension).

`Coefficients` and `Arguments`, have been designed to go in multilinear forms (they are, in each case, subclassed from a `FormArgument` type). The same UFL expression for a form

```
term_1 * term_0 * dx
```

can be a 2- 1- or 0-linear form depending on the types of `term_1` and `term_0`. Here are the three possibilities:

1. a 2-linear form
$$V_1 \times V_0 \to \mathbb{R} \tag{2.4.2}$$
such as
$$\int_\Omega u(x)v(x)dx \in \mathbb{R}\ \forall\, u \in U, v \in V \tag{2.4.3}$$
has u and v each as an instance of an `Argument`, where we instantiate u as being argument number 1 ($u \in V_1$) and v as being argument number 0 ($v \in V_0$). To match the naming used in the finite element method where we multiply our PDEs by a test function and integrate the expression

   (i) a `TrialFunction` (the variable we solve for) is an `Argument` instantiated as position 1 (here u) and

   (ii) a `TestFunction` (the variable we multiply by) is an `Argument` instantiated as argument 0 (here v).

   In UFL we write this as

```
u = TrialFunction(U)
v = TestFunction(V)
bilinear_form = u * v * dx
```

   for some appropriately defined finite element function spaces `U` and `V`.

2. A 1-linear form
$$V_0 \to \mathbb{R} \tag{2.4.4}$$
such as
$$\int_\Omega h(x)v(x)dx \in \mathbb{R} \text{ for specified } h \in H \text{ and } \ \forall\, v \in V \tag{2.4.5}$$

has h as a `Coefficient`, since it's a specific function of x, whilst v is an `Argument` instantiated as argument number 0 ($v \in V_0$ so it's a `TestFunction`). These often turn up as forcing terms in PDEs where you know a specified forcing function but still have to multiply by a test function and integrate. In UFL we write this as

```
h = Coefficient(H)
v = TestFunction(V)
one_form = h * v * dx
```

3. A 0-linear form (which is a number)

$$K \tag{2.4.6}$$

such as

$$\int_\Omega h(x)f(x)dx \in \mathbb{R} \text{ for specified } h \in H, f \in F \tag{2.4.7}$$

has both h and f as `Coefficients`. In UFL we write this as

```
h = Coefficient(H)
g = Coefficient(G)
zero_form = h * g * dx
```

2.4.2 Assembly

The integrals in equations 2.4.3, 2.4.5 and 2.4.7 look very similar but they mean very different things thanks to their UFL type information.

We realise that meaning when we *assemble* our UFL forms into a concrete output: that means giving data to the `Coefficients` which match a specific discretised function space or constant value and creating finite dimensional operators corresponding to our 2-linear and 1-linear forms. This happens whenever we come to solve a PDE, for example by calling `solve` in Firedrake: before we use our chosen solver, we need to turn our linear variational problem 'find $u \in U$ such that

$$a(;u,v) = L(;v) \quad \forall\, v \in V\text{'} \tag{2.4.8}$$

(Eq. 1.2.1) into a system of equations

$$\boldsymbol{Ax} = \boldsymbol{b} \tag{2.4.9}$$

to solve for $\boldsymbol{x}$, giving the basis coefficients of u. Assembly in Firedrake can be done outside of `solve` via an `assemble` function.

It is at this point that our changes to UFL and Firedrake become important. Whilst all libraries which use UFL have an assembly operation, UFL was not equipped to provide the outputs of assembly with a type. This made it difficult to, for example, compose operations which used the output of assembling a 1-linear form. In each case assembling now does the following:

1. the assembled 2-linear form for equation 2.4.3 where

$$u = \sum_{i=0}^{\dim(U)-1} u_i \phi_i \in U \tag{2.4.10}$$

$$v = \sum_{j=0}^{\dim(V)-1} v_j \psi_j \in V \tag{2.4.11}$$

   is written in Firedrake as

```
u = TrialFunction(U)
v = TestFunction(V)
bilinear_form = u * v * dx
assembled_bilinear_form = assemble(bilinear_form)
```

   This can be thought of mathematically as leaving the coefficients u_i and v_j of each space out of the equation. So

$$\int_\Omega u(x)v(x)dx \text{ becomes } \int_\Omega \phi_i(x)\psi_j(x)dx = M_{ij}. \tag{2.4.12}$$

   This is a $\dim(U) \times \dim(V)$ matrix which is stored in `assembled_bilinear_form`: i.e. it can be an operator that takes a vector v_j that corresponds to $v_j\psi_j$ to produces a vector u_i that corresponds to $u_i\phi_i$. If $U = V$ then then assembling the 2-linear form (the integral) performs an L^2 inner product $\langle \bullet_1, \bullet_0 \rangle_{L^2}$ and the matrix M_{ij} is the mass matrix for the finite element function space. All such matrices that result from forming an inner product on a finite element function space are assembled 2-linear forms, irrespective of whether they have been assembled by Firedrake or not.

   In a linear variational problem, the matrix M_{ij} is the matrix A_{ij} that is inverted to solve for x_i in Eq. 2.4.9.

   Whilst `bilinear_form` is symbolically a 2-linear operator, assembly concretely computes its expansion into a particular matrix given a particular basis. `assembled_bilinear_form` is now a `firedrake.Matrix` type, which inherits from a new `ufl.Matrix` class. Prior to this work, `firedrake.Matrix` was not a subclass of a UFL type.

2. The assembled 1-linear form for equation 2.4.5 where

$$h = \sum_{i=0}^{\dim(H)-1} h_i \phi_i \in H \tag{2.4.13}$$

$$v = \sum_{j=0}^{\dim(V)-1} v_j \psi_j \in V \tag{2.4.14}$$

is written in Firedrake as

```
h = Function(H)
v = TestFunction(V)
one_form = h * v * dx
assembled_one_form = assemble(one_form)
```

By default the `Function` constructor sets all basis function coefficients to zero. Introducing data to the `Function` is the topic of chapter 3. Here, assembly computes an expansion of the 1-linear form in the dual basis given by the choice of inner product (in effect, one leaves the coefficients v_j out of the equation)

$$\int_\Omega h(x)v(x)dx \text{ becomes } \sum_{i=0}^{\dim(H)-1} \int_\Omega h_i\phi_i(x)\psi_j(x)dx. \tag{2.4.15}$$

This can be thought of as a $1 \times \dim(V)$ row vector which operates on a column vector of basis function coefficients v_j (one for each ψ_j) and produces a number. The row vector is

$$v^*_{0j} = \sum_{i=0}^{\dim(H)-1} h_{0i}M_{ij} \tag{2.4.16}$$

where the ordering of h_i again corresponds to $h_i\phi_i$.[5] We call the output v^* since it must be a covector in $V^* : V \to \mathbb{R}$. If we treat it as a cofunction, as in Eq. 2.3.3, we take the j^{th} entry of the row vector as the basis cofunction coefficients

$$v^* = \sum_{j}^{\dim(V)} v^*_j\psi^*_j \tag{2.4.17}$$

$$= \sum_{j}^{\dim(V)} \sum_{i}^{\dim(H)} h_iM_{ij}\psi^*_j \in V^*. \tag{2.4.18}$$

If $H = V$ then this is an L^2 inner product again $\langle h, \bullet_0\rangle_{L^2}$: M is again referred to as the mass matrix and the 1-form is the inverse Riesz representer for H which gives the $h^* \in H^*$ that corresponds to $h \in H$. For more see section 1.4.

In a linear variational problem, the j^{th} entry in the row vector v^*_{0j} is the j^{th} entry of the right hand side vector b_j Eq. 2.4.9.

Here `one_form` is symbolically equivalent to a cofunction in V^*, assembly once again makes it concretely so. `assembled_one_form` is now a `firedrake.Cofunction` which inherits from `ufl.Cofunction`. Prior to this work, `assembled_one_form` was a `firedrake.Function` in a function space V. That worked because, as previously stated, covectors to functions in finite element function spaces have the same number of dimensions. Nevertheless this was incorrect and open to mathematical mistakes.

3. Lastly the assembled 0-linear form for equation 2.4.7 where

$$h = \sum_{i=0}^{\dim(H)-1} h_i \phi_i \in H \tag{2.4.26}$$

$$f = \sum_{j=0}^{\dim(F)-1} f_j \psi_j \in F \tag{2.4.27}$$

is written in Firedrake as

```
h = Function(H)
f = Function(F)
zero_form = h * f * dx
assembled_zero_form = assemble(zero_form)
```

[5]In the specific case of equation 2.4.5 and its assembled equivalent 2.4.16 the calculation we are doing is

$$\int_\Omega h(x)v(x)dx = \sum_{i=0}^{\dim(H)-1} \sum_{j=0}^{\dim(H)-1} \int_\Omega h_i\phi_i(x)v_j\psi_j(x)dx \tag{2.4.19}$$

where v_j are unknown. If we factor out v_j we get an integral we can actually evaluate as a row-vector/matrix/column-vector product.

$$\begin{aligned}\int_\Omega h(x)v(x)dx &= \sum_{i=0}^{\dim(H)-1} \sum_{j=0}^{\dim(H)-1} \int_\Omega h_i\phi_i(x)\psi_j(x)dx\ v_j && (2.4.20)\\ &= \sum_{i=0}^{\dim(H)-1} \sum_{j=0}^{\dim(H)-1} h_{0i}M_{ij}v_j. && (2.4.21)\end{aligned}$$

Since we don't yet know v_j we have a row-vector which we treat as an operator

$$v^*_{0j} = \sum_{i=0}^{\dim(H)-1} h_{0i}M_{ij}. \tag{2.4.22}$$

This is equivalent to treating our unknown argument $v \in V$ as the set of raw basis functions ψ_j

$$\begin{aligned}v^*_{0j} &= \int_\Omega h(x)\psi_j(x)dx && (2.4.23)\\ &= \sum_{i=0}^{\dim(H)-1} \int_\Omega h_i\phi_i(x)\psi_j(x)dx && (2.4.24)\\ &= \sum_{i=0}^{\dim(H)-1} h_{0i}M_{ij}. && (2.4.25)\end{aligned}$$

and can be thought of mathematically as leaving all the coefficients in

$$\int_\Omega h(x)f(x)dx = \sum_{i=0}^{\dim(H)-1} \sum_{j=0}^{\dim(F)-1} \int_\Omega h_i\phi_i(x) f_j\psi_j(x)dx \tag{2.4.28}$$

This is a real number which corresponds to

$$\sum_{i=0}^{\dim(H)-1} \sum_{j=0}^{\dim(F)-1} h_{0i} M_{ij} f_j. \tag{2.4.29}$$

`assembled_zero_form` will contain this value. If $H = F$ then this is the specific value for the L^2 inner product of these functions $\langle h, f\rangle_{L^2}$.

So whilst `zero_form` was symbolically a number, assembly made it concretely so, i.e. `assembled_zero_form` is a Python `float`. This has not been changed.

Chapter 3

Getting Data into Finite Element Functions: FInAT Dual Evaluation Interpolation

3.1 Motivation

Before we can start solving PDEs using the finite element method, we need a way to take expressions such as $f(x, y) = x^2 + y$ and represent them on our mesh. Furthermore, we might have a solution to one PDE in one finite element function space which we want to use in another.

In this chapter two ways to do that are introduced, one of which - *dual evaluation* - we will use throughout this thesis. As we will see, dual evaluation is computationally efficient and, as will be demonstrated throughout this thesis, has wide applications. After the definitions, an efficient implementation of dual evaluation will be demonstrated, which is able to exploit the structure of certain finite elements to reduce computational cost. It does this by making use of the pre-existing high-level abstractions used by Firedrake to compose finite elements.

3.2 Dual Evaluation Interpolation

In most implementations of finite element methods we have a set of 'global' coordinates x covering our meshed domain and a set of 'local' coordinates $\tilde{x}$ defined on some reference cell. For each mesh cell we transform from global to local coordinates, perform an operation, then transform our result back. *Nodal dual evaluation*, also referred to as simply *dual evaluation*, is a way of representing arbitrary functions in finite element function spaces which can be done in a straightforward cell-by-cell manner using these transformations.

This is an outline of the definition of dual evaluation interpolation found in Sect. 3.3 of [35] using our notation. As a reminder, finite elements on a reference cell can be represented using a Ciarlet triple [34]

$$(\mathcal{K}, \mathcal{P}, \mathcal{N}) \tag{3.2.1}$$

where $\mathcal{N} = \{\tilde{\phi}_i^*\}_{i=0}^k$ is a set of $k+1$ cofunctions which are defined on the reference cell $\mathcal{K}$ and are known as the *nodal dual basis*. The nodal dual basis is often shortened to the *dual basis* or the *nodes*.

3.2.1 Local Interpolation

Local interpolation $\mathcal{I}_{\mathcal{P}(\mathcal{K})}$ of some locally defined function $\tilde{f}$ into the local function space $\mathcal{P}(\mathcal{K})$ is given by the linear operator

$$\left[\mathcal{I}_{\mathcal{P}(\mathcal{K})}(;\tilde{f})\right](\tilde{x}) = \sum_{i=0}^{k-1} \tilde{\phi}_i^*(;\tilde{f})\tilde{\phi}_i(\tilde{x}) \tag{3.2.2}$$

which we see is the *evaluation* of the dual basis $\{\tilde{\phi}_i^*\}_{i=0}^k$ on the function $\tilde{f}$. Each dual evaluation $\tilde{\phi}_i^*(;\tilde{f})$ gives us the coefficient for the corresponding basis function $\tilde{\phi}_i(\tilde{x})$.

Strictly speaking, if $\tilde{\phi}_i^*$ are some set of cofunctions to $\tilde{\phi}_i$, as Ciarlet's finite element definition

[34] states, then

$$\tilde{\phi}_i^* : \mathcal{P}(\mathcal{K})^* \to \mathbb{R}. \tag{3.2.3}$$

This implies that $\tilde{f} \in \mathcal{P}(\mathcal{K})$, in which case

$$\tilde{f}(x) = \sum_{j=0}^{k-1} \tilde{f}_j \tilde{\phi}_j(x) \tag{3.2.4}$$

and

$$\begin{aligned}
\left[\mathcal{I}_{\mathcal{P}(\mathcal{K})}(\tilde{f})\right](x) &= \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} \tilde{\phi}_i^*(;\tilde{f}_j\tilde{\phi}_j)\tilde{\phi}_i(x) && (3.2.5)\\
&= \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} \tilde{\phi}_i^*(;\tilde{\phi}_j)\tilde{f}_j\tilde{\phi}_i(x) && (3.2.6)\\
&= \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} \delta_{ij}{}^{1}\tilde{f}_j\tilde{\phi}_i(x) && (3.2.7)\\
&= \sum_{j=0}^{k-1} \tilde{f}_j\tilde{\phi}_j(x) = \tilde{f}(x). && (3.2.8)
\end{aligned}$$

This isn't really an interpolation operation, since it gives us the function we already had. We want the operation to take a function from a different function space $\bar{\mathcal{P}}(\bar{\mathcal{K}})$ into this one. I.e.

$$\mathcal{I}_{\mathcal{P}(\mathcal{K})} : \bar{\mathcal{P}}(\bar{\mathcal{K}}) \to \mathcal{P}(\mathcal{K}) \tag{3.2.9}$$

This requires us to define an extension to our dual basis functionals, let us call them $\bar{\tilde{\phi}}_i^*(;\tilde{f})$, as

$$\bar{\tilde{\phi}}_i^*(;\tilde{f}) : \bar{\mathcal{P}}(\bar{\mathcal{K}}) \to \mathbb{R}. \tag{3.2.10}$$

This can be done without needing to redefine the dual basis as long they can be evaluated in the new space $\bar{\mathcal{P}}(\bar{\mathcal{K}})$. For example $\bar{\tilde{\phi}}_i^*$ can't require the evaluation of a point on the source reference cell $\bar{\mathcal{K}}$ which is not in target reference cell $\mathcal{K}$. So long as this can be done with sufficient accuracy in the function space we are interpolating from, we can perform our interpolation and get results which have properties which we expect the new space to give us, such as having particular values at particular points (for more see Maddison and Farrell [36]).

If $\tilde{f} \in \bar{\mathcal{P}}(\bar{\mathcal{K}})$, i.e.

$$\tilde{f}(x) = \sum_{j=0}^{l-1} \tilde{f}_j \tilde{\psi}_j(x) \tag{3.2.11}$$

[1]from Ciarlet's definition

where $\tilde{\psi}_j \in \bar{\mathcal{P}}(\bar{\mathcal{K}})$ and there are l local basis functions then we have

$$\left[\mathcal{I}_{\mathcal{P}(\mathcal{K})}(\tilde{f})\right](x) = \sum_{i=0}^{k-1}\sum_{j=0}^{l-1} \bar{\tilde{\phi}}_i^*(;\tilde{\psi}_j)\tilde{f}_j\tilde{\phi}_i(x) \tag{3.2.12}$$

$$= \sum_{i=0}^{k-1}\sum_{j=0}^{l-1} \tilde{A}_{ij}\tilde{f}_j\tilde{\phi}_i(x) \tag{3.2.13}$$

$\tilde{A}_{ij}$ is the $k \times l$ local interpolation matrix where each $\tilde{A}_{ij} = \bar{\tilde{\phi}}_i^*(;\tilde{\psi}_j)$.

It is important to highlight that the interpolation operator is unique for a particular choice of extended dual basis $\bar{\tilde{\phi}}_i^*$. Let $\bar{\tilde{\phi}}_i^*$ be mappings from some function space $W \to \mathbb{R}$ (W being the space of continuous functions $\mathbb{C}^0$ for example), making them members of W^*. These can then be restricted to mappings from $\mathcal{P}(\mathcal{K}) \to \mathbb{R}$ giving the nodal dual basis $\tilde{\phi}_i^* \in \mathcal{P}(\mathcal{K})^*$.

There may be several functionals in W^*, such as nodes of another finite element basis, which have the same restriction to $\mathcal{P}(\mathcal{K})^*$. For example, a 1 dimensional functional $\bar{\tilde{\phi}}_i^*$ might be defined as the integral of the reference cell. When restricted to act on linear polynomials, this will be equal to a particular point evaluation on the cell. In that case, one could equivalently define $\bar{\tilde{\phi}}_i^*$ as that point evaluation. The two extensions are equivalent for the linear polynomials but not for functions in W. In Firedrake, via FIAT, the extended dual basis, and therefore the nodes of finite elements, are defined as quadrature sums (see Sect. 3.7).

3.2.2 Global Interpolation

Global interpolation over the entire mesh Ω for the complete finite element function space $\mathrm{FS}(\Omega)$, which we denote $\mathcal{I}_{\mathrm{FS}(\Omega)}$, is the cell-local application (i.e. transformed to local reference coordinates) of $\mathcal{I}_{\mathcal{P}(\mathcal{K})}$ to a globally defined function $f \in V$.

$$\mathcal{I}_{\mathrm{FS}(\Omega)} : V \to \mathrm{FS}(\Omega) \tag{3.2.14}$$

If we exclusively consider global space, where we have a global set of $\dim(\mathrm{FS}(\Omega))$ basis functions $\phi_i \in \mathrm{FS}(\Omega)$ and associated extended dual basis functionals $\bar{\phi}_i^*$ then global interpolation is

$$\left[\mathcal{I}_{\mathrm{FS}(\Omega)}(;f)\right](x) = \sum_{i=0}^{\dim(\mathrm{FS}(\Omega))-1} \bar{\phi}_i^*(;f)\phi_i(x). \tag{3.2.15}$$

If $f \in V$ has its own global basis $\psi_j \in V$, i.e.

$$f(x) = \sum_{j=0}^{\dim(V)-1} f_j\psi_j(x) \tag{3.2.16}$$

then

$$\left[\mathcal{I}_{\mathrm{FS}(\Omega)}(;f)\right](x) = \sum_{i=0}^{\dim(\mathrm{FS}(\Omega))-1} \sum_{j=0}^{\dim(V)-1} \bar{\phi}_i^*(;f_j\psi_j)\phi_i(x) \tag{3.2.17}$$

$$= \sum_{i=0}^{\dim(\mathrm{FS}(\Omega))-1} \sum_{j=0}^{\dim(V)-1} \bar{\phi}_i^*(;\psi_j)f_j\phi_i(x) \tag{3.2.18}$$

$$= \sum_{i=0}^{\dim(\mathrm{FS}(\Omega))-1} \sum_{j=0}^{\dim(V)-1} A_{ij}f_j\phi_i(x). \tag{3.2.19}$$

A_{ij} , with entries $\phi_i^*(;\psi_j)$, is the $\dim(\mathrm{FS}(\Omega)) \times \dim(V)$ global interpolation matrix A_{ij}. Here, the union of all function spaces to which all $\bar{\phi}_i^*$ can be applied is referred to as X.

Note that for cross mesh interpolation, we can only define this as the cell-by-cell application of $\mathcal{I}_{\mathcal{P}(\mathcal{K})}$ if each local dual basis functional $\tilde{\phi}_i^*$ can be evaluated on the source mesh's local reference cell. If not, we can still define the global interpolator as in Eq. 3.2.15, but must evaluate each extended global dual basis function $\bar{\phi}_i^*$ in global coordinates. For this to be a valid extension we usually strictly require the finite element function spaces to be continuous at cell boundaries for the extended dual basis to be well defined (the functionals might involve a point evaluation at a cell boundary) but this is not always explicitly checked for. Cross mesh interpolation is returned to in chapter 8.

A note on nomenclature: we generally assume that the dual basis functionals are valid, and therefore usually write $\bar{\phi}_i^*$ as ϕ_i^*.

3.2.3 An example of Local Dual Evaluation

Here and in the rest of this chapter, local coordinates and local dual evaluation will be used exclusively. To avoid unnecessary visual clutter, the tildes and hats are disregarded, i.e. $\tilde{f}$ is now f, $\tilde{\phi}_i$ is now ϕ_i, and $\tilde{x}$ is now x.

Take for example the second order Lagrange basis polynomials on a line in figure 3.1 each of which are 1 at each of $\{0, 0.5, 1\}$ and 0 elsewhere. These points are decided in the Ciarlet triple finite element definition by the dual basis functionals

$$\begin{aligned} \phi_0^*(;f) &= f(0) \\ \phi_1^*(;f) &= f(0.5) \\ \phi_2^*(;f) &= f(1) \end{aligned} \tag{3.2.20}$$

which then define the element function space $\mathcal{P}(\mathcal{K})$ as the span of basis functions $\{\phi_j\}$ that

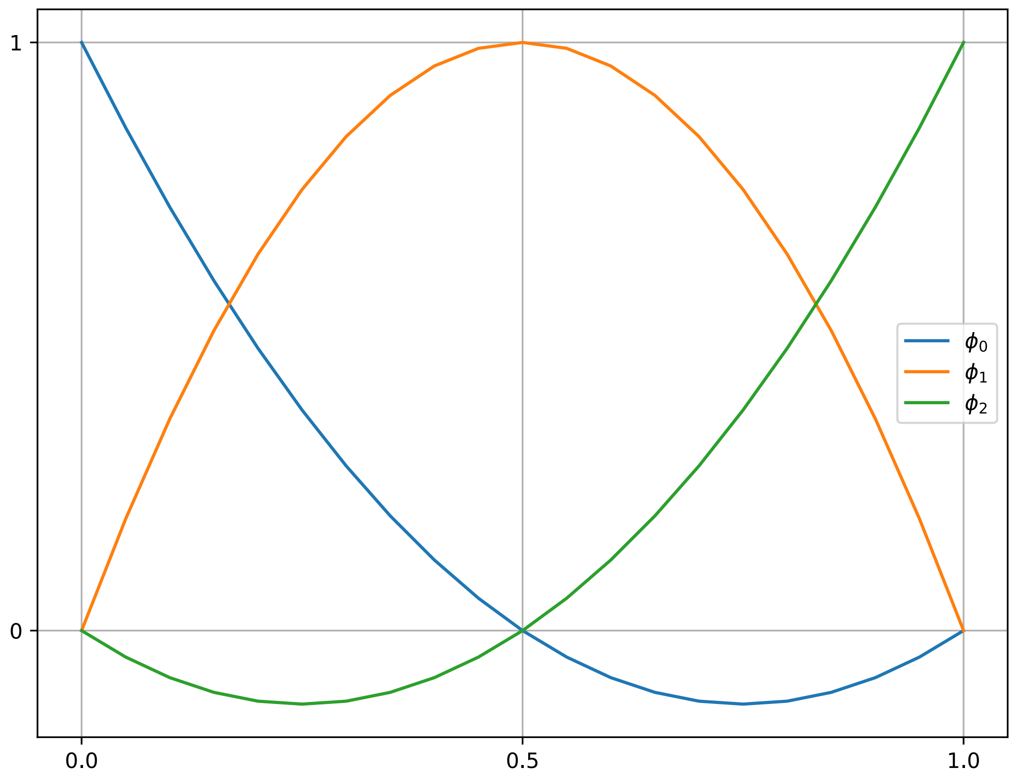


Figure 3.1: The second order Lagrange basis polynomials on an interval $[0, 1]$.

obey the biorthogonality relationship

$$\phi_i^*(; \phi_j) = \delta_{ij} = \begin{cases} 1 & \text{if } i = j, \\ 0 & \text{otherwise.} \end{cases} \tag{3.2.21}$$

When we interpolate some function or expression f into this space we need a set of coefficients for these basis functions which are the values of f at the locations where the basis functions have magnitude 1. If $f \in \mathcal{P}(\mathcal{K})$ these correspond to the dual evaluation of f with the correct dual basis functional to then perform the correct point evaluations

$$\begin{aligned} \left[\mathcal{I}_{\mathcal{P}(\mathcal{K})}(f)\right](x) &= f_0\phi_0(x) + f_1\phi_1(x) + f_2\phi_2(x) \\ &= f(0)\phi_0(x) + f(0.5)\phi_1(x) + f(1)\phi_2(x) \\ &= \phi_0^*(; f)\phi_0(x) + \phi_1^*(; f)\phi_1(x) + \phi_2^*(; f)\phi_2(x). \end{aligned} \tag{3.2.22}$$

To calculate this for functions which are not members of $\mathcal{P}(\mathcal{K})$ we need an extension to the dual basis functionals. The dual basis in equation 3.2.20 is made up of so-called *point evaluation nodes*: they are the evaluations of f at given points. To extend the dual basis functionals, we take the obvious choice of using point evaluations at the same locations

$$\begin{aligned}\bar{\phi}_0^*(;g) &= g(0)\\ \bar{\phi}_1^*(;g) &= g(0.5)\\ \bar{\phi}_2^*(;g) &= g(1)\end{aligned} \tag{3.2.23}$$

The extended dual basis is valid as long as we can evaluate g at 0, 0.5 and 1 on the reference element.

3.3 Galerkin Projection

In Sect. 3.2, dual evaluation was described as *a* way of representing arbitrary functions in finite element function spaces. The other commonly used approach is Galerkin projection. This is not the topic of this work, but it is worth briefly discussing the differences between the two.

For example, given some arbitrary function f, then the function $f_h \in \mathrm{FS}(\Omega)$ such that

$$\langle v, f_h \rangle_{L^2} = \langle v, f \rangle_{L^2} \quad \forall\, v \in \mathrm{FS}(\Omega) \tag{3.3.1}$$

is the L^2 Galerkin projection of f onto V. Galerkin projections can also be found for other inner products.

Finding f_h given some f is a problem that one can specify and solve, but it is not something that can usually be done cell-by-cell. This is inherantly more computationally expensive than dual evaluation. For example, to solve Eq. 3.3.1 we have to solve a matrix-vector equation with the *global* mass matrix for the function space on the domain of interest.

Of course, the solution to a Galerkin projection problem is not the same thing as the result of dual evaluation interpolation. L^2 Galerkin projection can introduce new maxima and minima into the solution which are not present in the original function f, whereas dual evaluation interpolation will not. L^2 Galerkin projection can, however, conserve certain properties of f which dual evaluation interpolation will not necessarily conserve, particularly when moving between meshes. Conservative interpolation is possible though, see the brief discussion in Sect. 8.1. Dual evaluation interpolation, as defined here, being an operator rather than an equation which must be solved makes it a more versatile mathematical tool: it can be placed within expressions and usually has a well defined derivative.

3.4 Existing Dual Basis and Interpolation Implementations

It is usual for systems which support multiple finite elements to include some package which represents each finite element for a given reference cell. Both Firedrake and legacy FEniCS use

FIAT [37] for this: for each element it provides a method for evaluating local basis functions at one or more reference cell points, a process known as *tabulation*[2]; tabulation will be returned to in chapter 4. FIAT derives the local basis functions from the local dual basis functions of a given element, as in Ciarlet's definition [34] from which the extended dual basis can be directly taken. The symbolic finite element definition library Symfem [38] and FEniCSx's tabulation library Basix [39] do the same. The finite element solver frameworks NGSolve [22] and MFEM [31, 32] also give elements a dual basis, though the element definitions are not in a separate library. Some other frameworks where this is documented such as DUNE [40] (used by both the DUNE-Fem [18], DUNE-PDELab [41] frameworks), FREEFEM [21] and Finch [20] implement the finite element directly without appealing to the nodal dual basis.

In principal anywhere we have a finite element definition which includes a dual basis should allow for dual evaluation interpolation. One merely has to coordinate transform a function or expression from each mesh cell to the local coordinates where the dual bases are defined, apply them to the function or expression to get the local basis function coefficients, then transform the result back to global coordinates. This is what Firedrake does with FIAT, FEniCSx does with Basix and NGSolve does with its internal library of finite elements. MFEM has a `GetLocalInterpolation` method of its finite element class which, from inspecting the source code, appears to retrieve the interpolation matrix using the nodal dual basis in certain cases, though this is not documented in the literature or the API documentation. The DUNE framework provides tools for nodal interpolation (i.e. dual evaluation) for certain finite elements, though apparently appeals to L^2 Galerkin Projection and other methods in some cases [42]: it is not made clear in Engwer et al. [42], Sander [40] or in the API documentation which operation is performed for a given element. Neither FREEFEM nor Finch appear to have any explicit dual evaluation interpolation functionality.

3.4.1 Firedrake Dual Evaluation

Exactly how one gets from a function or expression on global space to local coordinates and back again is, unfortunately, not documented for dual evaluation for any of the aforementioned libraries. Finite element systems usually perform calculations cell-by-cell by building a so-called *kernel* which runs on each mesh cell in turn. In Firedrake (and, presumably, elsewhere), we do this for dual evaluation interpolation.

The Two Stage Form Compiler (TSFC, Homolya et al. [8]) builds a kernel which (1.) transforms a given function or expression from its global coordinates on a mesh cell to local reference cell coordinates, (2.) performs the dual evaluation operation (prior to this work, using FIAT as described in Sec. 3.7), then (3.) transforms the result back to global coordinates.

This kernel needs to be run over each mesh cell and deal with mesh parallel domain decomposition parallelism (see chapter 7, Sect. 7.2). The kernel needs to be supplied with data from

[2]hence FIAT standing for FInite element Automatic Tabulator

global space and the result needs to be supplied back to a new data structure in global space which corresponds to a new finite element function. This is all handled by the PyOP2 library [15]. Firedrake then wraps this in a high level API: see listing 5.

Note that Firedrake's interpolation operation can be used on UFL expressions which may be nonlinear in their arguments, as on line 29 of Listing 5. When that occurs, it is not a linear operator since the expression itself requires evaluation before it can be dual evaluated. This has implications on finding its derivative, which we do in chapter 5, Sect. 5.5. For now, when we discuss the dual evaluation interpolation operator, we treat it as being strictly linear.

```
1  from firedrake import *
2  import numpy as np
3  
4  # Create a mesh and two function spaces
5  omega = UnitSquareMesh(2, 2)
6  P1CG = FunctionSpace(omega, "CG", 1)
7  P2CG = FunctionSpace(omega, "CG", 2)
8  
9  w = Function(P2CG)  # w is a function in P2CG
10 
11 # Any of the following interpolate w in P2CG into u in P1CG
12 u = interpolate(w, P1CG)  # interpolate into the function space
13 interpolate(w, u)         # interpolate into the function u
14 u.interpolate(w)          # use the interpolate method
15 i = Interpolator(w, u)    # create a a reusable interpolator...
16 i.interpolate()           # ...and use it to fill u
17 
18 # We can also create the interpolation matrix A : P2CG -> P1CG
19 A = Interpolator(TestFunction(P2CG), P1CG)
20 # and use it to interpolate w in P2CG into some v in P1CG
21 v = A.interpolate(w)
22 # and get the adjoint operation A^* : P1CG^* -> P2CG^*
23 u_star = assemble(TestFunction(P1CG) * dx)  # a cofunction in P1CG^*
24 v_star = Cofunction(P2CG.dual())            # a cofunction in P2CG^*
25 A.interpolate(u_star, transpose=True, output=v_star)
26 
27 # Interpolation also provides a way to set function values from symbolic
28 # expressions
29 x, y = SpatialCoordinate(omega)
30 w.interpolate(x**2 + y**2)
31 assert np.isclose(w.at(0.5,0.5), 0.5)
32 
33 # Functions themselves can also be included in symbolic expressions
34 g = Function(P1CG).interpolate(2*w)
35 assert np.isclose(g.at(0.5,0.5), 1.0)
```

Listing 5: Firedrake's dual evaluation interpolation API.

3.4.2 FInAT

FInAT [14] inherits and wraps all of FIAT's element definitions and provides performance enhancements for elements which can be defined in terms of the outer product of lower rank equivalents (quadrilateral reference cells being given as the outer product of two interval reference cells for example, as shown in figure 3.2); tabulating on such elements is able to make use of this tensor product structure in a process known as 'sum-factorisation' [43]. See section 3.8.2 for more. Similar capability is also available in MFEM, deal.II [33] and, to a limited extent,in Basix.

Firedrake, via the kernel builder TSFC, uses FInAT to improve element tabulation performance when solving PDEs. Whilst FInAT wraps all of FIAT's element definitions it does not provide a wrapper of the dual basis. Such a wrapper would need to allow elements to be composed as tensor products of one another such that FInAT and TSFC's sum-factorisation machinery can work. The tensor algebra Domain Specific Language 'GEM'[3], which is built into TSFC, is used extensively in FInAT, for example when describing a set of points. It is through symbolic manipulation of 'GEM' that tensor-product structure is preserved by FInAT. It therefore makes sense that the dual basis of elements should be advertised as GEM tensors and that their composition and eventual dual evaluation should be expressed as tensor manipulations in GEM.

3.5 A primer on tensors in UFL and GEM

TSFC's primary Domain Specific Language (DSL), 'GEM' is used throughout this chapter. Whilst it is explained in detail in Homolya et al. [8], key terms are also explained here.

Expressions in finite element function spaces, expressed in Firedrake using UFL, generally boil down to large tensor contractions. A 0-linear form for an L^2 inner product, for example, is a vector-matrix-vector product where the two vectors are the coefficients of the two sets of basis functions and the matrix is the mass matrix. GEM, which can be compiled from UFL, ensures that information such as 'which vector contracts with which matrix index' is maintained at a symbolic level, and allows optimisations such as sum factorisation to be expressed symbolically prior to further compilation into loops.

Many of its types are inherited directly from UFL, which is aware of the tensor-contraction nature of the calculations involved. GEM jettisons the finite element specific parts of UFL, which are not necessary in TSFC, and introduces new types which are particularly useful for constructing kernels.

Tensors in UFL have *shape* and *free indices*. This is most easily explained through demonstration. Imagine we have a tensor $\boldsymbol{M}$ which has dimensions 2×3 (i.e. it is a matrix). In UFL this can be represented as

[3]GEM is not an acronym, just the name of the language

1. a tensor valued object $\boldsymbol{M}$: this has shape $(2,3)$ but no free indices.

2. A scalar object M_{ij}: this has shape $()$ and two *free indices* i and j where i has an extent of 2 and j has an extent of 3.

3. A vector valued object $\boldsymbol{M}_i$: this has shape $(3,)$ and a free index i of extent 2.

4. A vector valued object $\boldsymbol{M}_j$: this has shape $(2,)$ and a free index j of extent 3.

Shape is always ordered: $\boldsymbol{M}$ has dimensions 2×3 (2 rows and 3 columns) so its shape is $(2,3)$. Free indices are unordered but are labelled and have an extent: once another tensor is introduced they represent dimensions to sum along in a contraction. For more see page 8 of Homolya et al. [8]. Free indices are represented by an `Index` type.

GEM inherits UFL's definition of shape, free indices and an `Index` type, and introduces new types which are summarised on page 13 of Homolya et al. [8]. Of particular note are `Literal` and `Variable`.

- `Literal` represents a tensor which is concretely known at the moment of creation (i.e. during creation of the TSFC kernel). These are initialised from a numpy [44] `ndarray`, the shape of which is used as the shape of the tensor. These have no free indices.

- `Variable` represents a tensor which is not concretely known at the moment of creation (i.e. which will be supplied to the TSFC kernel after it has been created). These are initialised with some shape, and again have no free indices.

GEM expressions can be built from `Literal` and `Variable` using `Indexed`, `IndexSum`, `ComponentTensor`, and `ListTensor`. These are also inherited from UFL and are strictly defined on pages 9 of Homolya et al. [8]. To the definitions, here are illustrative examples:

- `Indexed`: An expression representing a tensor which has been indexed: it has shape $()$ and 0 or more free indices. To quote Homolya et al. [8] "`Indexed` can convert shape to free indices".

  For example, if `A` is a tensor valued expression with shape $(2,3)$ (i.e. it corresponds to a matrix $\boldsymbol{A}$ of dimension 2×3) then

  – `Indexed(A, (Index(1), Index(2))` is the scalar valued expression A_{ij} with shape $()$ and two free indices, `Index(1)` with extent 2, and `Index(2)` with extent 3,

  – `Indexed(A, (Index(1), 1))` is the scalar valued expression A_{i1} with shape $()$ and one free index, `Index(1)` with extent 2, and

  – `Indexed(A, (0, 1))` is the scalar valued expression A_{01} with shape $()$ and no free indices.

- `IndexSum`: An expression for summing along a given free index.

  Given a scalar valued expression `B`, corresponding to B_{ij} with shape () and free indices `Index(1)` corresponding to i and `Index(2)` corresponding to j (B_{ij} is typically an `Indexed`) then

  - `IndexSum(B, (Index(1), Index(2)))` is an expression for summation along indices i and j, i,e, $\sum_{i,j} B_{ij} = c$. The result has no free indices.
  - `IndexSum(B, (Index(1),))` is an expression for summation along index i, i,e, $\sum_i B_{ij} = C_j$. The result now has a single free index `Index(2)` corresponding to j.
  - `IndexSum(B, (Index(2),))` is an expression for summation along index j, i,e, $\sum_j B_{ij} = D_i$. The result now has a single free index `Index(1)` corresponding to i.

- `ComponentTensor`: An expression representing a tensor with shape and 0 or more free indices. These can turn free indices back into shape.

  For example, given the scalar valued expression `B`, corresponding to B_{ij} from the `IndexSum` example above, then

  - `ComponentTensor(B, (Index(1),))` is a vector valued expression with shape $(\text{extent}(i),)$ and a single free index `Index(2)` corresponding to j. This represents $\boldsymbol{B}_j$.
  - `ComponentTensor(B, (Index(2),))` is a vector valued expression with shape $(\text{extent}(j),)$ and a single free index `Index(1)` corresponding to i. This represents $\boldsymbol{B}_i$.
  - `ComponentTensor(B, (Index(1), Index(2)))` is a scalar expression with shape $(\text{extent}(i), \text{extent}(j))$ and no free indices. This represents $\boldsymbol{B}$.
  - `ComponentTensor(B, (Index(2), Index(1)))` is a scalar expression with shape $(\text{extent}(j), \text{extent}(i))$ and no free indices. This represents $\boldsymbol{B}^T$.

- `ListTensor`: An expression representing a concatenation of tensor expressions along a new 1st dimension. Given two tensor expressions `E` and `D`, each with shape $(3, 3)$ the `ListTensor(A, B)` created from them will have shape $(2, 3, 3)$.

3.6 Summary of Contributions

Section 3.8 contains the main contributions. I firstly came up with a generalised formulation of dual evaluation as a tensor contraction (equation 3.8.1). I implemented this in FInAT[4], making full use of GEM to represent the tensors involved, such that highly optimised dual evaluation

[4]See https://github.com/FInAT/FInAT/pull/89 for FInAT changes.

routines could be compiled with TSFC. This required adding an API for dual evaluation to FInAT elements, where each element advertises a dual basis and performs the tensor contraction in a general dual evaluation routine.

Appropriate dual bases have been implemented for most elements:

1. Where the tensor contraction would be multiplying by an identity tensor, this is indicated symbolically (equation 3.8.7) such that code is not generated in that case.

2. Where elements are wrappers around other elements which raise their rank ('tensor elements') the new dual basis and dual evaluation routines allow for an optimisation known as *delta elimination* (see section 3.8.1).

3. Where elements have a tensor product structure ('tensor product elements'), the structure is preserved such that TSFC can perform relevant optimisations such as *sum-factorisation* (see section 3.8.2).

Being able to maintain structure has led to a dramatic improvement in performance (see the end of section 3.8.2). Mathematical expressions for the calculations performed can be found throughout section 3.8. Care had to be taken to ensure compatibility with interpolation onto elements where the dual basis is specified after code is generated (i.e. at run-time), such as when interpolating onto a `VertexOnlyMesh` (introduced in chapter 4). Cases where expressions contain unknowns also had to be catered for. Some specific further optimisations have been suggested, see section 3.8.3.

TSFC and Firedrake also had to be modified[5,6] to use this new API and various bugs had to be fixed. Tests were added throughout. An initial implementation of FInAT dual evaluation, which moved the existing FIAT dual evaluation code (discussed in section 3.7) from TSFC into FInAT, had been done by Matthew Kan[7] as part of a masters project: this was used as a base for this work but has largely been superseded (dual evaluation being a tensor contraction for example). Lawrence Mitchell[8] also contributed by giving advice and performing edits.

3.7 Existing FIAT Dual evaluation

We start by looking at the pre-existing implementation. Consider the example of this invented element which consists of 3rd order polynomials (4 degrees of freedom) on a reference interval

[5]See `https://github.com/firedrakeproject/tsfc/pull/250` for TSFC changes.
[6]See `https://github.com/firedrakeproject/firedrake/pull/2115` for Firedrake changes.
[7]ting.kan19@imperial.ac.uk
[8](Formerly) Department of Computer Science, Durham University

cell

$$\begin{aligned}
\phi_0^*(;f) &= f(0)\\
\phi_1^*(;f) &= f(1)\\
\phi_2^*(;f) &= \int_0^1 f(x)\,dx \\
\phi_3^*(;f) &= \int_0^1 xf(x)\,dx.
\end{aligned} \tag{3.7.1}$$

In FIAT, MFEM and Basix all dual basis functionals (nodes) are represented as weighted sums of function evaluations at particular points

$$\phi_i^*(;f) = \sum_j w_j f(x_j) \tag{3.7.2}$$

which are also the extended dual basis

$$\bar{\phi}_i^*(;f) = \sum_j w_j f(x_j). \tag{3.7.3}$$

This extension is well-defined on all functions that are defined at the quadrature points. In FIAT, this is calculated via a complicated data structure based around Python dictionaries.[9] For point evaluation nodes such as ϕ_0^* we have, from the biorthogonality relationship (equation 3.2.21), $w_j = \delta_{0j} = 1$ when $j = 0$ and that $x_j = x_0 = 0$. Similarly for ϕ_1^* we have $w_j = \delta_{1j} = 1$ when $j = 1$ and that $x_j = x_1 = 1$. For integral nodes (often called 'moment' nodes) on an interval, such as ϕ_2^* and ϕ_3^* in equation 3.7.1, an exact Gaussian quadrature rule for evaluating the integral with 3rd order polynomials is used by FIAT:

$$\phi_2^*(;f) = \int_0^1 f(x)\,dx = \sum_j w_j f(x_j) = \\ \frac{5}{18} \times f\left(\frac{1-\sqrt{\frac{3}{5}}}{2}\right) + \frac{4}{9} \times f(0.5) + \frac{5}{18} \times f(0.88729833..) \tag{3.7.4}$$

$$\phi_3^*(;f) = \int_0^1 xf(x)\,dx = \sum_j w_j f(x_j) = \\ 0.03130602.. \times f\left(\frac{1-\sqrt{\frac{3}{5}}}{2}\right) + \frac{2}{9} \times f(0.5) + 0.24647176.. \times f(0.88729833..) \tag{3.7.5}$$

The vast majority of elements employ some combination of point evaluation and integral

[9]NGSolve represents basis functionals as specially defined C++ lambda functions for each finite element.

nodes. For derivative nodes such as those found on Hermite elements [35] a similar representation as weighted sums can be done (and is found in FIAT) but has yet to be implemented in FInAT.

Code for performing dual evaluations of point and integral nodes was added to the dual evaluation kernel compilation routines in TSFC in 2020 by bypassing FInAT and using the FIAT element representation directly. Each dual basis functional weighted sum in i was considered sequentially for a given function or expression f to interpolate via dual evaluation. This caused any element structure, such as the ability to perform sum-factorisation, to be lost. Dual evaluation kernels therefore were unoptimised and used many more FLoating point OPerations (FLOPs) than necessary. They also contained unnecessarily unrolled loops; this made them difficult to understand and increased their memory requirements. In the next section, we discuss how these problems were solved by making full use of FInAT.

3.8 FInAT Dual Evaluation

In general, dual basis evaluation can be represented as a contraction of an evaluation tensor Q with a tensor-valued function to dual evaluate $\boldsymbol{f} \in \mathbb{R}^{S_1 \times \ldots \times S_N}$ applied to each point in a point set tensor x (which is lower ranked than Q):

$$\phi_i^*(; \boldsymbol{f}) = \sum_{j,k,l,\ldots,\beta_1,\ldots,\beta_N} Q_{ijkl\ldots\beta_1\ldots\beta_N} f_{\beta_1\ldots\beta_N}(x_{jkl\ldots}). \tag{3.8.1}$$

where the index i covers all the dual basis functionals and the indices represented by greek letters correspond to the tensor components of $\boldsymbol{f}$

$$\text{extent}(\beta_1) = S_1, \ \ldots, \ \text{extent}(\beta_N) = S_N \tag{3.8.2}$$

and

$$f_{\beta_1\ldots\beta_N}(x) = [\boldsymbol{f}(x)]_{\beta_1\ldots\beta_N} \tag{3.8.3}$$

is an indexing operation on $\boldsymbol{f}$ that gives its scalar components. Note that this expression applies to elements whose basis functions are scalar or tensor valued but does not apply to FInAT `TensorFiniteElement` which is described later.

For the invented element in equation 3.7.1 where f evaluates a single point x_j (i.e. there are no higher indices kl etc. which feature in the dual evaluation) and $f \in \mathbb{R}$ then equation 3.8.1 is the matrix-vector contraction:

$$\phi_i^*(; f) = \sum_j Q_{ij} f(x_j) \tag{3.8.4}$$

$$\begin{pmatrix} \phi_0^*(;f) \\ \phi_1^*(;f) \\ \phi_2^*(;f) \\ \phi_3^*(;f) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & \frac{5}{18} & \frac{4}{9} & \frac{5}{18} \\ 0 & 0 & 0.03130602.. & \frac{2}{9} & 0.24647176.. \end{pmatrix} \begin{pmatrix} f(0) \\ f(1) \\ f(\frac{1-\sqrt{\frac{3}{5}}}{2}) \\ f(0.5) \\ f(0.88729833..) \end{pmatrix}. \tag{3.8.5}$$

The representation in equation 3.8.1 has been added to FInAT. Each element advertises a `dual_basis` property that returns a FInAT `AbstractPointSet x` representing the points x to sum over and a GEM `ComponentTensor` representing the evaluation tensor Q which is cached to avoid unnecessary recalculation.

An `AbstractPointSet` is the abstract base class for a number of concrete classes which are used to represent sets of points. The most commonly used is the `PointSet` class: it is instantiated with a vector array of points and creates a GEM expression from the array, containing a GEM `Literal` created with the vector itself, which can be used for tensor operations. If the array of points have tensor shape, there is a similar `TensorPointSet` class. Alongside these, an `UnknownPointSet` has been introduced by the author. This is instantiated with a GEM `Variable`, allowing the points to remain a free variable to be supplied to the kernel at run time. For an application of `UnknownPointSet`, see Sect. 4.5.3.

The API for performing dual evaluation of the dual basis of an element `el` is then

```
evaluation, basis_indices = el.dual_evaluation(fn)
```

where `evaluation` is an optimised GEM expression for the contraction 3.8.1, `fn` (representing f) is any Python callable that takes `x` as an argument and returns a GEM tensor `fn(x)` that can be contracted with `Q` along the point indices j and point shape indices $kl...$, and `basis_indices` are the nodal basis indices i. Note that extra free indices may be found in the dual evaluated expression represented by `fn` (typically when the expression contains unknowns represented as UFL `Arguments`) or in the expression for the point set represented by `x` (when the points are to be specified at run-time and therefore do not have a concretely-known `GEM.Literal` expression). These are not modified by this contraction and remain in the GEM `expression` which is used to build an interpolation kernel. Note also that the free indices of the advertised x are indexed from the advertised Q in the `dual_basis` to make the contraction straightforward.

Much of the time the `evaluation` is an `IndexSum` along the indices to be contracted. Where Q is an identity matrix, which occurs for elements whose dual basis is composed entirely from single unweighted point evaluations (i.e. exclusively point evaluation nodes), Q is represented in GEM as a single Kronecker delta.

$$\boldsymbol{Q} = \mathbb{I} \tag{3.8.6}$$

$$\implies Q_{ij} = \delta_{ij}. \tag{3.8.7}$$

Before the GEM expression for the contraction is compiled into a kernel the delta is eliminated, removing the necessity to perform any summation:

$$\begin{aligned}\phi_i^*(;f) &= \sum_j Q_{ij} f(x_j) \\ &= \sum_j \delta_{ij} f(x_j) \\ &= f(x_i).\end{aligned} \tag{3.8.8}$$

When the expression being dual evaluated is tensor valued, further delta elimination is often still possible as is outlined next.

3.8.1 Tensor Finite Elements and Delta Elimination

FInAT tensor finite elements provide an opportunity to speed up dual evaluation. These are wrappers around another, typically scalar, base element which add one or more dimensions of a requested shape to the element basis such that, at any given location x in the element, the function is appropriately tensor valued. If the base element has scalar basis functions $\hat{\phi}_i$ and the requested tensor shape is $(S_1, ..., S_N)$ (i.e. the tensor function is in $\mathbb{R}^{S_1 \times ... \times S_N}$) then the tensor valued basis is

$$\boldsymbol{\phi_{i\alpha_1 ... \alpha_N}} = \hat{\phi}_i \boldsymbol{e_{\alpha_1}} \otimes ... \otimes \boldsymbol{e_{\alpha_N}} \tag{3.8.9}$$

where $\{\boldsymbol{e_{\alpha_1}}\}_{\alpha_1=1}^{S_1}$ are the standard basis vectors of $\mathbb{R}^{S_1}$ and $\{\boldsymbol{e_{\alpha_N}}\}_{\alpha_N=1}^{S_N}$ are the standard basis vectors of $\mathbb{R}^{S_N}$.

It is now necessary to introduce the concepts of `index_shape` and `value_shape` which are properties of GEM tensors:

- `index_shape` is a tuple that describes the total number of the basis functions, allowing efficient iteration over them.
- `value_shape` is a tuple that describes the shape of the basis functions, irrespective of how many there are.

For a scalar basis like that in equation 3.7.1, the `index_shape` is stored as a tuple with one entry:

$$(\text{extent}(i),). \tag{3.8.10}$$

The `value_shape` in that case would be an empty tuple:

$$(). \tag{3.8.11}$$

In equation 3.8.1 there are $\text{extent}(i) \times S_1 \times ... \times S_N$ basis functions. To allow iteration over these the `index_shape` is stored as

$$(\text{extent}(i), S_1, ..., S_N). \tag{3.8.12}$$

The `value_shape` meanwhile are the extra dimensions which are added to each base element by the basis vectors which is stored as

$$(\text{extent}(\alpha_1), ..., \text{extent}(\alpha_N)) = (S_1, ..., S_N). \tag{3.8.13}$$

In certain cases, for example with Raviert Thomas elements, the final base element is *intrinsically non-scalar valued*. Where we construct a tensor finite element from these we still perform an outer product with basis vectors as in equation 3.8.1 but find that the `value_shape`, which *always reflects the shape of the basis functions* (irrespective of how many there are), is not a subset of the `index_shape`, which *always reflects the total number of basis functions.*

Tensor finite elements can also be made from other tensor finite elements. Where we have some pre-existing `value_shape` $(B_1, ..., B_{\hat{N}})$ the introduced tensor shape is appended as $(B_1, ..., B_{\hat{N}}, S_1, ..., S_N)$ to give the final `value_shape`. Here we will stick to the scalar base element case for simplicity.

In general an `index_shape` has structure: it is a tuple rather than a number. This allows efficient iteration over the tensor structure through a performance optimisation known as *delta elimination* which is described later in the section. However, because of this structure, the general dual evaluation expression given in equation 3.8.1 *does not apply here* and is implemented in a separate method for tensor finite elements.

As an aside which aids the comprehension of the vector example later, we note that the scalar components of the basis functions in 3.8.1 can be expressed with Kronecker deltas

$$[\boldsymbol{\phi_{i\alpha_1...\alpha_N}}]_{\gamma_1...\gamma_N} = \delta_{\gamma_1\alpha_1}...\delta_{\gamma_N\alpha_N}\hat{\phi}_i \tag{3.8.14}$$

where indices have been introduced for each tensor dimension: $1 \leq \gamma_1 \leq S_1$ to $1 \leq \gamma_N \leq S_N$. Square brackets for the indexing operation have been included to help differentiate between the indices of the tensor valued basis functions (where `index_shape` is $(\text{extent}(i), \text{extent}(\alpha_1), ..., \text{extent}(\alpha_N))$) and the indexing into the tensors themselves along each axis ($[\cdot]_{\gamma_1...\gamma_N}$).

Returning to the main analysis, a tensor valued expression on a tensor finite element $\boldsymbol{f}(x)$ has scalar components given by

$$f_{\alpha_1...\alpha_N}(x) = \sum_{\beta_1,...,\beta_N} \delta_{\alpha_1\beta_1}...\delta_{\alpha_N\beta_N} f_{\beta_1...\beta_N}(x). \tag{3.8.15}$$

Dual evaluation of $\boldsymbol{f}(x)$ is then

$$\begin{aligned}\phi^*_{i\alpha_1...\alpha_N}(;\boldsymbol{f}(x)) &= \hat{\phi}^*_i(;f_{\alpha_1...\alpha_N}(x)) \\ &= \sum_{\beta_1,...,\beta_N} \hat{\phi}^*_i(;\delta_{\alpha_1\beta_1}...\delta_{\alpha_N\beta_N} f_{\beta_1...\beta_N}(x))\end{aligned} \tag{3.8.16}$$

which implies that the general dual evaluation expression for FInAT tensor finite elements is

$$\begin{aligned}\phi^*_{i\alpha_1\dots\alpha_N}(;\boldsymbol{f}) &= \sum_{j,k,l,\dots,\beta_1,\dots,\beta_N} \hat{Q}_{ijkl\dots}\delta_{\alpha_1\beta_1}\dots\delta_{\alpha_N\beta_N} f_{\beta_1\dots\beta_N}(x_{jkl\dots}) \\ &= \sum_{j,k,l,\dots} \hat{Q}_{ijkl\dots} f_{\alpha_1\dots\alpha_N}(x_{jkl\dots})\end{aligned} \tag{3.8.17}$$

where $\hat{\boldsymbol{Q}}$ is the dual evaluation tensor advertised by the base element.

The advertised $\boldsymbol{Q}$ is the tensor product of the base $\hat{\boldsymbol{Q}}$ with, first, an $S_1 \times S_1$ identity matrix represented by the $\delta_{\alpha_1\beta_1}$, all the way to an $S_N \times S_N$ identity matrix represented by the $\delta_{\alpha_N\beta_N}$:

$$Q_{ijkl\dots\alpha_1\beta_1\dots\alpha_N\beta_N} = \hat{Q}_{ijkl\dots}\delta_{\alpha_1\beta_1}\dots\delta_{\alpha_N\beta_N} \tag{3.8.18}$$

$$\boldsymbol{Q} = \hat{\boldsymbol{Q}} \otimes \mathbb{I}_{\boldsymbol{S_1\times S_1}} \otimes \dots \otimes \mathbb{I}_{\boldsymbol{S_N\times S_N}}. \tag{3.8.19}$$

This $\boldsymbol{Q}$ can only be used with equation 3.8.17, not 3.8.1. The Kronecker deltas simply raise the rank of $\hat{\boldsymbol{Q}}$ as appropriate for contraction with the indexed $\boldsymbol{f}$ so that it can be correctly advertised as the `dual_basis`. Thankfully, since they are symbolic, they can be eliminated before compiling the GEM code to simplify the contraction to that which is given in the second equality of equation 3.8.17, i.e. that the base element dual basis applied to to the correct components of $\boldsymbol{f}$ gives tensor element dual evaluation.

Vector Example

To give a concrete example, in the vector case we have shape $(S_1,)$

$$\boldsymbol{\phi_{i\alpha_1}} = \hat{\phi}_i \boldsymbol{e_{\alpha_1}} \tag{3.8.20}$$

with scalar components given by

$$[\boldsymbol{\phi_{i\alpha_1}}]_{\gamma_1} = \boldsymbol{e}_{\gamma_1} \cdot \boldsymbol{\phi_{i\alpha_1}} = \boldsymbol{e}_{\gamma_1} \cdot \hat{\phi}_i \boldsymbol{e_{\alpha_1}} = \delta_{\gamma_1\alpha_1}\hat{\phi}_i \tag{3.8.21}$$

i.e. there are extent$(i) \times S_1$ basis functions: each scalar function in extent(i) is replicated for each spatial dimension in S_1. Note that the three greek letter indices here α_1, β_1 and γ_1 all have extent S_1.

A vector valued expression $\boldsymbol{f}(x)$ has scalar components

$$f_{\alpha_1}(x) = \sum_{\beta_1} \delta_{\alpha_1\beta_1} f_{\beta_1}(x) \; (= \boldsymbol{e}_{\alpha_1} \cdot \boldsymbol{f}(x)). \tag{3.8.22}$$

Dual evaluation of this is given by

$$\begin{aligned}\phi^*_{i\alpha_1}(;\boldsymbol{f}(x)) &= \hat{\phi}^*_i(;f_{\alpha_1}(x)) \\ &= \sum_{\beta_1} \hat{\phi}^*_i(;\delta_{\alpha_1\beta_1} f_{\beta_1}(x))\end{aligned} \tag{3.8.23}$$

which implies the general dual evaluation expression for FInAT tensor finite elements with vector shape:

$$\begin{aligned}\phi^*_{i\alpha_1}(;\boldsymbol{f}) &= \sum_{j,k,l,\ldots,\beta_1} \hat{Q}_{ijkl\ldots}\delta_{\alpha_1\beta_1} f_{\beta_1}(x_{jkl\ldots}) \\ &= \sum_{j,k,l,\ldots} f_{\alpha_1}(x_{jkl\ldots}).\end{aligned} \tag{3.8.24}$$

The advertised $\boldsymbol{Q}$ can be seen to be the tensor product of the base $\hat{\boldsymbol{Q}}$ with an $S_1 \times S_1$ identity matrix represented by the $\delta_{\alpha_1\beta_1}$:

$$Q_{ijkl\ldots\alpha_1\beta_1} = \hat{Q}_{ijkl\ldots}\delta_{\alpha_1\beta_1} \tag{3.8.25}$$

$$\boldsymbol{Q} = \hat{\boldsymbol{Q}} \otimes \mathbb{I}_{\boldsymbol{S_1}\times\boldsymbol{S_1}}. \tag{3.8.26}$$

As before, this $\boldsymbol{Q}$ can only be used with equation 3.8.17 or 3.8.24, not 3.8.1. Note that we can also now write out the biorthogonality of the primal and dual spaces:

$$\phi^*_{i\gamma_1}(;\boldsymbol{\phi_{k\alpha_1}}) = \hat{\phi}^*_i(;[\boldsymbol{\phi_{k\alpha_1}}]_{\gamma_1}) = \hat{\phi}^*_i;(\delta_{\gamma_1\alpha_1}\hat{\phi}_k) = \delta_{\gamma_1\alpha_1}\hat{\phi}^*_i(;\hat{\phi}_k) = \delta_{\gamma_1\alpha_1}\delta_{ik}. \tag{3.8.27}$$

3.8.2 Tensor Product Finite Elements

FInAT element tabulations, expressed in GEM, can be composed as the tensor product of lower rank elements, the typical example being elements on a quadrilateral as the outer product of two appropriately defined elements on intervals (see figure 3.2).

The GEM compiler and optimisation routines are able to recognise tensor product structure and perform relevant optimisations such as sum-factorisation on the resultant kernels: the loop nests that the tensor contractions give have certain operations lifted from inner to outer loops minimising FLOPs. [14] To illustrate this consider the tensor contraction

$$[\boldsymbol{E}]_{il} = [\boldsymbol{ABC}]_{il} = \sum_j \sum_k A_{ij}B_{jk}C_{kl} \tag{3.8.28}$$

which requires contraction loops over j and k for each i and l. Naively the indexed multiplication operation involving all 3 tensors would be done in the innermost loop

$$\text{extent}(i) \times \text{extent}(l) \times \text{extent}(j) \times \text{extent}(k) \tag{3.8.29}$$

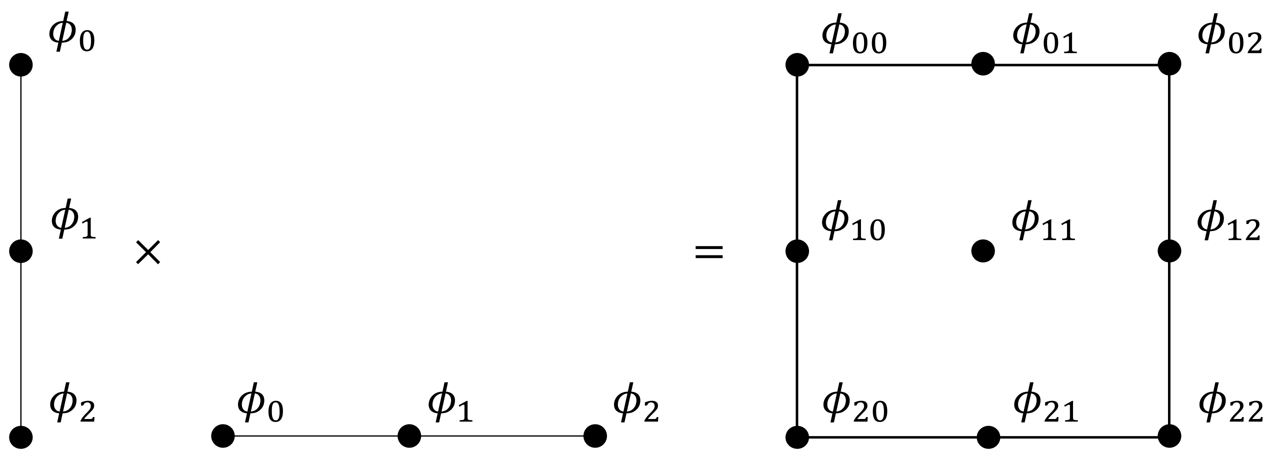


Figure 3.2: A quadrilateral element can be composed from the tensor product of two interval elements. Here the interval elements have 2nd order Lagrange point evaluation nodes as shown in figure 3.1). The extended dual basis is equation 3.2.23. The resulting element is a quadrilateral with 2nd order Lagrange point evaluation nodes.

times in total (see algorithm 1) which scales as

$$\mathcal{O}(p^{2d}) \tag{3.8.30}$$

for tensor depth p and number of contraction indices d.

Algorithm 1 The naive contraction $[\boldsymbol{E}]_{il} = \sum_j \sum_k A_{ij} B_{jk} C_{kl}$.

```
1:  for all $i$ do
2:      for all $l$ do
3:          $E_{il} \leftarrow 0$
4:          for all $j$ do
5:              for all $k$ do
6:                  $E_{il} \leftarrow E_{il} + A_{ij} \times B_{jk} \times C_{kl}$
7:              end for
8:          end for
9:      end for
10: end for
```

$\boldsymbol{A}$ is not involved in the contraction along k so the result of this can be stored in a temporary variable

$$[\boldsymbol{D}]_{jl} = \sum_k B_{jk} C_{kl} \tag{3.8.31}$$

which requires

$$\text{extent}(j) \times \text{extent}(l) \times \text{extent}(k) \tag{3.8.32}$$

indexed multiplication operations. All that remains is to contract $\boldsymbol{A}$ with $\boldsymbol{D}$

$$[\boldsymbol{E}]_{il} = [\boldsymbol{ABC}]_{il} = \sum_j A_{ij} \left(\sum_k B_{jk} C_{kl} \right) = \sum_j A_{ij} D_{jl} \tag{3.8.33}$$

which requires a further

$$\text{extent}(i) \times \text{extent}(l) \times \text{extent}(j) \tag{3.8.34}$$

indexed multiplication operations. The total number of indexed multiplication operations is then reduced to

$$\text{extent}(j) \times \text{extent}(l) \times \text{extent}(k) + \text{extent}(i) \times \text{extent}(l) \times \text{extent}(j) \tag{3.8.35}$$

at the expense of needing to store $\boldsymbol{D}$ (see algorithm 2) which scales as

$$\mathcal{O}(p^{d+1}) \tag{3.8.36}$$

for tensor depth p and number of contraction indices d. It is this 'lifting out' or 'factoring

Algorithm 2 The sum-factorised contraction $[\boldsymbol{E}]_{il} = \sum_j \sum_k A_{ij} B_{jk} C_{kl} = \sum_j A_{ij} D_{jl}$.

```
1:  for all $j$ do
2:      for all $l$ do
3:          $D_{jl} \leftarrow 0$
4:          for all $k$ do
5:              $D_{jl} = D_{jl} + B_{jk} \times C_{kl}$
6:          end for
7:      end for
8:  end for
9:  for all $i$ do
10:     for all $l$ do
11:         $E_{il} \leftarrow 0$
12:         for all $j$ do
13:             $E_{il} \leftarrow E_{il} + A_{ij} \times D_{jl}$
14:         end for
15:     end for
16: end for
```

out' of the unique sums into temporary variables which is referred to as *sum-factorisation* of a tensor contraction.

In general sum-factorisation can be applied to any tensor contraction loops where indices only repeat in particular terms of a larger contraction: a temporary variable is created and the relevant summation is lifted out of the inner loop.

Consider an element with a tensor product structure which is built from lower rank elements of the same shape (the *factors*). The dual evaluations of the factors (from equation 3.8.1) are given by

$$\phi^{*1}_{i_1}(;\boldsymbol{f}) = Q^1_{i_1 j_1 kl\ldots\beta_1\ldots\beta_N} f_{\beta_1\ldots\beta_N}(x^1_{j_1 kl\ldots}) \tag{3.8.37}$$

and

$$\phi^{*2}_{i_2}(;\boldsymbol{f}) = Q^2_{i_2 j_2 kl\ldots\beta_1\ldots\beta_N} f_{\beta_1\ldots\beta_N}(x^2_{j_2 kl\ldots}). \tag{3.8.38}$$

Since we have a tensor product structure, the basis and dual basis are indexed by their factors

$$\phi^*_i(\boldsymbol{f}) = \phi^*_{i_1 i_2}(\boldsymbol{f}). \tag{3.8.39}$$

Similarly, points sets on the tensor product element x are indexed from the factors

$$x_{j_1 j_2 kl\ldots} = x^1_{j_1 kl\ldots} x^2_{j_2 kl\ldots} \tag{3.8.40}$$

and are stored as FInAT `TensorPointSets` constructed directly from the x^1 and x^2. The dual evaluation expression implied by equation 3.8.1 therefore has the form

$$\phi^*_{i_1 i_2}(;\boldsymbol{f}) = Q_{i_1 i_2 j_1 j_2 kl\ldots\beta_1\ldots\beta_N} f_{\beta_1\ldots\beta_N}(x_{j_1 j_2 kl\ldots}) \tag{3.8.41}$$

where we observe that Q is the tensor product of Q^1 and Q^2 from the factors

$$Q_{i_1 i_2 j_1 j_2 kl\ldots} = Q^1_{i_1 j_1 kl\ldots\beta_1\ldots\beta_N} Q^2_{i_2 j_2 kl\ldots\beta_1\ldots\beta_N} \tag{3.8.42}$$

$$\boldsymbol{Q} = \boldsymbol{Q^1} \otimes \boldsymbol{Q^2}. \tag{3.8.43}$$

The full dual evaluation expression is then

$$\phi^*_{i_1 i_2}(;\boldsymbol{f}) = Q^1_{i_1 j_1 kl\ldots\beta_1\ldots\beta_N} Q^2_{i_2 j_2 kl\ldots\beta_1\ldots\beta_N} f_{\beta_1\ldots\beta_N}(x^1_{j_1 kl\ldots} x^2_{j_2 kl\ldots}). \tag{3.8.44}$$

Since there are unique contraction indices i_1 and j_1 in Q^1 and x^1, as well as i_2 and j_2 in Q^2 and x^2, sum-factorisation can be applied to this contraction similarly to in equation 3.8.33.

Note that each point set and expression tensor could already have been composed as products of factors: the number of indices in the contraction simply increases in this case. The expression $\boldsymbol{f}$ which is dual evaluated may also have some tensor-product structure (i.e. be composed as $f_{\gamma\delta} = f^1_\gamma f^2_\delta$) which allows further sum-factorisation speed-ups.

Because this is all expressed in GEM, dual evaluations expressed in this way can have their resultant expression kernels optimised by TSFC as usual to improve performance. The result is a drastic improvement in element-local kernel FLOP count which, given a polynomial degree p (which is proportional to the maximum extent of $\boldsymbol{Q}$) and spatial dimension d (which is proportional to the number of contraction indices), scales as $\mathcal{O}(p^{d+1})$ rather than $\mathcal{O}(p^{2d})$ because of the ability to apply sum-factorisation to the Q and x contractions. A test has been added to TSFC which ensures that this is the case[10].

3.8.3 Future Improvements

At present $\boldsymbol{Q}$ is a dense tensor but, as the example in 3.8.5 demonstrates, it is generally sparse even after delta elimination. Additional performance enhancements can therefore be reaped by skipping the zero multiplications in the contraction: this will require a new node in GEM's Abstract Syntax Tree such as `GEM.SparseLiteral` for tensors like $\boldsymbol{Q}$ and a new contraction node such as `GEM.SparseContraction` for the contraction with a second dense tensor (general sparse-sparse contraction is much more difficult to build loops for than sparse-dense contraction and is not necessary here).

The $\boldsymbol{Q}$ tensor has been implemented for most cases where there is a well-defined dual evaluation operation. It has not yet been implemented for so-called *enriched elements*: given two elements with local basis functions $\{\phi_i\}_{i=0}^{k}$ and $\{\phi_j\}_{j=0}^{l}$, the enriched element created from them has a basis

$$\{\phi_i\}_{i=0}^{k} \cup \{\phi_j\}_{j=0}^{l} \tag{3.8.45}$$

For dual evaluation to produce something sensible with such elements, we need the opposing sets of basis and dual basis functions to be orthogonal to one another to avoid unexpected cross terms in the interpolation:

$$\phi_i^*(;\psi_j) = 0 \text{ and} \tag{3.8.46}$$

$$\psi_i^*(;\phi_j) = 0 \;\forall\, i, j. \tag{3.8.47}$$

This will only be true in special cases.

Firedrake uses enriched elements to define H(div) and H(curl) elements on quadrilaterals and hexahedra (for example 'Raviart–Thomas cubical H(curl)' and 'Raviert Thomas Cubical H(div)'[11]). These are, respectively, spaces where the normal or tangent components of a vector-valued function are continuous along cell edges. This is often a requirement for functions with conservation requirements and are often used for modelling fluids. The reasoning is laid out in McRae et al. [45]. In this case the sets of basis functions are orthogonal to each other so it makes sense to define it.

This will likely involve concatenating the $\boldsymbol{Q}$ tensors of the underlying elements along an appropriate axis, though this has not been investigated fully. Such a concatenation would also imply some ordering of the union - this would have to be explicitly checked against the element definitions. For triangular and tetrahedral meshes, H(div) and H(curl) conforming elements are not instances of Enriched Element in Firedrake and do have a $\boldsymbol{Q}$ tensor defined. For more

[10]See https://github.com/firedrakeproject/tsfc/blob/351994d0ba192b4cb53692ca98d552f9859b5283/tests/test_interpolation_factorisation.py. Note this test was written by Lawrence Mitchell.

[11]see the quadrilateral variants at https://defelement.com/elements/nedelec1.html and https://defelement.com/elements/qdiv.html respectively

on Enriched Elements and H(div) and H(curl) spaces see McRae et al. [45]

Chapter 4

A New Abstraction for Point Evaluation: Interpolation onto Vertex-Only Meshes

This chapter adapts and reuses, without further attribution, text and diagrams from Nixon-Hill et al. [3] which were created by the author.

4.1 Motivation

Point data are sets of values which are defined at particular locations in space, possibly over some period of time. These are found everywhere. Take the example of geosciences: The air temperature at a weather station, ice sheet elevation measurements from satellite altimetry and ocean salinity and temperature from drifting buoys are all examples of point data that may be measured from or assimilated into a geoscientific model. Models themselves contain point data. Since finite element functions are defined everywhere, we can request a value of the function anywhere we like. These can be used in PDEs or for coupling models together.

Point data manipulations are conspicuously absent from UFL: there is no operation which takes a UFL expression, such as a `Function`[1] on some mesh, and returns a UFL expression which is equivalent to the point evaluation of that expression. If there were, we could create expressions involving points (Dirac deltas for example). Prior to this work, such terms were typically approximated in Firedrake as narrow Gaussians (so-called 'bump functions').

The assimilation of point data into PDE models in Firedrake is a good example: UFL and Firedrake operations can be *taped* with the automated adjoint generation tool pyadjoint [46] which is integrated with Firedrake. If UFL has point data as part of its type system, such problems can be solved. For more on these see chapters 5 and 6. The addition of point data to UFL is therefore useful for Firedrake (and is a much requested feature) and adds a concept necessary to define any statistics which have no spatial extent.

Whilst there are perhaps many ways that point data could be introduced to UFL, ideally we would come up with a formulation which is sufficiently general and abstract that it can be applied to other finite element libraries. This would make this work significantly more useful.

Point evaluation in Firedrake was, prior to this work, limited to point wise data output of Firedrake `Functions` via a simple `Function.at` method which reports the value of the function at the specified point(s). As well as not returning a UFL expression, this has many problems. It bypasses Firedrake's code generation pathway (typically accessed by compiling or assembling UFL forms), it does not work well in parallel (the same point must be specified on all MPI processes and, if there's a disagreement about which cell owns the point, the method will fail), and does not provide exact solutions for all finite elements.

[1]`Function` is a subclass of UFL `Coefficient`, it represents finite element function data in the Firedrake ecosystem

4.2 Existing Work

No fully generalisable approach for interacting with point data exists in current finite element libraries to the author's knowledge.

Libraries which do not tie themselves to a mathematical framework like UFL are able to deal with point data by hard coding point evaluations and point forcing terms. For example, deal.II [33] contains functionality for dealing with point sources via a `create_point_source_vector` function, and it has a particles class for dealing with large numbers of points. MFEM [31, 32] has a `DeltaCoefficient` type for representing Dirac deltas. Linking these together, and integrating them with the rest of the library, is left to the user.

Legacy FEniCS [17] and its rewrite FEniCSx, by far the most similar libraries to Firedrake, are bound to UFL and therefore have no symbolic way of dealing with point data. Point evaluation must be done manually using low level cell-search methods which are not inherantly parallel safe (see discussion in Sect. 7.5).

DUNE-PDELab [41] and DUNE-fem [18] do not have any documented ability to interact with point data outside of low-level basis-function point evaluation routines. MOOSE [25] has a `PointValueSampler` class for sampling points and application-specific point source and sink terms that can be inserted into compatible PDEs[2]. Elmer-FEM [23] has no documented point evaluation capability but does provide PDE specific point source and sink terms. Finch [20], FREEFEM [21], and GOMA [24] have no documented capability at all.

Where point data interaction are not available, one is forced to make approximations such as the 'bump functions' mentioned in the previous section or an extrapolation of a sampled field to the whole domain. The consequences of such approximations are discussed in chapter 6.

More approaches to dealing with point data are discussed in chapter 7, for example in Sect. 7.3.

4.3 Point data as finite element functions

We can integrate point data with the paradigm of finite element functions being members of finite element function spaces defined over meshes (see Sect. 1.2) by looking carefully at what we mean by point data. Point data can be separated into two parts: (a) the locations $\{X_i\}$ of the N data points at a given time (a point cloud) and (b) the N values $\{y_i(X_i)\}_{i=0}^{N-1}$ associated with the point cloud (see Fig. 4.1). Finite element functions have a similar distinction: (a) the discretised shape of the domain of interest (the mesh Ω) and (b) the values associated with that mesh (the weights applied to the basis functions).

Applying the finite element distinction to the locations of the data (a) suggests a 'point

[2]see https://mooseframework.inl.gov/modules/porous_flow/sinks.html

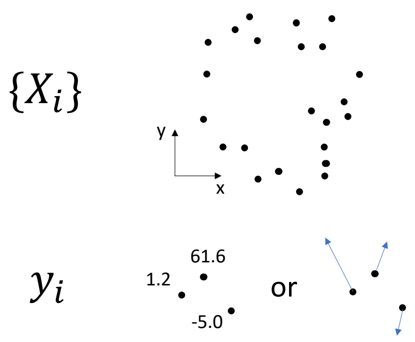


Figure 4.1: Point data consist of (a) a set of N spatial coordinates $\{X_i\}_{i=0}^{N-1}$ (a 'point cloud') and (b) the scalar, vector or (not shown) tensor values $\{y_i(X_i)\}_{i=0}^{N-1}$ at those coordinates. Maintaining this distinction is key when trying to form a rigorous way of handling point data for any numerical method, such as the finite element method, which separates the idea of domain and values on the domain.

cloud mesh' formed of N disconnected vertices at each location X_i:

$$\Omega_v = \{X_i\}_{i=0}^{N-1}. \tag{4.3.1}$$

We refer to this as a 'vertex-only mesh'. This is an unusual mesh: a vertex has no extent (it is topologically zero dimensional) but exists at each location X_i in a space of geometric dimension $\dim(X_i)$. Fortunately meshes with topological dimension less than their geometric dimension are not unusual: 2D meshes of the surface of a sphere in 3D space are commonly used to represent the surface of the earth. Such domains are typically called 'immersed manifolds'. Disconnected meshes are also not unheard of: the software responsible merely needs to be able to iterate over all the cells of the mesh. In this case each cell is a vertex X_i. We can therefore legitimately construct such a mesh.

We now need to consider the values $\{y_i(X_i)\}_{i=0}^{N-1}$ (b). Only one value, be it scalar, vector or tensor, can be given to each cell (i.e. each point or vertex). Fortunately a finite element function space for this case exists: the space of zero order discontinuous Lagrange polynomials

$$y \in \mathrm{P0DG}(\Omega_v) \tag{4.3.2}$$

where P0DG stands for Polynomial degree 0 Discontinuous Galerkin.[3] Here y is a single discontinuous function which contains all of our point data values at all of our point locations (mesh vertices)

$$y(x) = \begin{cases} y(X_i) & \text{if } x = X_i, \\ \text{undefined} & \text{elsewhere.} \end{cases} \tag{4.3.3}$$

[3]For more on why this function space was deemed appropriate, see Appendix A.1.1, specifically 'Notes on the Choice of Function Space and Integration Behaviour'.

Integrating this over our vertex-only mesh Ω_v gives

$$\int_{\Omega_v} y(x)dx = \sum_{i=0}^{N-1} y(X_i) \quad \forall\, y \in \text{P0DG}(\Omega_v) \tag{4.3.4}$$

where dx is a sum of Dirac measures $\mathrm{d}\delta_{X_i}$ at each vertex X_i (a discrete measure with unity weightings).[4] A fuller analysis of vertex-only meshes and functions on them can be found in the appendix to this chapter (Sect. A.1.1).

This definition lets us directly reason about point data in finite element language and yield useful results. Recall that, in the finite element method, we approximate fields on domains with finite element functions (Sect. 1.2). So long as the locations of our vertices X_i of our vertex-only mesh Ω_v are within the domain of our 'parent' mesh

$$\Omega_v \subseteq \Omega \tag{4.3.5}$$

then we can go from some function u in some finite element function space defined on our parent mesh

$$u \in \text{FS}(\Omega) \tag{4.3.6}$$

to one defined on our vertex-only mesh

$$u_v \in \text{P0DG}(\Omega_v) \tag{4.3.7}$$

by performing point evaluations at each vertex location $u(X_i)\ \ \forall\ i$.

The operator for this can be formulated as dual evaluation interpolation into P0DG, i.e.

$$\mathcal{I}_{\text{P0DG}(\Omega_v)} : \text{FS}(\Omega) \to \text{P0DG}(\Omega_v) \tag{4.3.8}$$

such that

$$\mathcal{I}_{\text{P0DG}(\Omega_v)}(; u) = u_v. \tag{4.3.9}$$

This operator is linear in u which we denote by a semicolon before the argument. The construction of this operator is described in the next section.

4.4 Point evaluation as dual evaluation interpolation

For $u_v = \mathcal{I}_{\text{P0DG}(\Omega_v)}(; u)$ where $u(x) \in \text{FS}(\Omega)$ we require, at each vertex cell X_i of our vertex-only mesh Ω_v, the point evaluation $u(X_i)$. The global dual bases for global interpolation (Eq.

[4]This measure is also discussed further in Appendix A.1.1, 'Notes on the Choice of Function Space and Integration Behaviour'.

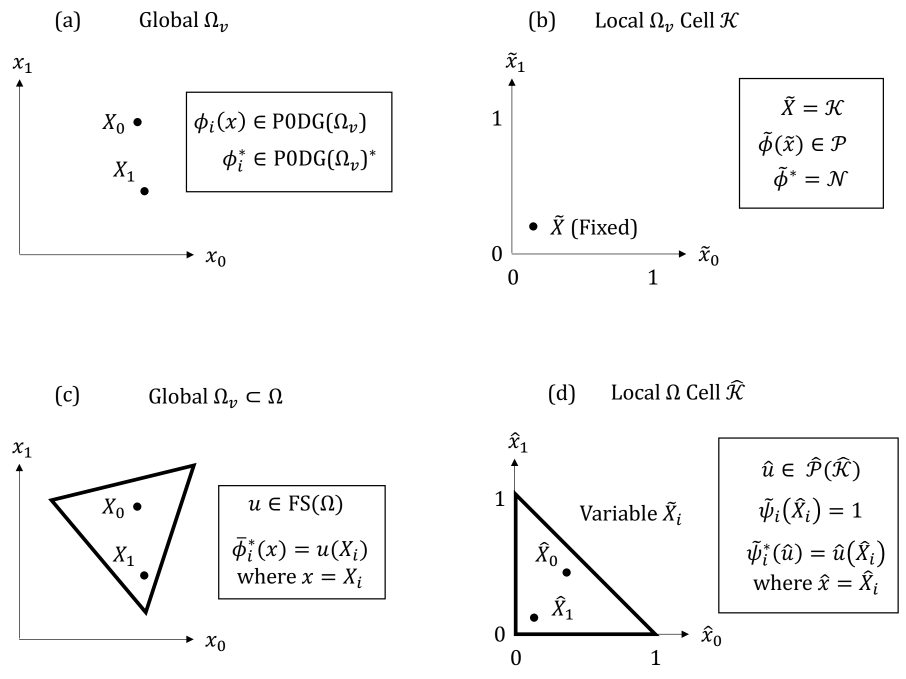

(a) Global Ω_v
x_1
x_0
X_0
X_1
$\phi_i(x) \in \text{P0DG}(\Omega_v)$
$\phi_i^* \in \text{P0DG}(\Omega_v)^*$
(b) Local Ω_v Cell $\mathcal{K}$
$\tilde{x}_1$
$\tilde{x}_0$
1
0
0
1
$\tilde{X}$ (Fixed)
$\tilde{X} = \mathcal{K}$
$\tilde{\phi}(\tilde{x}) \in \mathcal{P}$
$\tilde{\phi}^* = \mathcal{N}$
(c) Global $\Omega_v \subset \Omega$
x_1
x_0
X_0
X_1
$u \in \text{FS}(\Omega)$
$\bar{\phi}_i^*(x) = u(X_i)$
where $x = X_i$
(d) Local Ω Cell $\hat{\mathcal{K}}$
$\hat{x}_1$
$\hat{x}_0$
1
0
0
1
Variable $\hat{X}_i$
$\hat{X}_0$
$\hat{X}_1$
$\hat{u} \in \hat{\mathcal{P}}(\hat{\mathcal{K}})$
$\tilde{\psi}_i(\hat{X}_i) = 1$
$\tilde{\psi}_i^*(\hat{u}) = \hat{u}(\hat{X}_i)$
where $\hat{x} = \hat{X}_i$

Figure 4.2: A summary of where the various functions and functionals described in this section are defined. (a) is the globally defined vertex-only mesh, upon basis functions ϕ_i and dual basis functions ϕ_i^* are defined. In the scalar case there is one for each vertex. (b) is the local cell of the vertex-only mesh $\mathcal{K}$ with a fixed local vertex $\tilde{X}$ and, in the scalar case, a single local basis function $\tilde{\phi}$ and functional $\tilde{\phi}^*$ from which the global function space P0DG(Ω_v) is built. In (c) we introduce a parent mesh Ω with some function $u \in \text{FS}(\Omega)$ that we want to point evaluate at each vertex. The global dual bases for global interpolation $\bar{\phi}_i^*$ which perform this for each vertex X_i, giving the coefficients to ϕ_i, ought to be able to operate on u. Rather than attempting to calculate this on the reference cell of the vertex-only mesh, we do so on the reference cell of the parent mesh $\hat{\mathcal{K}}$. (d) shows this reference cell where the local function $\hat{u}$ is defined: here $\hat{\mathcal{P}}$ is the local function space defined on $\hat{\mathcal{K}}$. We introduce new basis functions $\tilde{\psi}_i$ and dual basis functionals $\tilde{\psi}_i^{\ *}$ each defined by a newly varying $\hat{X}_i$ in $\hat{\mathcal{K}}$, where the functional performs the point evaluation. When the coordinate transform back to global space (c) is performed, each of these corresponds to $\bar{\phi}_i^*$.

3.2.15) are then

$$\bar{\phi}_i^*(;u) = u(X_i) \tag{4.4.1}$$
$$= \delta_{X_i}(;u) \tag{4.4.2}$$

where δ_{X_i} is the Dirac delta *functional* (as opposed to the Dirac delta that is placed inside an integrand which often appears in physics contexts) that produces the evaluation of u at X_i.[5]

Looking at the local definition of interpolation (Eq. 3.2.2) we require, for each X_i, a single local dual basis functional ($k = 1$ in Eq. 3.2.2) to perform the point evaluation at each vertex cell using a fixed reference cell. This would have us transforming u on a particular vertex cell to some $\tilde{u}$ on our reference vertex such that our local dual basis functional is

$$\tilde{\phi}^*(;\tilde{u}) = \tilde{u}(\tilde{X}) \tag{4.4.5}$$

This is the single dual basis functional that can be used in the Ciarlet triple formulation [34] of the P0DG function space on a vertex-only mesh:

$$(\mathcal{K}, \mathcal{P}, \mathcal{N}) \tag{4.4.6}$$

where $\mathcal{K} = \tilde{X}$ is a reference vertex and we have one node $\mathcal{N} = \tilde{\phi}^*$ which is the single dual basis functional to $\mathcal{P}^*$, the dual to our local function space $\mathcal{P}$. This gives a local basis function of $\mathcal{P}$[6]

$$\tilde{\phi}(\tilde{x}) = \begin{cases} 1 & \text{where } \tilde{x} = \tilde{X}, \\ 0 & \text{otherwise.}^{7} \end{cases} \tag{4.4.8}$$

The Ciarlet definition of the finite element P0DG(Ω_v) is derived step-by-step in the appendix, Sect. A.1.1.

[5]For vector or tensor valued function spaces $\boldsymbol{u} \in \mathbb{R}^{S_1 \times \ldots \times S_N}$ we follow the notation of chapter 3 Sect. 3.7 where $\{\boldsymbol{e}_{\boldsymbol{\alpha_1}}\}_{\alpha_1=1}^{S_1}$ are the standard basis vectors of $\mathbb{R}^{S_1}$ and $\{\boldsymbol{e}_{\boldsymbol{\alpha_N}}\}_{\alpha_N=1}^{S_N}$ are the standard basis vectors of $\mathbb{R}^{S_N}$. Here we have dual basis functions for each for each dimension

$$\phi^*_{i\alpha_1 \ldots \alpha_N}(;\boldsymbol{u}) = \boldsymbol{u}(X_i) \cdot \boldsymbol{e}_{\boldsymbol{\alpha_1}} \otimes \ldots \otimes \boldsymbol{e}_{\boldsymbol{\alpha_N}} \tag{4.4.3}$$
$$= \delta_{X_i}(;\boldsymbol{u}) \cdot \boldsymbol{e}_{\boldsymbol{\alpha_1}} \otimes \ldots \otimes \boldsymbol{e}_{\boldsymbol{\alpha_N}}. \tag{4.4.4}$$

[6]For vector or tensor valued function spaces, we have a local basis function for each local Cartesian basis vector or tensor respectively

$$\tilde{\boldsymbol{\phi}}_{\boldsymbol{\alpha_1} \ldots \boldsymbol{\alpha_N}}(\tilde{x}) = \begin{cases} \tilde{\boldsymbol{e}}_{\boldsymbol{\alpha_1}} \otimes \ldots \otimes \tilde{\boldsymbol{e}}_{\boldsymbol{\alpha_N}} & \text{where } \tilde{x} = \tilde{X}, \\ 0 & \text{otherwise.} \end{cases} \tag{4.4.7}$$

[7]This second case is never realised since $\tilde{\phi}$ only takes on a value at the reference vertex $\tilde{X}$, but it follows from the definition of $\tilde{\phi}^*$ via the Ciarlet triple formulation and allows us to build up global basis functions which have more than one vertex.

We can equivalently let the reference vertex, and therefore our functionals, vary for each X_i

$$\tilde{\psi}_i^*(;\hat{u}) = \hat{u}(\hat{X}_i) \tag{4.4.9}$$

where $\hat{u}$ is our locally defined function on the reference cell of our *parent* mesh Ω and $\hat{x}$ are the reference coordinates in that cell. Now for each vertex X_i in Ω_v, $\hat{X}_i$ is its equivalent location in the reference cell of the parent mesh Ω. This gives us a local basis function

$$\tilde{\psi}_i(\hat{x}) = \begin{cases} 1 & \text{where } \hat{x} = \hat{X}_i, \\ 0 & \text{otherwise.} \end{cases} \tag{4.4.10}$$

This is equivalent to $\tilde{\phi}(\tilde{x})$ since, after transforming back to global coordinates, they both equal 1 at X_i.[8] Figure 4.2 is a summary diagram showing where these various functions and functionals are defined.

The global dual evaluation interpolation operator then requires the following for each vertex cell X_i in our vertex-only mesh Ω_v:

1. finding the cell of the parent mesh Ω that X_i resides in,
2. finding the equivalent reference coordinate $\hat{X}_i$ in that cell,
3. transforming our function u to the parent mesh reference cell giving $\hat{u}$,
4. performing the point evaluation $\hat{u}(\hat{X}_i)$ and
5. transforming the result back to global coordinates giving $u(X_i)$.

This formalises the process of point evaluation with everything remaining a finite element function defined on a mesh. These functions can have concrete values or be symbolic unknowns. If the symbolic unknown is a point, we can now express that in the language of finite elements. For example

$$\int_{\Omega_v} \mathcal{I}_{\mathrm{P0DG}(\Omega_v)}(;f(x))dx = \sum_{i=0}^{N-1} f(x_i) = \sum_{i=0}^{N-1} \int_{\Omega} f(x)\delta(x-x_i)dx. \tag{4.4.11}$$

Given the discussion in chapter 6, note here that what we call an 'interpolation' operation is exact and, excepting finite element function spaces with discontinuities, unique: we get the value of u at the points $\{X_i\}$ on the new mesh Ω_v. We are able to perform this exact interpolation because u is a function which has a value everywhere.

Note that by picking a reference point $\hat{X}_i$ in a unique cell for each X_i we allow interpolation from discontinuous Galerkin finite element function spaces, even though point evaluation is not well defined on the cell boundaries (the nature of the method implies that a function value

[8]The vector/tensor case is similarly equivalent.

in one cell at the boundary will be different to the value on the adjoining cell). Those who use such spaces will probably still want to be able to point evaluate so point evaluation is still allowed: points exactly at cell boundaries are chosen to belong in one the cells they border.[9]

4.5 Firedrake implementation

4.5.1 Vertex-only mesh

Prior to this work, all meshes in Firedrake required an underlying PETSc [9, 10] DMPlex to represent the topology of the mesh as detailed in Lange et al. [47]. DMPlex is a data management structure for representing unstructured grids and meshes: the connectivity of vertices, lines, planes and volumes are represented as a Directed Acyclic Graph (DAG).

Since we have no connectivity to consider we can instead use PETSc's implementation of point clouds, the DMSwarm data structure, to represent the mesh topology. DMSwarm and DMPlex are both subclasses of an abstract DM class for representing grids and meshes: this lets us swap out DMPlex for DMSwarm and build data structures on top of it in exactly the way Lange et al. describe[10]. Details of DMPlex, DMSwarm, and DM can be found in the PETSc manual [10]. More details on the specific implementation here can be found in Sect. A.1.2 in the appendix.

The Firedrake implementation is called `VertexOnlyMesh` which, at construction, takes a list of coordinates and a 'parent' mesh to be immersed in. The coordinates are stored on the DMSwarm in a 'field' which associates each DMSwarm entity (topological vertex) with a piece of data. The DMSwarm is also directly linked to the parent mesh DM using in-built PETSc tools.

We then search for each coordinate within the parent mesh and identify the parent mesh cell (step 1 in Sect. 4.4) alongside the reference coordinate $\hat{X}_i$ in that cell (step 2). These are also stored in DMSwarm 'fields' to associate them with each vertex. The cell search algorithm is convered in detail in chapter 7 (Sect. 7.4).

The map from local coordinates $\hat{x}$ to global coordinates x

$$x = G(\hat{x}) \tag{4.5.1}$$

[9]See the description of the cell location and voting algorithms in chapter 7 for how this is implemented in Firedrake.

[10]Whilst the swap of DMPlex for DMSwarm may not be strictly necessary, it aided the development of our implementation thanks to special DMSwarm features such as the ability to associate itself with another DM. A future reimplementation could perhaps swap DMSwarm back to DMPlex.

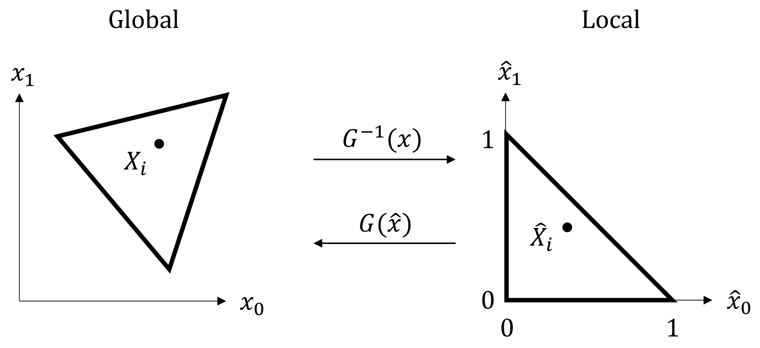


Figure 4.3: G and its inverse, G^{-1} map between local and global coordinate systems, respectively. Here the map between a reference triangle, as used by Firedrake, and a triangle on a mesh in global coordinates is shown with the corresponding point locations $\hat{X}_i$ and X_i.

is known a-priori for a given mesh.[11] We find the inverse

$$\hat{x} = G^{-1}(x) \tag{4.5.4}$$

for each X_i from G using Newton's method with initial guess $\hat{X}_i^0$ at the centre of the reference cell.[12] This was pre-existing Firedrake functionality. See Fig. 4.3 for an illustration.

DMSwarm natively supports embedding in a DMPlex so DMPlex's cell numberings, which differ from Firedrake's, are similarly stored at this point in a pre-existing DMSwarm field.[13] A full list of all the fields stored on the DMSwarm can be found in the appendix to chapter 7 (Sect. A.3.2).

[11]The mesh coordinates are stored as a vector valued finite element function on the mesh such that

$$x = \sum_{i=0}^{\dim(\mathrm{FS}(\Omega))-1} x_i \psi_i(x) \tag{4.5.2}$$

where ψ_i are the vector valued basis functions and x_i are global coordinate coefficients ('locations' of the basis functions). Most meshes have the finite element function as degree 1 continuous polynomials $\mathrm{FS}(\Omega) = \mathrm{P1CG}(\Omega)$ in which case x_i are the coordinates of the vertices of the mesh. The vertex-only mesh has its coordinate function as a vector valued $\mathrm{P0DG}(\Omega_v)$ space - each x_i here are also the vertex coordinates. Meshes without straight edges can be represented with higher order coordinate functions: these are called *high-order meshes*. The global basis functions are derived from a reference-cell-local basis $\tilde{\psi}_i \in \mathcal{P}$ where $\mathcal{P}$ is our smaller reference-cell-local function space. We weight these by our global coordinates to get the mapping

$$G(\hat{x}) = \sum_{i=0}^{\dim(\mathcal{P})-1} x_i \tilde{\psi}_i(\hat{x}) = x. \tag{4.5.3}$$

[12]Where the transformation from global to reference coordinates is affine, as is often the case for non-high-order meshes, this converges in one iteration.

[13]The two cell numberings do not have a one-to-one mapping in the case of extruded meshes [48], instead both base mesh DMPlex cell and extrusion height are saved.

4.5.2 Making a finite element function space on a vertex-only mesh

The construction of a finite element function space, in Firedrake called a `FunctionSpace`, proceeds exactly as shown in Fig. 2 of [47] but with DMPlex now a DMSwarm. Only minor modifications had to be made to the Firedrake software stack to allow for creation of the P0DG(Ω_v) finite element function space. These generally involved making sure that a reference cell which was a point would behave as expected since no use for this had previously been considered. All other finite element function spaces are disallowed on a vertex-only mesh since they require cells which have some extent.

Vector and tensor valued equivalents, needed to store point evaluations from vector or tensor valued finite element function spaces on parent meshes, are created from the scalar P0DG(Ω_v) finite element function space. The reference FInAT element is a wrapper around the scalar element, as described in Sect. 3.8.1, such that, at any given location x in the element, the function is appropriately vector or tensor valued. The full finite element function space is built up from the reference FInAT elements, so is also appropriately vector or tensor valued.

4.5.3 Point evaluation operation

Firedrake's implementation of global dual evaluation interpolation was described in Sect. 3.4.1. So long as the high level API of interpolation and its adjoint is maintained, it can be taped by pyadjoint and can be used for PDE constrained optimisation problems such as data assimilation. Up to this point, only dual evaluation interpolations between finite element function spaces on the same mesh have been considered. Here we are interpolating across meshes: from the parent mesh Ω to the vertex-only mesh Ω_v.

PyOP2 can loop over mesh cells, but which mesh should it loop over? If it were to loop over the parent mesh Ω, we would need to identify any vertices of vertex-only mesh Ω_v inside it, then perform the necessary dual evaluations in each case. Of course there will then be cases where some cells of Ω contain no vertex cells of Ω_v, leading to loops where no calculations are performed. Conversely, where there are large numbers of vertex cells of Ω_v in each cell of Ω, it may make sense to gather all the vertex locations and have TSFC build a kernel which can assign values to however many vertices of Ω_v there are. Since we have already saved the parent cell number and parent mesh reference cell location $\hat{X}_i$ on each vertex X_i of Ω_v, the most straightforward option is to instead loop over its vertex cells.

Having made this decision, Firedrake needs to (a) construct a dual evaluation kernel, (b) assign data from that kernel to the P0DG(Ω_v) finite element function space and (c) appropriately loop over the vertex-only mesh cells X_i.

For step (a), we use TSFC: as described Sect. 3.4.1 it is responsible for transforming to and from reference cells but does not play a part in assigning data to or from global function spaces (that is PyOP2's role). The kernel therefore need not know about differing meshes: from its point of view, the dual evaluation operation only involves the parent mesh: the target space has a single local dual basis functional[14]

$$\tilde{\psi}_i^*(;\hat{u}) = \hat{u}(\hat{X}_i) \tag{4.5.5}$$

(from Eq. 4.4.9) and the kernel it compiles will produce a value corresponding to that.

For step (c), equation 4.5.5 is executed for every point in the vertex-only mesh by PyOP2. The implementation details of this are in the appendix, Sect. A.1.3.

Providing the correct FInAT element: runtime tabulation

Inside TSFC we need to supply an appropriate FInAT element for point evaluation on the reference cell with the dual basis functional

$$\tilde{\psi}_i^*(;\hat{u}) = \hat{u}(\hat{X}_i) \tag{4.5.6}$$

from Eq. 4.4.9. $\hat{X}_i$ will vary for each reference cell. Specifying a new FInAT element for each point X_i is not feasible: Our kernel needs to be generalisable to any $\hat{X}_i$ we supply. We store $\hat{X}_i$ for every vertex-only mesh cell X_i, all we need then is for $\hat{X}_i$ to be an argument to the kernel.

Evaluation of basis functions at an arbitrary point on a reference element is referred to as 'runtime tabulation'. 'Tabulation' is the operation of evaluating local basis functions at one or more points and 'runtime' implies that this is performed after the generation of the kernel (i.e. the point is an argument supplied to the kernel). TSFC can automatically create kernels with runtime specified arguments by creating GEM `Variables` that correspond to them (see Sect. 3.5).

FInAT and FIAT already have an element which performs arbitrary point evaluations on the reference cell called a `QuadratureElement`. These are intended for experimentation with arbitrary quadrature rules for integration: one supplies them with a quadrature rule which are a set of weights w_i and locations on the reference cell x_i to sum. The dual basis of the element are defined to be the evaluations at the points

$$\tilde{\psi}_i^*(;\tilde{f}) = \tilde{f}(x_i).^{15} \tag{4.5.7}$$

For our purposes, we need a quadrature rule with a single point in the reference cell (the weight is inconsequential) the location of which we specify at runtime. For details see the appendix, Sect. A.1.4.

The kernel is then compiled in Firedrake as follows:

1. Identify if our interpolation target is P0DG(Ω_v).

[14]or one for each tensor dimension

[15]The $\boldsymbol{Q}$ tensor for FInAT dual evaluation is identity in this case: see Eq. 3.8.7.

2. Change this to an equivalent `QuadratureElement` on the reference cell of the parent mesh built from a FInAT quadrature rule with a single runtime-specified point. [16]

3. Supply the quadrature element and the parent mesh (upon which the expression to interpolate is found) to the TSFC dual evaluation kernel builder.

4. Receive a dual evaluation kernel which has a point on the reference cell to dual evaluate at as an argument to the kernel.

This can be executed once for each X_i with $\hat{X}_i$ as the point evaluate at.

4.6 Demonstration

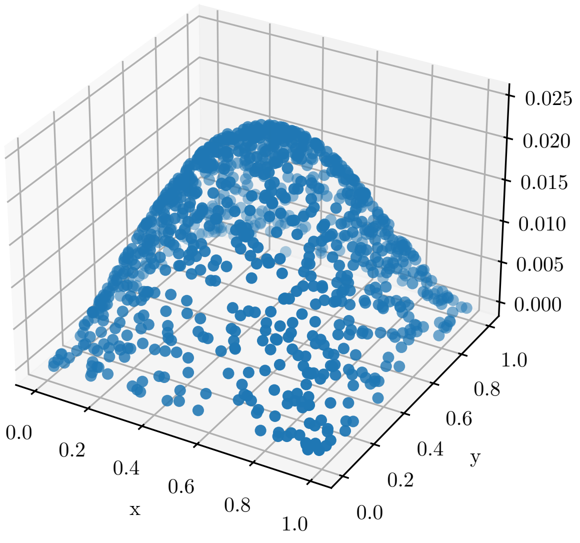


Figure 4.4: 1000 samples at randomly generated coordinates of the solution to the Poisson equation from Eq. 1.3.2, generated with the code in Listing 6.

Listing 6 demonstrates the new Firedrake functionality. The Poisson equation we solved in Sect. 1.3 (Eq. 1.3.2, Listing 2 with forcing function[17] and solution plotted in Fig. 1.3) is sampled at a list of coordinates by creating a vertex-only mesh at those coordinates. A P0DG finite element function space on the vertex-only mesh is created and the solution u to Poisson's equation interpolated onto it. A plot of the resulting function is shown in Fig. 4.4.

[16]If we are dual evaluation interpolating from a vector or tensor valued finite element function space we must be dual evaluation interpolating into a vector or tensor valued finite element function space built from the scalar P0DG(Ω_v) space. These kinds of dual evaluation operations are described in Sect. 3.8.1. The equivalent `QuadratureElement` in these cases is identically built from the scalar `QuadratureElement` which will perform a scalar point evaluation for each dimension.

[17]Note: as for the previous example of solving Poisson's equation (Listing 2) the random forcing term `f` here is not mesh independent so adaptation of the example should be approached with caution.

```python
from firedrake import *

def poisson_point_eval(coords):
    """Solve Poisson's equation on a unit square for a random forcing term
    with Firedrake and evaluate at a user-specified set of point coordinates.

    Parameters
    ----------
    coords: numpy.ndarray
        A point coordinates array of shape (N, 2) to evaluate the solution at.

    Returns
    -------
    firedrake.function.Function
        A finite element function containing the point evaluatations.
    """
    omega = UnitSquareMesh(20, 20)
    P2CG = FunctionSpace(omega, family="CG", degree=2)
    u = TrialFunction(P2CG)
    v = TestFunction(P2CG)

    # Random forcing Function with values in [1, 2].
    f = RandomGenerator(PCG64(seed=0)).beta(P2CG, 1.0, 2.0)

    a = inner(grad(u), grad(v)) * dx
    L = f * v * dx
    bc = DirichletBC(P2CG, 0, "on_boundary")
    u_sol = Function(P2CG)  # solution will be stored here
    solve(a == L, u_sol, bc)

    omega_v = VertexOnlyMesh(omega, coords)
    P0DG = FunctionSpace(omega_v, "DG", 0)
    return interpolate(u_sol, P0DG)
```

Listing 6: An example point evaluation in Firedrake: the last three lines are our new functionality. A plot of the output with 1000 randomly generated points is shown in Fig. 4.4.

Vertex-only meshes can be created in 1D, 2D and 3D meshes, semi-structured extruded meshes [48], meshes with periodic boundary conditions, and immersed manifold meshes. All cell types Firedrake supports (intervals, triangles, tetrahedra, quadrilaterals and hexahedra) are supported. Extensive automated tests have been added to Firedrake which test DMSwarm creation and vertex-only mesh creation in all of these mesh types, alongside various other tests (behaviour at cell boundaries for example). At the time of writing, only high order meshes cannot have vertex-only meshes immersed in them.[18] The creation of P0DG finite element

[18]This would require changes to the bounding box algorithm, discussed in Sect. 7.4, with improvements suggested in Sect. 7.8.

function spaces on vertex-only meshes are extensively tested to make sure they behave as expected when compared to other Firedrake finite element function spaces.

Other than finite element function spaces which cannot, at present, be dual evaluated using FInAT (notably enriched elements - see Sect. 3.8.3), all are supported.

4.6.1 Solving a Point Forced PDE

```python
from firedrake import *
import numpy as np

omega = UnitSquareMesh(20, 20)
U = FunctionSpace(omega, "CG", 1)

forcing_coords = np.asarray([[0.5, 0.25], [0.5, 0.75], [0.25, 0.5], [0.75, 0.5]])
omega_v = VertexOnlyMesh(omega, forcing_coords)

# Create the RHS point forcing cofunction
P0DG = FunctionSpace(omega_v, "DG", 0)
x, y = SpatialCoordinate(omega_v)
f_v = Function(P0DG).interpolate(x * y)
v_v = TestFunction(P0DG)
L_v = assemble(f_v * v_v * dx)
L = Cofunction(U.dual())
I = Interpolator(TestFunction(U), P0DG)
I.interpolate(L_v, output=L, transpose=True)

# LHS Bilinear form
u = TrialFunction(U)
v = TestFunction(U)
a = inner(grad(u), grad(v)) * dx

bc = DirichletBC(U, 0, "on_boundary")
u_sol = Function(U)  # solution will be stored here
solve(a == L, u_sol, bc)
```

Listing 7: Code for solving the Poisson equation on a unit square with multiple point sources as the right hand side (Eq. 4.6.16). The point sources are samples of $f(x, y) = x \times y$ of which there are 4: $f(0.5, 0.25) = 0.125$, $f(0.5, 0.75) = 0.375$, $f(0.25, 0.5) = 0.125$ and $f(0.75, 0.5) = 0.375$. The solution is shown in Fig. 4.5

The new abstraction ought to allow PDEs with Dirac deltas in as source and sink terms to be solved. Consider the Poisson equation again

$$-\nabla^2 u = f \tag{4.6.1}$$

this is a linear PDE, so we can express it in weak form as a linear variational problem “find

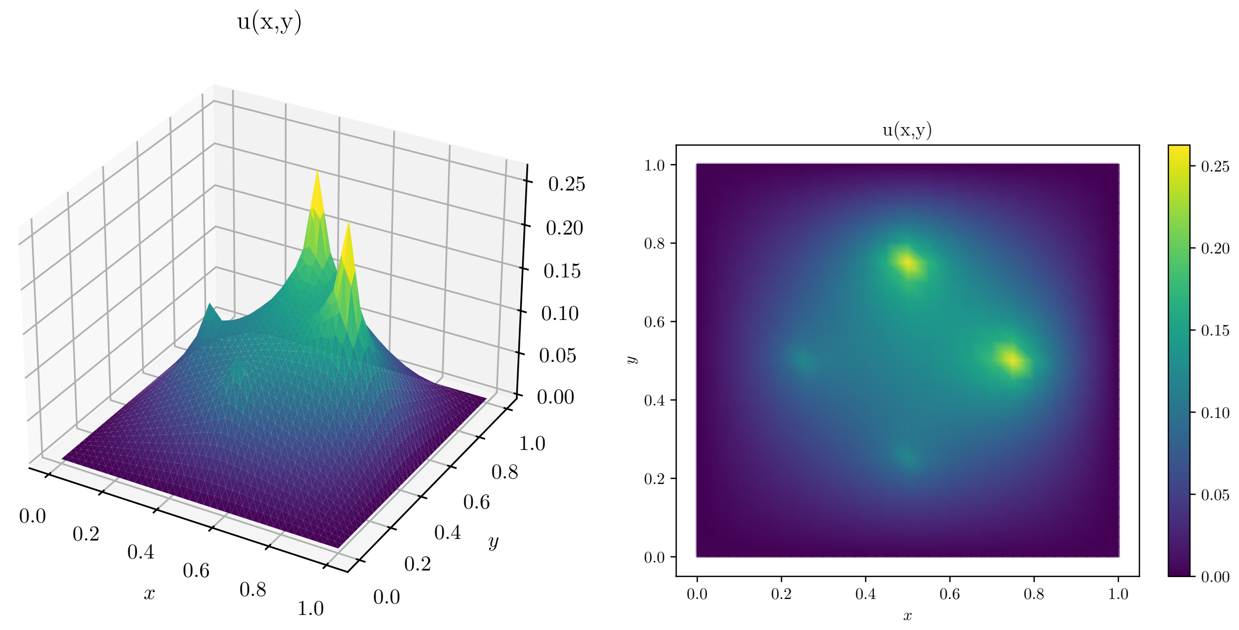


Figure 4.5: The solution to the Poisson equation for 4 points sources at $(0.5, 0.25)$, $(0.5, 0.75)$, $(0.25, 0.5)$ and $(0.75, 0.5)$, sampled from $f(x, y) = x \times y$ created using the code in Listing 7.

$u \in U$ such that

$$\int_\Omega \nabla u \cdot \nabla v \; dx = \int_\Omega f v \quad \forall \, v \in U \; dx \tag{4.6.2}$$

with boundary conditions $u = 0$ on the boundary". The form of this is

$$a(; u, v) = L(; v) \tag{4.6.3}$$

where a is a 2-linear form and L is a 1-linear form (a cofunction) in U^*.

Sometimes we need to solve a linear variational problem where the 1-linear form is constrained by some linear operator

$$L(; v) = L(; A(; u)) \; \forall \, u \in U \tag{4.6.4}$$

where

$$A : U \to V \tag{4.6.5}$$

can be calculated for all test functions $u \in U$ giving some $v \in V$. From the definition of an operator adjoint (Eq. 2.1.2), recalling that $L \in U^*$, we have

$$L(; v) = L(; A(; u)) = A^*(; L)(; u) \quad \forall \, u \in U. \tag{4.6.6}$$

It is common, particularly in physics, to modify the right hand side forcing term with a

Dirac delta

$$\int_\Omega \nabla u \cdot \nabla v \; dx = \sum_{i=0}^{N-1} \int_\Omega f(x) v(x) \delta(x - X_i) \; dx \quad \forall \, v \in U. \tag{4.6.7}$$

This represents N samples at X_i of some continuous function f over the domain. This forcing function only takes on values at the points: it can be rewritten in terms of functions on a vertex-only mesh. Firstly

$$\sum_{i=0}^{N-1} \int_\Omega f(x) \delta(x - X_i) \; dx = \int_{\Omega_v} \mathcal{I}_{\mathrm{P0DG}(\Omega_v)}(; f) \; dx = \sum_{i=0}^{N-1} f(X_i) \in K \tag{4.6.8}$$

is a 0-linear form. We relabel

$$f_v = \mathcal{I}_{\mathrm{P0DG}(\Omega_v)}(; f) \in \mathrm{P0DG}(\Omega_v) \tag{4.6.9}$$

since we can evaluate the expression for f directly in $\mathrm{P0DG}(\Omega_v)$ on each vertex X_i of Ω_v without reference to the parent mesh:

$$f_v(X_i) = f(X_i) \tag{4.6.10}$$

Multiplying this by the test function $v_v \in \mathrm{P0DG}(\Omega_v)$ gives us a 1-linear form (cofunction)

$$L_v \in \mathrm{P0DG}(\Omega_v)^*. \tag{4.6.11}$$

where

$$L_v(; v_v) = \int_{\Omega_v} f_v v_v \; dx \; \forall \, v_v \in \mathrm{P0DG}(\Omega_v). \tag{4.6.12}$$

To formulate the linear variational problem over the whole domain Ω, we need a 1-linear form $L \in U^*$ which takes on the same values as L_v wherever the point evaluations of some $v \in U$ match the values of $v_v \in \mathrm{P0DG}$, i.e.

$$L(; v) = L_v\big(; \mathcal{I}_{\mathrm{P0DG}(\Omega_v)}(; v)\big) \; \forall \, v \in V \tag{4.6.13}$$

$$= \mathcal{I}^*_{\mathrm{P0DG}(\Omega_v)}(; L_v)(; v) \; \forall \, v \in V \tag{4.6.14}$$

from Eq. 4.6.6. Here

$$\mathcal{I}^*_{\mathrm{P0DG}(\Omega_v)} : \mathrm{P0DG}(\Omega_v)^* \to U^*. \tag{4.6.15}$$

The forcing term is therefore

$$\sum_{i=0}^{N-1} \int_\Omega f(x) v(x) \delta(x - X_i) \; dx = \mathcal{I}^*_{\mathrm{P0DG}(\Omega_v)}\left(; \int_{\Omega_v} f_v v_v \; dx\right) \in U^* \tag{4.6.16}$$

which we solve $\forall \, v_v \in \mathrm{P0DG}(\Omega_v)$ thereby constraining the test functions $v \in U$ for the rest of the terms in the linear variational problem. The construction of the adjoint dual evaluation interpolation operator from the interpolation matrix is described in chapter 5, Sect. 5.4.1.

Code for solving the Poisson equation with this right hand side is shown in Listing 7 and the solution u in Fig. 4.5.

4.7 Future Work

The requirement of a parent mesh at vertex-only mesh construction is a limitation of the current implementation. However, most[19] conceivable use cases of vertex-only meshes require association with another mesh. For example, point evaluations may be done more than once without changing the list of coordinates. Since the search over parent mesh cells and identification of coordinate locations within them is computationally expensive, we avoid computing this every time by storing the information on the vertex-only mesh. Nevertheless, making this an optional constructor argument and allowing the association of a `VertexOnlyMesh` with multiple meshes is possible future work.

The choice to have PyOP2 loop over vertex-only mesh cells rather than parent mesh cells may not be optimal for cases where we have large numbers of vertex-only mesh cells in each cell of the parent mesh. The current implementation has TSFC transforming to and from the parent mesh cell reference coordinates for each vertex. Looping over parent mesh cells could reduce this overhead but would introduce additional complexity to the data structures and would not be optimal for cases where we have a sparse vertex-only mesh. Any investigation of this approach would require careful profiling of the TSFC kernels for each case.

The re-appropriation of `QuadratureElement` for runtime dual evaluation works for this case, but somewhat hides its real use. For full generality a new element type could be created, e.g. `RuntimeElement`, which would build a dual basis from (optionally) runtime-specified point evaluation locations and weights to apply to those locations as in Eq. 3.7.2:

$$\phi_i^*(;f) = \sum_j w_j f(x_j). \tag{4.7.1}$$

4.8 Summary of Contributions

I have developed a mathematical abstraction, vertex-only meshes and P0DG finite element function spaces on them, that allow one to reason about point data in the finite element method framework. I have defined an operation, dual-evaluation interpolation, which is the point evaluation operation. This is made possible because finite element functions, which approximate fields, have, with the exception of discontinuous Galerkin methods, well defined values everywhere on a domain. To my knowledge, this is the first fully generalisable approach for integrating point data and finite element methods.

By separating out the data from the spatial coordinates, I avoid making assumptions about the use case. Whilst there are plenty of libraries that consider particles within a finite element

[19]but not all, see Sect. 7.6.3 for an example

method ecosystem (as I have identified), they are generally specialised towards particular use cases such as particle tracing and particle-in-cell methods (for more on these see chapter 7, Sect. 7.3). Since my point evaluation operation is defined at a high level, it can be interacted with as a mathematical operator without needing to worry about implementation. I have demonstrated that this can be used to describe point source terms and implemented an example. As I will go on to show, this will also allow me to differentiate the operator[20], allowing not only interrogation of point data in a finite element function, but also assimilation of that data.

I have implemented my abstractions in Firedrake: Vertex-only meshes are created in a similar way to other Firedrake meshes, though use a special constructor to let them be associated with another mesh and accept a set of coordinates and store their mesh topology in a PETSc DMSwarm rather than a DMPlex (see Sect. 4.5.1).

I identified a straightforward algorithm for implementing the dual evaluation operation (see Sect. 4.4) which ought to be applicable to other finite element libraries. I used this algorithm in my implementation, making the necessary modifications to loop over vertex-only mesh cells and retrieve the parent mesh cell and generate a kernel with TSFC using its runtime-tabulation capability (see Sect. 4.5.3). To integrate with my FInAT dual evaluation capability, described in chapter 3, I used the newly created FInAT `UnknownPointSet` with the pre-existing FInAT `QuadratureElement` (see appendix A.1.4). The implementation has been extensively tested and is now the recommended point evaluation API in Firedrake.

[20]see Sect. 5.5

Chapter 5

Applying Automatic Differentiation to Interpolation

5.1 Introduction

Consider some arbitrary function F which maps some set of inputs to a set of outputs: If we want to know how changing certain inputs affects the outputs we call that *sensitivity analysis*. If F is some model of a phenomenon and we want to know which inputs give us a particular set of outputs we call that *solving an inverse problem* or *variational data assimilation*. If we want to change some parameters of F to correlate particular inputs and outputs we, amongst other things, call this *machine learning*. Knowing the gradient of F - how much the outputs of F changes given perturbations in its inputs - is useful when trying to solve all these problems.

Take for example an inverse problem: we have a so-called *forward model* of a phenomenon we are interested in (F) and measurements of that phenomenon. For the sake of simplicity we say that the outputs of F need to match our measurements. We set up an optimisation problem where we minimise, for given inputs to our forward model, the difference between the model outputs and the measurement. We call this a misfit function, misfit functional or objective function depending on which field you work in:

$$\min_{\text{inputs}} \text{misfit}(\text{inputs}) = \|F(\text{inputs}) - \text{measurements}\|^2_{\text{some norm}} \tag{5.1.1}$$

$$:= \min_{x} J(x) = \|F(x) - y\|^2_{\text{some norm}}. \tag{5.1.2}$$

We could blindly change our parameters until we find the lowest value of our misfit - clever ways of doing this are called *Monte Carlo methods*. Alternatively we note that the gradient of the misfit is zero at a minimum

$$\frac{\mathrm{d}J}{\mathrm{d}x} = 0 \tag{5.1.3}$$

and perform *gradient based optimisation*. In the simplest case, this means moving $J(x)$ for some x in the direction of steepest gradient until our gradient is zero. Since our measurements y are fixed this means finding the gradient of F.

A field of research which has emerged for this purpose is Automatic or Algorithmic Differentiation (AD). Typically this considers F as an algorithm or computer program, i.e.

$$F : \mathbb{R}^n \to \mathbb{R}^m \tag{5.1.4}$$

which is made up of a series of fundamental operations encapsulated in the code for that program. Since we have a series of fundamental operations, AD makes the leap of seeing that we can define F as a gigantic composition of functions

$$F = f_p \circ f_{p-1} \circ ... \circ f_1 \tag{5.1.5}$$

which we can then apply the chain rule to.

Here a comprehensive description of AD as it is applied to Firedrake using the dolfin-adjoint/pyadjoint [46] tool will be given. This will start by introducing the key mathematical concepts then describe how they can be used to build an AD tool appropriate for us. The aim here is to allow us to apply this to the interpolation operator in Firedrake and to do so in a way that gives us gradients quickly. An initial application of this work is found in chapter 6. A summary of the contributions of this chapter can be found in section 5.6.

5.2 Gateaux Derivatives

Gradient based minimisation techniques require a way of computing the gradient of a function or functional with respect to one or more of its parameters. Consider some arbitrary function f which maps between function spaces U and V

$$f : U \to V \tag{5.2.1}$$

i.e.

$$f(u) \in V \; \forall \, u \in U. \tag{5.2.2}$$

This is a function which operates *on arbitrary functions* $u \in U$ (which we consider to be vectors) to produce new functions in V. We need a way of expressing the derivative of f with respect to u given some small perturbation in the vector function space U.

The Gateaux derivative, which is defined for all complete normed vector spaces, and therefore all finite element function spaces, is suitable for this and, following the notation used by Schwedes et al. [29], is defined as the bounded linear map such that

$$\begin{gathered} df_u(u; u') = \lim_{\epsilon \to 0} \left(\frac{f(u + \epsilon u') - f(u)}{\epsilon} \right) \\ \forall \, u' \in U \end{gathered} \tag{5.2.3}$$

where the subscript $_u$ indicates that the derivative is with respect to u. Our perturbation in the U direction is the new argument u' which df_u is necessarily linear in; the semicolon ';' indicates this.[1] All arguments to the left of ';' are possibly nonlinear, whilst those to the right are necessarily linear. df_u is therefore a map

$$df_u : U \times \underbrace{U}_{\text{linear}} \to V, \tag{5.2.4}$$

(where the second U necessarily linear) i.e.

$$df_u(u; u') \in V. \tag{5.2.5}$$

[1]If, as the limit $\epsilon \to 0$ is taken, one yields something which is not linear in u', we say that the function f is not Gateaux differentiable.

Much as Hilbert spaces generalise Euclidean vectors ($u \in \mathbb{R}^n$ or $c \in \mathbb{C}^n$) and their associated spaces ($\mathbb{R}^n$ and $\mathbb{C}^n$) to include infinite dimensions and vectors which are functions, the Gateaux derivative generalises the directional derivative to generic vectors (such as functions) with, potentially, infinite dimensions. In the case where f is a scalar function applied to a vector, i.e.

$$f : \mathbb{R}^n \to \mathbb{R}, \tag{5.2.6}$$

the Gateaux derivative yields

$$df_u(u; u') = \nabla f \cdot u' \tag{5.2.7}$$

If, as we show in the appendix section A.2.1, we consider a vector function applied to a vector, i.e.

$$f : \mathbb{R}^n \to \mathbb{R}^m \tag{5.2.8}$$

the Gateaux derivative yields a Jacobian matrix vector product

$$df_u(u; u') = J_f \cdot u'. \tag{5.2.9}$$

This pattern can be extended: returning to the case of $f : V \to U$ we can consider df_u as a $V \times U$ sized infinite dimensional tensor operator applied to $u' \in U$ since df_u is linear in u'. This gives a very useful alternative notation which we use to aid comprehension throughout this chapter

$$df_u(u; u') := \left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_u \cdot u' \tag{5.2.10}$$

where $|_u$ is shorthand for 'evaluated at u'.

It is key to emphasise here that this is a derivative of a function f of functions $u \in U$: the Gateaux derivative can be thought of as a formalisation of the so-called 'functional derivative' found in the calculus of variations that is used in physics and engineering for Lagrangian and Hamiltonian mechanics.

If we have more than one argument we can also define a partial derivative. Given the (not necessarily linear) map

$$g : U \times U \to V \tag{5.2.11}$$

i.e.

$$g(u, \mu) \in V \ \forall\, u, \mu \in U \tag{5.2.12}$$

we define

$$\begin{gathered} \partial g_u(u, \mu; u') = \lim_{\epsilon \to 0} \left(\frac{g(u + \epsilon u', \mu) - g(u, \mu)}{\epsilon} \right) \\ \forall\, u' \in U \end{gathered} \tag{5.2.13}$$

as the partial derivative with respect to u where u' is a linear perturbation in the u direction.

Since we have 3 arguments, this is now a map

$$\partial g_u : U \times U \times \underbrace{U}_{\text{linear}} \to V \tag{5.2.14}$$

(where the third U is necessarily linear) i.e.

$$\partial g_u(u, \mu; u') \in V. \tag{5.2.15}$$

As with equation 5.2.10, this can be expressed as

$$\partial g_u(u, \mu; u') := \frac{\partial g}{\partial u} \cdot u'. \tag{5.2.16}$$

We can understand the partial Gateaux derivative in finite dimensions by considering a scalar function of a vector f as a scalar function of the scalar vector elements g

$$f : \mathbb{R}^n \to \mathbb{R} \equiv g : \underbrace{\mathbb{R} \times \mathbb{R} \times ... \times \mathbb{R}}_{n \text{ times}} \to \mathbb{R} \text{ i.e.} \tag{5.2.17}$$

$$f(u) \equiv g(u_0, ..., u_{n-1}) \tag{5.2.18}$$

Now that we have a function of multiple variables we can take a partial derivative with respect to one of them

$$\partial g_{u_i} : \underbrace{\mathbb{R} \times \mathbb{R} \times ... \times \mathbb{R}}_{n \text{ times}} \times \underbrace{\mathbb{R}}_{\text{new!}} \to \mathbb{R} \tag{5.2.19}$$

which recovers the usual definition of the partial derivative in vector calculus taken with respect to a vector element u_i (a scalar number), multiplied by the direction vector u'

$$\partial g_{u_i}(u; u') := \underbrace{\left.\frac{\partial g}{\partial u_i}\right|_u}_{\equiv \left.\frac{\partial f}{\partial u_i}\right|_u} \cdot u'. \tag{5.2.20}$$

5.2.1 Chain Rule

As with most derivatives we can define a chain rule. Consider the following

$$J : V \to \mathbb{R} \text{ i.e. } J(v) \in \mathbb{R}\ \forall v \in V, \tag{5.2.21}$$

$$f : U \to V \text{ i.e. } f(u) \in V\ \forall u \in U, \tag{5.2.22}$$

$$J \circ f : U \to \mathbb{R} \text{ i.e. } J(f(u)) \in \mathbb{R}\ \forall u \in U, \tag{5.2.23}$$

where '$\circ$' is the composition operator hence it implying J being a function of f. We can think of $J \circ f$ as the composition of two maps

$$J \circ f : U \underbrace{\rightarrow}_{f} V \underbrace{\rightarrow}_{J} \mathbb{R}. \tag{5.2.24}$$

We want the derivative with respect to u

$$d[J \circ f]_u : U \times \underbrace{U}_{\text{linear}} \rightarrow \mathbb{R} \text{ i.e. } d[J \circ f]_u(u; u') \in \mathbb{R}\ \forall u \in U. \tag{5.2.25}$$

The chain rule here has us substituting in $v = f|_u \in V$ (see the composite map equation 5.2.24), where $|_u$ is again shorthand for 'evaluated at u', to get

$$d[J \circ f]_u(u; u') = dJ_v(v = f|_u; v' = df_u(u; u')) \tag{5.2.26}$$

which, in the generic infinite-dimensional tensor notation of equation 5.2.10, is the familiar expression

$$\left.\frac{\mathrm{d}[J \circ f]}{\mathrm{d}u}\right|_u \cdot u' = \left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v=f|_u} \cdot \underbrace{\left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_u \cdot u'}_{v'|_{v=f|_u}}. \tag{5.2.27}$$

The idea of composite maps (equation 5.2.24) is highlighted here to show how the chain rule has us dealing with the derivatives of each underlying map (i.e. function) separately. As expressions get more complex, for example when trying to take the derivative of an entire computer program as we will go on to discuss, we will see that we just need to employ the chain rule.

For completeness, the generic expression for the chain rule, where we are dealing with multi-variable functions, is given by the expression for the total derivative. To define this we introduce a new multivariate functional J' which maps from V and a new function space X

$$J' : V \times X \rightarrow \mathbb{R} \text{ i.e. } J'(v, x) \in \mathbb{R}\ \forall v \in V, x \in X, \tag{5.2.28}$$
$$f : U \rightarrow V \text{ i.e. } f(u) \in V\ \forall u \in U, \tag{5.2.29}$$
$$g : U \rightarrow X \text{ i.e. } g(x) \in U\ \forall x \in X, \tag{5.2.30}$$
$$J' \circ (f, g) : U \rightarrow \mathbb{R} \text{ i.e. } J'(f(u), g(u)) \in \mathbb{R}\ \forall u \in U, \tag{5.2.31}$$

where $J' \circ (f, g)$ is used to express the composition of J' on the tuple of functions f and g. Our composite map is

$$J' \circ (f, g) : U \underbrace{\rightarrow}_{(f,g)} (V, X) \underbrace{\rightarrow}_{J'} \mathbb{R}. \tag{5.2.32}$$

We want the total derivative with respect to u

$$d[J' \circ (f,g)]_u : U \times \underbrace{U}_{\text{linear}} \to \mathbb{R} \text{ i.e. } d[J' \circ (f,g)]_u(u;u') \in \mathbb{R} \tag{5.2.33}$$

that is given by

$$d[J' \circ (f,g)]_u(u;u') = \partial J'_v\big(v = f|_u; v' = df_u(u;u')\big) \\ + \partial J'_x\big(x = g|_u; x' = dg_u(u;u')\big). \tag{5.2.34}$$

In the generic infinite-dimensional tensor notation of equations 5.2.10 and 5.2.16 this is

$$\left.\frac{\mathrm{d}[J' \circ (f,g)]}{\mathrm{d}u}\right|_u \cdot u' = \left.\frac{\partial J'}{\partial v}\right|_{v=f|_u} \cdot \underbrace{\left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_u \cdot u'}_{v'|_{v=f|_u}} + \left.\frac{\partial J'}{\partial x}\right|_{x=g|_u} \cdot \underbrace{\left.\frac{\mathrm{d}g}{\mathrm{d}u}\right|_u \cdot u'}_{x'|_{x=g|_u}}. \tag{5.2.35}$$

If a partial derivative is required (here equal to the total derivative since $J' \circ (f,g)$ is only a function of u) the expressions in each notation are

$$\partial[J' \circ (f,g)]_u(u;u') = \partial J'_v\big(v = f|_u; v' = \partial f_u(u;u')\big) \\ + \partial J'_x\big(x = g|_u; x' = \partial g_u(u;u')\big). \tag{5.2.36}$$

and

$$\left.\frac{\partial[J' \circ (f,g)]}{\partial u}\right|_u \cdot u' = \left.\frac{\partial J'}{\partial v}\right|_{v=f|_u} \cdot \underbrace{\left.\frac{\partial f}{\partial u}\right|_u \cdot u'}_{v'|_{v=f|_u}} + \left.\frac{\partial J'}{\partial x}\right|_{x=g|_u} \cdot \underbrace{\left.\frac{\partial g}{\partial u}\right|_u \cdot u'}_{x'|_{x=g|_u}}. \tag{5.2.37}$$

An example of the chain rule being used can be found in the appendix, section A.2.2. The beginnings of the relationship between the chain rule and Automatic Differentiation (AD) is discussed at the end of the example.

5.3 Automatic Differentiation

Let us return to the introduction (section 5.1), where we considered an algorithm or computer program to be

$$F : \mathbb{R}^n \to \mathbb{R}^m \tag{5.3.1}$$

which we apply the chain rule to

$$F = f_p \circ f_{p-1} \circ \dots \circ f_1. \tag{5.3.2}$$

In most AD literature one considers F as a map between two sets of real numbers as above. In Firedrake F is a PDE where one argument is a *control* variable $m \in M$ giving a solution

$z \in Z$ which we typically use in an optimisation setting (minimising some function, such as a misfit) as above. Solving such problems is known as *PDE constrained optimisation.* The most general case then is to consider F as a map between arbitrary vector spaces

$$F : M \to Z \tag{5.3.3}$$

where, above, we have $M = \mathbb{R}^n$ and $Z = \mathbb{R}^m$. It is important to keep this generalisation because in the finite element method we consider vector spaces which are finite dimensional but are generally not $\mathbb{R}^n$. This has important impacts on the adjoint approach where we are forced to consider the covectors to vectors which we refer to as their adjoints. As we saw in section 1.4 this means finding the particular inverse Riesz representer for the vector.

Being aware that the Riesz map is not necessarily 'take the transpose' (which it is only for $\mathbb{R}^n$ with the l_2 inner product) highlights problems which otherwise would be difficult to understand: For example, users of Firedrake and pyadjoint/dolfin-adjoint (discussed later) expect to get *convergence*: when they supply finer and finer meshes the solution to their problem, be it solving a PDE or a PDE constrained optimisation problem, converges towards the true solution. Assuming we are using the adjoint method, then each time we take the adjoint of a given vector in a particular discretised vector space we use a different inner product (matrix M in equation 1.4.35). This is fine since there is an 'equivalence of norms' theorem which states that all norms in a finite dimensional vector spaces are equivalent up to some constant which stays the same for the given vector space. However, when we change meshes or bases, such as when we perform refinement, we change to a different finite dimensional vector space: here the equivalence of norms theorem still holds but the constants will not be the same. Rather than our optimisation problem tending to the same solution, we can find ourselves ending up with different solutions particularly if we refine our mesh in non-uniform ways. This is made worse by optimisation algorithms typically having some stopping condition to avoid going on forever: since different norms have different associated error in finite dimensions, you can reach a stopping condition in a very different part of the PDE's solution space depending on the specifics of the mesh refinement.

AD's eponymous differentiation works by taking the derivative with respect to some argument multiplied by a *perturbation direction* of that argument. As we will go on to see, the perturbation direction is propagated through the function decomposition. Henceforth we will refer to the perturbation direction as the *direction.* This form of differentiation is, by definition, a directional derivative of which the Gateaux derivative is the most general case. Fortunately the Gateaux derivative is the correct derivative to use for our general case of maps between vector spaces. Unlike elsewhere in AD literature, our description is here generalised to use Gateaux derivatives throughout.

Maintaining such a high level mathematical abstraction allows concrete discretisation to happen when a particular mesh and finite element function space basis is supplied. This is very convenient: at the programmatic level one can easily change mesh and basis from being, for example, 2nd order discontinuous Lagrange polynomials on a course mesh of a domain to 3rd order Bernstein polynomials on a fine mesh.

AD has two forms: The first, and most easy to understand, is the *tangent linear approach*, also known as the *forward mode* of AD and *forward accumulation*. This was first described by Wengert [49] and, as we will see, is the usual approach one might manually take to applying the chain rule to a set of functions.

The second is the, often much harder to understand, *adjoint approach*, also known as the *reverse mode* of AD, or *reverse accumulation*. This has a more complex history spanning multiple fields and has been reinvented several times (see section 3.3 of Baydin et al. [50]). Of particular note, the adjoint approach is the generalisation of *backpropagation* algorithms used in machine learning for training neural networks (see section 3.2 of Baydin et al. [50]). In essence, the idea is to save decomposed function intermediate values then use the chain rule in the opposite direction to that usually taken - this will be explored in more depth later.

AD tools are generally implemented in one of two ways. One approach is to analyse code bases using sets of rules that define the derivatives of given operations to produce new code for finding the gradient - a so-called *source-to-source* approach. So $x = \sin(y)$ is transformed to $x' = \cos(y) \cdot y'$. The other is to augment the variable types in your code with information about the derivative: i.e. still use your rules but introduce a new type which contains both x and x' which can then be propagated through the code and modified as necessary. This requires modification of the source code such that functions can accept the augmented type. Since the usual Object Oriented Programming way to do that is known as *overloading* this approach is known as *operator overloading*. For more see chapter 2 of [51].

Partly due to the recent popularity of machine learning, and AD's important role in it, there are now many examples of AD libraries available such as Google JAX[52] and within machine learning tools like TensorFlow[53] and PyTorch[54]. A very non-exhaustive, but wide ranging, list of libraries can be found at `autodiff.org`.

We will explain the tangent linear and adjoint approaches using example functions which we then go on to use in our explanation of pyadjoint[46] and the specific work here of making interpolation compatible with pyadjoint. Consider a scenario where we have

$$J \circ f \circ (h, k) : M \to \mathbb{R} \text{ i.e. } J(f(h(m), h(m))) \in \mathbb{R} \tag{5.3.4}$$

$J \circ f \circ (h, k)$ is the decomposition of the program F which we wish to apply AD to. Breaking this down we have

$$J : V \to \mathbb{R} \text{ i.e. } J(v) \in \mathbb{R}, \tag{5.3.5}$$

$$f : U \times G \to V \text{ i.e. } f(u,g) \in V, \tag{5.3.6}$$

$$h : M \to U \text{ i.e. } h(m) \in U, \tag{5.3.7}$$

$$k : M \to G \text{ i.e. } k(m) \in G. \tag{5.3.8}$$

J is a functional, such as in integration over a domain of V. f can be any operation which takes us from two function spaces U and G into one V. h and k are functions applied to a variable m. We assign intermediate variables as outputs of each function

$$j = J|_v \tag{5.3.9}$$

$$v = f|_{(u,g)} \tag{5.3.10}$$

$$u = h|_m \tag{5.3.11}$$

$$g = k|_m. \tag{5.3.12}$$

In combining function definitions (here equations 5.3.5 to 5.3.8) with intermediate output variable definitions (here equations 5.3.9 to 5.3.12) we have defined a Directed Acyclic Graph (DAG). Our DAG is shown pictorially in figure 5.1. A DAG is the natural way of displaying any sequence of operations that can be broken down into a composition of functions with intermediate output variables (i.e. an algorithm which can be broken down using the chain rule). Crucially we note that output function variables can be inputs to more than one downstream function: here we limit ourselves to a simple case where the variable m is an argument of both h and k. This will be explored in more detail in section A.2.5. This simple case allows us to write down our whole program F as a function decomposition $J \circ f \circ (h,k)$. However in general you need to write out or draw a DAG to fully describe a function decomposition.

The Gateaux derivative of $J \circ f \circ (h,k)$ is

$$d[J \circ f \circ (h,k)]_m : M \times \underbrace{M}_{\text{linear}} \to \mathbb{R} \text{ i.e. } d[J \circ f \circ (h,k)]_m(m;m') \in \mathbb{R}. \tag{5.3.13}$$

We can Curry this into two functions

$$d[J \circ f \circ (h,k)]_m(m;m') = \underbrace{D_1(m)}_{D_0 = D_1|_m}(;m') \tag{5.3.14}$$

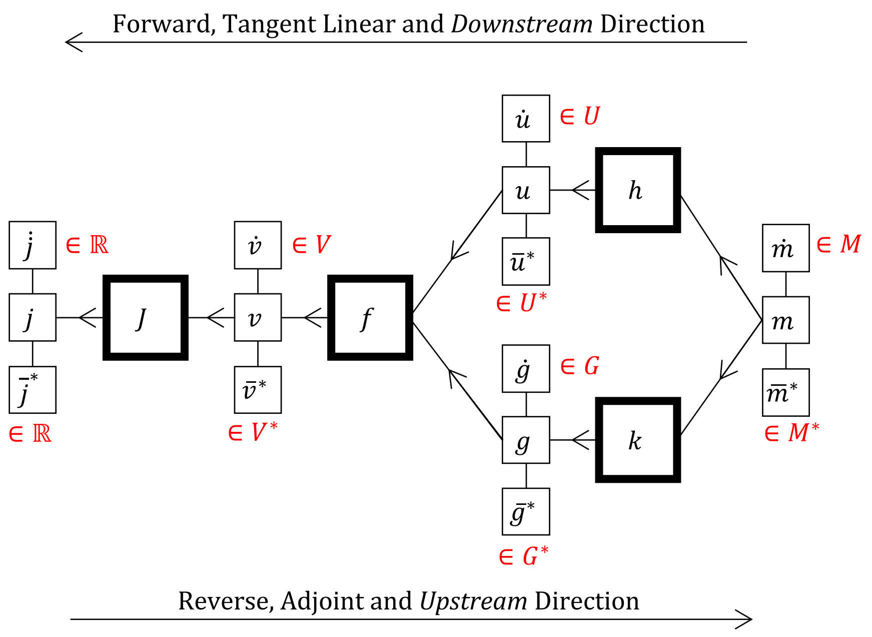


Figure 5.1: Directed Acyclic Graph (DAG) for the operations given in equations 5.3.5 to 5.3.8 linked by the intermediate output variables in equations 5.3.9 to 5.3.12. Functions are in large boxes with input and output variables directly connected. Associated Tangent Linear Mode (TLM) variables are shown with dots above, whilst adjoint mode variables have bars above.[2] The vector spaces that variables are members of are shown in red above-right for input/output/TLM variables and below for adjoint variables.

where D_1 is the not-necessarily-linear operation of supplying the point at which to evaluate

$$D_1 : M \to (\underbrace{M}_{\text{linear}} \to \mathbb{R}) \text{ i.e. } D_1(m)(;\bullet) \in (\underbrace{M}_{\text{linear}} \to \mathbb{R}) \;\forall m \in M \tag{5.3.15}$$

$$= d[J \circ f \circ (h,k)]_m(m;\bullet) \tag{5.3.16}$$

$$:= \left.\frac{d[J \circ f \circ (h,k)]}{dm}\right|_m \cdot \bullet \tag{5.3.17}$$

and D_0 is the necessarily-linear operation of supplying the direction

$$D_0 : \underbrace{M}_{\text{linear}} \to \mathbb{R} \text{ i.e. } D_0(;m') \in \mathbb{R} \;\forall m' \in M \tag{5.3.18}$$

$$= d[J \circ f \circ (h,k)]_m(m;m') \tag{5.3.19}$$

$$:= \left.\frac{d[J \circ f \circ (h,k)]}{dm}\right|_m \cdot m'. \tag{5.3.20}$$

which can be written as

$$D_0 = D_1|_m \text{ for given } m \in M. \tag{5.3.21}$$

[2]These are the primal form of the adjoint mode variables. Each has a dual-space equivalent ($\bar{v}^* \in V^*$ for example), which becomes relevant when discussing differing approaches to adjoint mode AD. See Sect. 5.3.2.

The symbol '$\bullet$' once again indicates where we place our operand.[3]

Our aim is to find the linear operator $D_0 = D_1|_m$ at m such that it can supplied with a direction. This is the process of 'linearisation' about a point, here m. We refer to m as the 'control'. Going forwards we will write this as $d[J \circ f \circ (h,k)]_m(m;\bullet)$. This operator is usually called the *Jacobian*, here taken with respect to m. By comparison see the 'Jacobian matrix', which is an expansion of this operator in a particular finite dimensional basis (see appendix section A.2.1).

5.3.1 Tangent Linear Approach

The Jacobian we wish to find takes a vector $m' \in M$ and gives us the Jacobian vector product. If we have a complete basis for our vector space M then, by operating on each basis vector in turn, we recover the complete Jacobian. This is the tangent linear approach to retrieving the full Jacobian: you work out how to find a directional derivative (more on that below) and then supply *seed* vectors on the right hand side until you recover the complete Jacobian.

In the tangent linear approach we introduce intermediate tangent linear operators which give us intermediate Jacobians of our functions with respect to our control:

$$\mathring{j}(;\bullet) = d[J \circ f \circ (h,k)]_m(m;\bullet) : M \to \mathbb{R} \text{ for given } m \in M \tag{5.3.22}$$

$$\mathring{v}(;\bullet) = d[f \circ (h,k)]_m(m;\bullet) : M \to V \text{ for given } m \in M \tag{5.3.23}$$

$$\mathring{u}(;\bullet) = dh_m(m;\bullet) : M \to U \text{ for given } m \in M \tag{5.3.24}$$

$$\mathring{g}(;\bullet) = dk_m(m;\bullet) : M \to G \text{ for given } m \in M \tag{5.3.25}$$

$$\mathring{m}(;\bullet) = dm_m(m;\bullet) = I(;\bullet) : M \to M \text{ for given } m \in M \tag{5.3.26}$$

where I is an identity operator which performs no operation (i.e. an identity matrix of $\text{span}(M) \times \text{span}(M)$ dimensions) and $\mathring{j}$ is the operator we aim to find. If we apply the chain rule to these operators, we find a neat pattern emerges.

We first note that

$$\begin{aligned}\mathring{u}(;\bullet) &= dh_m\big(m; m' = dm_m(m;\bullet)\big) \\ &= dh_m\big(m; m' = \mathring{m}(;\bullet)\big),\end{aligned} \tag{5.3.27}$$

and

$$\begin{aligned}\mathring{g}(;\bullet) &= dk_m\big(m; m' = dm_m(m;\bullet)\big) \\ &= dk_m\big(m; m' = \mathring{m}(;\bullet)\big).\end{aligned} \tag{5.3.28}$$

Then, using the chain rule (immediately swapping partial derivatives for ordinary where they

[3]Note that our alternative notation for Gateaux derivatives is similar, though not identical, to the Liebniz notation for AD introduced by Christianson [55].

are equal - i.e. where we have functions of 1 variable) we find that

$$\begin{aligned}\mathring{v}(;\bullet) &= d[f\circ(h,k)]_m(m;\bullet)\\ &= \partial f_u\big(u=h|_m, g=k|_m; u'=dh_m(m;\bullet)\big) + \partial f_g\big(u=h|_m, g=k|_m; g'=dk_m(m;\bullet)\big)\\ &= \partial f_u\big(u=h|_m, g=k|_m; u'=\mathring{u}(;\bullet)\big) + \partial f_g\big(u=h|_m, g=k|_m; g'=\mathring{g}(;\bullet)\big)\end{aligned} \tag{5.3.29}$$

and

$$\begin{aligned}\mathring{j}(;\bullet) &= d[J\circ f\circ(h,k)]_m(m;\bullet)\\ &= dJ_v\big(v=f|_{u=h|_m,g=k|_m}; v'=d[f\circ(h,k)]_m(m;\bullet)\big)\\ &= dJ_v\big(v=f|_{u=h|_m,g=k|_m}; v'=\mathring{v}(;\bullet)\big).\end{aligned} \tag{5.3.30}$$

We see then that we can propagate our directional derivative seed $\dot{m}$ forwards from $\mathring{m}$, to $\mathring{u}$ and $\mathring{g}$, eventually reaching $\mathring{j}$. In practice this is done for a concrete seed by introducing intermediate variables

$$\dot{m} = dm_m(m;m'=\dot{m})(=I(;\dot{m}))\in M \text{ for given } m\in M \tag{5.3.31}$$

$$\dot{u} = dh_m(m;m'=\dot{m})\in U \text{ for given } m\in M \tag{5.3.32}$$

$$\dot{g} = dk_m(m;m'=\dot{m})\in G \text{ for given } m\in M \tag{5.3.33}$$

$$\dot{v} = \partial f_u(u=h|_m, g=k|_m; u'=\dot{u}) + \partial f_g(u=h|_m, g=k|_m; g'=\dot{g})\in V \text{ for given } m\in M \tag{5.3.34}$$

$$\dot{j} = dJ_v(v=f|_{u=h|_m,g=k|_m}; v'=\dot{v})\in\mathbb{R} \text{ for given } m\in M \tag{5.3.35}$$

where we have replaced the operators $\mathring{m}(;\bullet)$ to $\mathring{j}(;\bullet)$ with our intermediate variables $\dot{m}$ to $\dot{j}$. These are referred to as ‘TLM variables’. The downstream-most TLM variable is the Jacobian-vector product for our given TLM seed vector $\dot{m}$. These TLM variables, alongside the definitions of the tangent linear operators define the DAG shown pictorially in figure 5.2.

Usefully, the above calculation of derivatives for some seed $\dot{m}$ can be done in lock step with the calculation of functions themselves for given m without having to save any intermediate values. Unfortunately to recover the Jacobian operator we have to traverse our DAG (i.e. compute directional/Gateaux derivatives) $\dim(M)$ times assuming that our basis is $\dim(M)$. So if $M=\mathbb{R}^m$ then our computational time scales with m. More specifically, if we were finding a Jacobian for a function $F:\mathbb{R}^n\to\mathbb{R}^m$ (which would be an $m\times n$ Jacobian matrix) then the time would scale as $n\cdot c\cdot \text{operations}(F)$ where c is a constant [56]. We can see, therefore, that an alternative approach is needed.

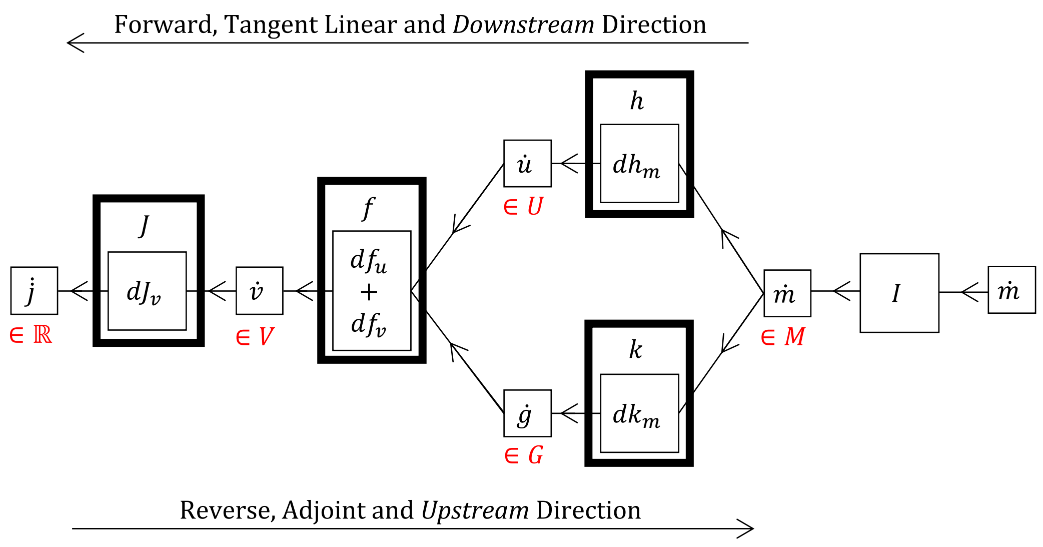


Figure 5.2: Directed Acyclic Graph (DAG) for the operations given in equations 5.3.22, 5.3.27, 5.3.28, 5.3.29 and 5.3.30 linked by the intermediate output variables in equations 5.3.31 to 5.3.35. Functions are in large boxes with input and output variables directly connected. In practice the initial identity operation is skipped. The *blocks* associated with each operation in pyadjoint which annotate each operation and allow derivatives to be calculated (see section 5.4) are shown as thick boxes. The vector spaces that variables are members of are shown below each set in red. We see that the DAG is similar to figure 5.1. For each function output variable there is a corresponding TLM variable. Instead of evaluating functions we evaluate the relevant derivative to get the next TLM variable.

In our alternative notation this is:

$$\mathring{j}(;\bullet) = \left.\frac{\mathrm{d}[J \circ f \circ (h,k)]}{\mathrm{d}m}\right|_m \cdot \bullet : M \to \mathbb{R} \text{ for given } m \in M \tag{5.3.36}$$

$$\mathring{v}(;\bullet) = \left.\frac{\mathrm{d}[f \circ (h,k)]}{\mathrm{d}m}\right|_m \cdot \bullet : M \to V \text{ for given } m \in M \tag{5.3.37}$$

$$\mathring{u}(;\bullet) = \left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_m \cdot \bullet : M \to U \text{ for given } m \in M \tag{5.3.38}$$

$$\mathring{g}(;\bullet) = \left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_m \cdot \bullet : M \to G \text{ for given } m \in M \tag{5.3.39}$$

$$\mathring{m}(;\bullet) = \left.\frac{\mathrm{d}m}{\mathrm{d}m}\right|_m \cdot \bullet = I \cdot \bullet : M \to M \text{ for given } m \in M \tag{5.3.40}$$

then

$$\begin{aligned}\mathring{u}(;\bullet) &= \left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_m \cdot \left.\frac{\mathrm{d}m}{\mathrm{d}m}\right|_m \cdot \bullet \\ &= \left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_m \cdot \mathring{m}(;\bullet),\end{aligned} \tag{5.3.41}$$

$$\begin{aligned}\mathring{g}(;\bullet) &= \left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_m \cdot \left.\frac{\mathrm{d}m}{\mathrm{d}m}\right|_m \cdot \bullet \\ &= \left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_m \cdot \mathring{m}(;\bullet),\end{aligned} \tag{5.3.42}$$

$$\begin{aligned}\mathring{v}(;\bullet) &= \left.\frac{\mathrm{d}[f\circ(h,k)]}{\mathrm{d}m}\right|_m \cdot \bullet \\ &= \left.\frac{\partial f}{\partial u}\right|_{u=h|_m, g=k|_m} \cdot \left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_m \cdot \bullet + \left.\frac{\partial f}{\partial u}\right|_{u=h|_m, g=k|_m} \cdot \left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_m \cdot \bullet \\ &= \left.\frac{\partial f}{\partial u}\right|_{u=h|_m, g=k|_m} \cdot \mathring{u}(;\bullet) + \left.\frac{\partial f}{\partial u}\right|_{u=h|_m, g=k|_m} \cdot \mathring{g}(;\bullet),\end{aligned} \tag{5.3.43}$$

and

$$\begin{aligned}\mathring{j}(;\bullet) &= \left.\frac{\mathrm{d}[J\circ f\circ(h,k)]}{\mathrm{d}m}\right|_m \cdot \bullet \\ &= \left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v=f|_{u=h|_m, g=k|_m}} \cdot \left.\frac{\mathrm{d}[f\circ(h,k)]}{\mathrm{d}m}\right|_m \cdot \bullet \\ &= \left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v=f|_{u=h|_m, g=k|_m}} \cdot \mathring{v}(;\bullet).\end{aligned} \tag{5.3.44}$$

The practical introduction of intermediate variables for a given seed is then

$$\dot{m} = \left.\frac{\mathrm{d}m}{\mathrm{d}m}\right|_m \cdot \dot{m}(= I \cdot \dot{m}) \in M \text{ for given } m \in M \tag{5.3.45}$$

$$\dot{u} = \left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_m \cdot \dot{m} \in U \text{ for given } m \in M \tag{5.3.46}$$

$$\dot{g} = \left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_m \cdot \dot{m} \in G \text{ for given } m \in M \tag{5.3.47}$$

$$\dot{v} = \left.\frac{\partial f}{\partial u}\right|_{u=h|_m, g=k|_m} \cdot \dot{u} + \left.\frac{\partial f}{\partial u}\right|_{u=h|_m, g=k|_m} \cdot \dot{g} \in V \text{ for given } m \in M \tag{5.3.48}$$

$$\dot{j} = \left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v=f|_{u=h|_m, g=k|_m}} \cdot \dot{v} \in \mathbb{R} \text{ for given } m \in M. \tag{5.3.49}$$

5.3.2 Adjoint Approach

Instead of calculating the derivative of each operator to the control m, we will instead investigate the derivative of our output (here j) to our *intermediate* variables (v, u and g) as well as, ultimately, m. Let us consider then the following partial derivative 'Jacobian' linear operators, each taken with respect to our intermediate variables:

$$\begin{aligned}
\hat{m}(;\bullet) &= \partial[J \circ f \circ (h,k)]_m(m;\bullet) : M \to \mathbb{R} \text{ for given } m \in M, && (5.3.50)\\
\hat{g}(;\bullet) &= \partial[J \circ f \circ (h,k)]_g(u,g;\bullet) : G \to \mathbb{R} \text{ for given } u \in U, g \in G, && (5.3.51)\\
\hat{u}(;\bullet) &= \partial[J \circ f \circ (h,k)]_u(u,g;\bullet) : U \to \mathbb{R} \text{ for given } u \in U, g \in G, && (5.3.52)\\
\hat{v}(;\bullet) &= \partial[J \circ f \circ (h,k)]_v(v;\bullet) : V \to \mathbb{R} \text{ for given } v \in V, && (5.3.53)\\
\hat{j}(;\bullet) &= \partial[J \circ f \circ (h,k)]_j(j;\bullet) = I(;\bullet) : \mathbb{R} \to \mathbb{R} \text{ for given } j \in \mathbb{R}. && (5.3.54)
\end{aligned}$$

where I is again an identity operator which performs no operation, here equal to 1 (i.e. an identity matrix of $\text{span}(\mathbb{R}) \times \text{span}(\mathbb{R}) = 1 \times 1$ dimensions) and $\hat{m}$ is now the Jacobian operator with respect to m which we aim to find (here the partial derivative of $J \circ f \circ (h,k)$ is equal to the total derivative). We will refer to these as 'hat' operators.

When we do a calculation of J for given m (run our 'forward model') and therefore build our DAG, we are able to note which downstream operators our variables are sensitive to and find that

$$\begin{aligned}
\hat{g}(;\bullet) &= \partial[J \circ f \circ (h,k)]_g(u,g;\bullet) = \partial[J \circ f]_g(u,g;\bullet), && (5.3.55)\\
\hat{u}(;\bullet) &= \partial[J \circ f \circ (h,k)]_u(u,g;\bullet) = \partial[J \circ f]_u(u,g;\bullet), && (5.3.56)\\
\hat{v}(;\bullet) &= \partial[J \circ f \circ (h,k)]_v(v;\bullet) = \partial J_v(v;\bullet). && (5.3.57)
\end{aligned}$$

As with the tangent linear approach, we find that we can calculate each linear operator from others, though here we start from the *end* of the calculation (i.e. as far downstream as we can get, which means traversing the DAG backwards) and have to use concrete values for the direction vectors to have the correct operator ranges. We first see that

$$\begin{aligned}
\hat{v}(;v') &= \partial J_v(v;v') \in \mathbb{R}\\
&= \hat{j}\big(;\partial J_v(v;v')\big)
\end{aligned} \qquad (5.3.58)$$

then apply the chain rule (again immediately swapping partial derivatives for ordinary where they are equal) to find

$$\begin{aligned}
\hat{u}(;u') &= \partial[J \circ f]_u(u,g;u')\\
&= dJ_v\big(v = f|_{u,g}; v' = \partial f_u(u,g;u')\big)\\
&= \hat{v}|_{v=f|_{u,g}}\big(;\partial f_u(u,g;u')\big) \in \mathbb{R},
\end{aligned} \qquad (5.3.59)$$

$$\begin{aligned}
\hat{g}(;g') &= \partial[J \circ f]_g(u,g;g')\\
&= dJ_v\big(v = f|_{u,g}; v' = \partial f_g(u,g;g')\big)\\
&= \hat{v}|_{v=f|_{u,g}}\big(;\partial f_g(u,g;g')\big) \in \mathbb{R},
\end{aligned} \qquad (5.3.60)$$

and

$$\begin{aligned}
\hat{m}(;m') &= \partial[J \circ f \circ (h,k)]_m(m;m') \\
&= dJ_v\big(v = f|_{u,g}; d[f \circ (h,k)]_m(m;m')\big) \\
&= \hat{v}|_{v=f|_{u,g}}\big(; d[f \circ (h,k)]_m(m;m')\big) \\
&= \hat{v}|_{v=f|_{u,g}}\Big(; \big(\partial f_u\big(u = h|_m, g = k|_m; dh_m(m;m')\big) + \partial f_g\big(u = h|_m, g = k|_m; dk_m(m;m')\big)\big)\Big) \\
&= \hat{u}|_{u=h|_m,g=k|_m}\Big(; dh_m(m;m')\Big) + \hat{g}|_{u=h|_m,g=k|_m}\Big(; dk_m(m;m')\Big) \in \mathbb{R}.
\end{aligned} \tag{5.3.61}$$

We see a pattern here: each 'hat' operator is built from the sum of immediately downstream 'hat' operators, each operating on a corresponding downstream Jacobian.

Our aim here is to be able to seed with a direction vector from the downstream-most function $j' \in \mathbb{R}$ upstream such that we, eventually, get the corresponding direction vector $m' \in M$. I.e. for each 'hat' Jacobian operator we need an equivalent operator that takes the downstream direction vector and tells us what to pass upstream. This is generally done in one of two subtly different ways each of which rely on slightly different definitions of the adjoint of a linear operator. Consider a linear operator on Hilbert spaces $A : U \to V$ with inner products $\langle \bullet, \bullet \rangle_U$ and $\langle \bullet, \bullet \rangle_V$

1. the first adjoint definition is the linear operator

$$A^\dagger : V \to U \tag{5.3.62}$$

   which satisfies

$$\langle A(;u), v \rangle_V = \langle u, A^\dagger(;v) \rangle_U \; \forall\, u \in U, v \in V. \tag{5.3.63}$$

2. The second adjoint definition is the one we already encountered in Eq. 2.1.2: it is the linear operator

$$A^* : V^* \to U^* \tag{5.3.64}$$

   which satisfies

$$v^*(;A(;u)) = A^*(;v^*)(;u) \; \forall\, v^* \in V^*, u \in U. \tag{5.3.65}$$

   Note in particular that A^* always corresponds to the application of a covector v^* to the operator A *after* it has been applied to its operand u:

$$A^*(;v^*)(;u') = v^*(;A(;u')) \; \forall\, v^* \in V^* \text{ for given } u' \in U. \tag{5.3.66}$$

The two definitions are linked: $A^\dagger$ is the same as A^* but with every input and output covector replaced by corresponding the vector yielded by the Riesz representer for the inner products $\langle \bullet, \bullet \rangle_U$ and $\langle \bullet, \bullet \rangle_V$. When dealing with Euclidean vectors with an l_2 inner product, as is usual in AD ($U = \mathbb{R}^n, V = \mathbb{R}^m$), A is a matrix and $A^\dagger = A^T$ (its transpose). Since A is a matrix we can consider it separately to its operand $u' \in \mathbb{R}^n$: since covectors are row vectors in $1 \times \mathbb{R}^n$ we see that $A^*(;\bullet) = \bullet^T \cdot A$.

Each type of adjoint can be used to perform adjoint mode AD by either

1. taking the $\bullet^\dagger$ adjoint of each intermediate Jacobian to which we feed, firstly, a primal upstream direction vector j' which gives us the next direction vector v' and so on, or

2. taking the $\bullet^*$ adjoint of each intermediate Jacobian, to which we feed the covector j'^* (which corresponds to j' for the inner product that defines the adjoint) which gives us v'^* (which corresponds to v') and so on.

The difference between these approaches is rarely highlighted in the AD literature since we traditionally consider maps between $\mathbb{R}^n$ and $\mathbb{R}^m$ where the relationship between vectors and covectors is so apparently trivial due to the use of the l_2 inner product. Either one transposes the Jacobian matrices and multiplies on the right, or one transposes the vector and multiplies on the left. The $A^\dagger$ adjoint turns out to be highly impractical when not in this scenario (see the end of Sect. 5.4.1). Here, adjoint mode AD is therefore only derived with the definition of the adjoint corresponding to A^* (a derivation using $A^\dagger$ can be found in appendix A.2.3).

In each case we need to look at our intermediate Jacobians evaluated at given points:

$$dJ_v(v;\bullet) : V \to \mathbb{R} \text{ for given } v \in V, \tag{5.3.67}$$
$$\partial f_u(u,g;\bullet) : U \to V \text{ for given } u \in U, g \in G, \tag{5.3.68}$$
$$\partial f_g(u,g;\bullet) : G \to V \text{ for given } u \in U, g \in G, \tag{5.3.69}$$
$$dh_m(m;\bullet) : M \to U \text{ for given } m \in M, \tag{5.3.70}$$
$$dk_m(m;\bullet) : M \to G \text{ for given } m \in M. \tag{5.3.71}$$

The adjoints of these, with respect to the inner products that defines the dual spaces, are

$$dJ_v^*(v;\bullet) : \mathbb{R}^* \to V^* \text{ for given } v \in V, \tag{5.3.72}$$
$$\partial f_u^*(u,g;\bullet) : V^* \to U^* \text{ for given } u \in U, g \in G, \tag{5.3.73}$$
$$\partial f_g^*(u,g;\bullet) : V^* \to G^* \text{ for given } u \in U, g \in G, \tag{5.3.74}$$
$$dh_m^*(m;\bullet) : U^* \to M^* \text{ for given } m \in M, \tag{5.3.75}$$
$$dk_m^*(m;\bullet) : G^* \to M^* \text{ for given } m \in M. \tag{5.3.76}$$

We can use these in the adjoints of our ‘hat’ operators (taken with respect to the same inner products) which are adjoint Jacobians operating on upstream adjoint Jacobians

$$\hat{j}^*(;\bullet) = I^*(;\bullet) = I(;\bullet) : \mathbb{R}^* \to \mathbb{R}^* \quad \forall j \in \mathbb{R} \tag{5.3.77}$$

$$\hat{v}^*(;\bullet) = \partial J_v^*\Big(v; \hat{j}^*(;\bullet)\Big) : \mathbb{R}^* \to V^* \text{ for given } v \in V, \tag{5.3.78}$$

$$\hat{u}^*(;\bullet) = \partial f_u^*\Big(u, g; \hat{v}|_{v=f|_{u,g}}^*(;\bullet)\Big) : \mathbb{R}^* \to U^* \text{ for given } u \in U, g \in G, \tag{5.3.79}$$

$$\hat{g}^*(;\bullet) = \partial f_g^*\Big(u, g; \hat{v}|_{v=f|_{u,g}}^*(;\bullet)\Big) : \mathbb{R}^* \to G^* \text{ for given } u \in U, g \in G, \tag{5.3.80}$$

$$\hat{m}^*(;\bullet) = dh_m^*\Big(m; \hat{u}|_{u=h|_m, g=k|_m}^*(;\bullet)\Big) + dk_m^*\Big(m; \hat{g}|_{u=h|_m, g=k|_m}^*(;\bullet)\Big) : \mathbb{R}^* \to M^* \text{ for given } m \in M. \tag{5.3.81}$$

The full adjoint Jacobian for our example is $\hat{m}^*(;\bullet)$. Supplying any of these with a direction vector $j'^* \in \mathbb{R}$ (recalling that $\mathbb{R}^* = \mathbb{R}$) gives an adjoint Jacobian vector product. To recover the Jacobian operator for our example we supply a seed of $j'^* = 1$ to $\hat{m}^*$ and take the adjoint of $\hat{m}^*(;1)$: i.e.

$$[\hat{m}^*(; j'^* = 1)]^* \cdot \bullet = d[J \circ f \circ (h, k)]_m(m; \bullet). \tag{5.3.82}$$

In general, were $J : V \to X$ and we had a complete basis for X then, by operating on each basis covector in turn, we would be able to recover the complete adjoint-Jacobian

In practice, we decompose our adjoint 'hat' operators into intermediate operations by substituting intermediate adjoint-Jacobian vector products ($\bar{v}^* \in V^*$, $\bar{u}^* \in U^*$ etc.) for the adjoint of each 'hat' linear operator ($\hat{v}^*(;\bullet)$, $\hat{u}^*(;\bullet)$ etc.):

$$\bar{j}^* = I^*(;\bar{j}^*) = I(;\bar{j}^*) \in \mathbb{R}^*(= \mathbb{R}) \tag{5.3.83}$$

$$\bar{v}^* = \partial J_v^*(v; \bar{j}^*) \in V^* \text{ for given } v \in V, \tag{5.3.84}$$

$$\bar{u}^* = \partial f_u^*(u, g; \bar{v}^*) \in U^* \text{ for given } u \in U, g \in G, \tag{5.3.85}$$

$$\bar{g}^* = \partial f_g^*(u, g; \bar{v}^*) \in G^* \text{ for given } u \in U, g \in G, \tag{5.3.86}$$

$$\bar{m}^* = dh_m^*(m; \bar{u}^*) + dk_m^*(m; \bar{g}^*)(= \hat{m}^*(;\bar{j}^*)) \in M^* \text{ for given } m \in M. \tag{5.3.87}$$

These are referred to as 'adjoint variables'. Alongside equations 5.3.72 to 5.3.76 these form a DAG which is shown pictorially in figure 5.3.

Reverse/adjoint mode requires a full forward run of the calculation we are doing. This is needed to (a) compute intermediate outputs, for example to supply the correct $v \in V$ when calculating a particular adjoint variable $\bar{v}$, and to (b) record downstream dependencies in a DAG. In general, the set of intermediate variables (a) is usually referred to as the *tape*. Here (b) has been implicitly expressed by the way we have been able to simplify our 'hat' linear operators (equations 5.3.55 to 5.3.57) before applying the chain rule.

In summary, to recover the Jacobian operator for some function we have to traverse the DAG that describes the function once to find and save all the intermediate variables on the tape, then traverse the DAG *in reverse* once for each dimension of the output space. Since our

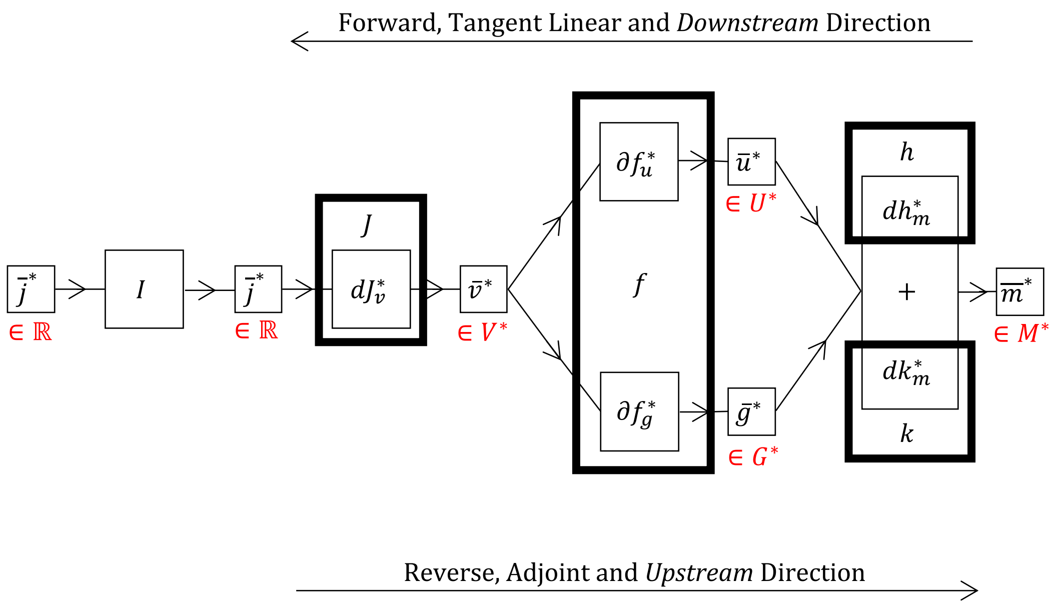


Figure 5.3: Directed Acyclic Graph (DAG) for the operations given in equations 5.3.72 to 5.3.76; linked by the intermediate output variables in equations 5.3.83 to 5.3.87. Functions are in large boxes with input and output variables directly connected. In practice the initial identity operation is skipped. The *blocks* associated with each operation in pyadjoint which annotate each operation and allow derivatives to be calculated (see section 5.4) are shown as thick boxes. The vector spaces that variables are members of are shown below each set in red. This DAG is very similar to the original example and TLM DAGs (figures 5.1 and 5.2 respectively) but with arrow directions reversed. Once again we see that for each variable we have a corresponding adjoint variable. Operations are now Jacobian evaluations although not all the blocks have arrows pointing towards them which makes implementing this difficult. How this is fixed is discussed and fixed in section A.2.5.

output space here is $\mathbb{R}$ this requires just a single reverse traversal! This is the optimal case for adjoint mode AD.

In general, if we were finding a Jacobian for a function $F : \mathbb{R}^m \to \mathbb{R}^n$ (which would be an $m \times n$ Jacobian matrix) then the time to compute it would scale as $m \cdot c \cdot \text{operations}(F)$ where c is a constant [56]. So where we we have $n > m$ a tangent linear approach is better whilst where we have $n < m$ (as is the case when optimising a functional where $m = 1$) an adjoint approach is better.

In our alternative notation the 'hat' Jacobian operators are

$$\hat{m}(;\bullet) = \left.\frac{\partial[J \circ f \circ (h,k)]}{\partial m}\right|_m \cdot \bullet : M \to \mathbb{R} \text{ for given } m \in M, \tag{5.3.88}$$

$$\hat{g}(;\bullet) = \left.\frac{\partial[J \circ f \circ (h,k)]}{\partial g}\right|_{u,g} \cdot \bullet : G \to \mathbb{R} \text{ for given } u \in U, g \in G, \tag{5.3.89}$$

$$\hat{u}(;\bullet) = \left.\frac{\partial[J \circ f \circ (h,k)]}{\partial u}\right|_{u,g} \cdot \bullet : U \to \mathbb{R} \text{ for given } u \in U, g \in G, \tag{5.3.90}$$

$$\hat{v}(;\bullet) = \left.\frac{\partial[J \circ f \circ (h,k)]}{\partial v}\right|_v \cdot \bullet : V \to \mathbb{R} \text{ for given } v \in V, \tag{5.3.91}$$

$$\hat{j}(;\bullet) = \left.\frac{\partial[J \circ f \circ (h,k)]}{\partial j}\right|_j \cdot \bullet = I(;\bullet) : \mathbb{R} \to \mathbb{R} \text{ for given } j \in \mathbb{R} \tag{5.3.92}$$

Noting which downstream operators our variables are sensitive to gives us

$$\hat{g}(;\bullet) = \left.\frac{\partial[J \circ f \circ (h,k)]}{\partial g}\right|_{u,g} \cdot \bullet = \left.\frac{\partial[J \circ f]}{\partial g}\right|_{u,g} \cdot \bullet, \tag{5.3.93}$$

$$\hat{u}(;\bullet) = \left.\frac{\partial[J \circ f \circ (h,k)]}{\partial u}\right|_{u,g} \cdot \bullet = \left.\frac{\partial[J \circ f]}{\partial u}\right|_{u,g} \cdot \bullet, \tag{5.3.94}$$

$$\hat{v}(;\bullet) = \left.\frac{\partial[J \circ f \circ (h,k)]}{\partial v}\right|_v \cdot \bullet = \left.\frac{\partial J}{\partial v}\right|_v \cdot \bullet. \tag{5.3.95}$$

$$\tag{5.3.96}$$

We now see that

$$\begin{aligned} \hat{v}(;v') &= \left.\frac{\partial J}{\partial v}\right|_v \cdot v' \\ &= \hat{j} \cdot \left.\frac{\partial J}{\partial v}\right|_v \cdot v' \end{aligned} \tag{5.3.97}$$

then apply the chain rule to find

$$\begin{aligned}\hat{u}(;u') &= \left.\frac{\partial[J \circ f]}{\partial u}\right|_{u,g} \cdot u' \\ &= \left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v=f|_{u,g}} \cdot \frac{\partial f}{\partial u}|_{u,g} \cdot u' \\ &= \hat{v}|_{v=f|_{u,g}} \cdot \frac{\partial f}{\partial u}|_{u,g} \cdot u',\end{aligned} \tag{5.3.98}$$

$$\begin{aligned}\hat{g}(;g') &= \left.\frac{\partial[J \circ f]}{\partial g}\right|_{u,g} \cdot g' \\ &= \left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v=f|_{u,g}} \cdot \frac{\partial f}{\partial g}|_{u,g} \cdot g' \\ &= \hat{v}|_{v=f|_{u,g}} \cdot \frac{\partial f}{\partial g}|_{u,g} \cdot g',\end{aligned} \tag{5.3.99}$$

and

$$\begin{aligned}\hat{m}(;m') &= \left.\frac{\partial[J \circ f \circ (h,k)]}{\partial m}\right|_{m} \cdot m' \\ &= \left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v=f|_{u,g}} \cdot \left.\frac{\mathrm{d}[f \circ (h,k)]}{\mathrm{d}m}\right|_{m} \cdot m' \\ &= \hat{v}|_{v=f|_{u,g}} \cdot \left(\left.\frac{\partial f}{\partial u}\right|_{u=h|_m, g=k|_m} \cdot \frac{\mathrm{d}h}{\mathrm{d}m}|_m \cdot m' + \left.\frac{\partial f}{\partial g}\right|_{u=h|_m, g=k|_m} \cdot \left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_m \cdot m' \right) \\ &= \hat{u}|_{u=h|_m, g=k|_m} \cdot \left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_m \cdot m' + \hat{g}|_{u=h|_m, g=k|_m} \cdot \left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_m \cdot m'.\end{aligned} \tag{5.3.100}$$

The intermediate Jacobians evaluated at given points are

$$\left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v} \cdot \bullet : V \to \mathbb{R} \text{ for given } v \in V, \tag{5.3.101}$$

$$\left.\frac{\partial f}{\partial u}\right|_{u,g} \cdot \bullet : U \to V \text{ for given } u \in U, g \in G, \tag{5.3.102}$$

$$\left.\frac{\partial f}{\partial g}\right|_{u,g} \cdot \bullet : G \to V \text{ for given } u \in U, g \in G, \tag{5.3.103}$$

$$\left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_{m} \cdot \bullet : M \to U \text{ for given } m \in M, \tag{5.3.104}$$

$$\left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_{m} \cdot \bullet : M \to G \text{ for given } m \in M. \tag{5.3.105}$$

The adjoints of the intermediate Jacobians evaluated at given points are

$$\left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v}^{*} \cdot \bullet : \mathbb{R}^* \to V^* \text{ for given } v \in V, \tag{5.3.106}$$

$$\left.\frac{\partial f}{\partial u}\right|_{u,g}^{*} \cdot \bullet : V^* \to U^* \text{ for given } u \in U, g \in G, \tag{5.3.107}$$

$$\left.\frac{\partial f}{\partial g}\right|_{u,g}^{*} \cdot \bullet : V^* \to G^* \text{ for given } u \in U, g \in G, \tag{5.3.108}$$

$$\left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_{m}^{*} \cdot \bullet : U^* \to M^* \text{ for given } m \in M, \tag{5.3.109}$$

$$\left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_{m}^{*} \cdot \bullet : G^* \to M^* \text{ for given } m \in M. \tag{5.3.110}$$

The adjoints of our 'hat' operators are

$$\hat{j}^*(;\bullet) = I^*(;\bullet) = I(;\bullet) : \mathbb{R}^* \to \mathbb{R}^* \quad \forall j \in \mathbb{R} \tag{5.3.111}$$

$$\hat{v}^*(;\bullet) = \left.\frac{\partial J}{\partial v}\right|_{v}^{*} \cdot \hat{j}^*(;\bullet) : \mathbb{R}^* \to V^* \text{ for given } v \in V, \tag{5.3.112}$$

$$\hat{u}^*(;\bullet) = \left.\frac{\partial f}{\partial u}\right|_{u,g}^{*} \cdot \hat{v}|_{v=f|_{u,g}}^{*}(;\bullet) : \mathbb{R}^* \to U^* \text{ for given } u \in U, g \in G, \tag{5.3.113}$$

$$\hat{g}^*(;\bullet) = \left.\frac{\partial f}{\partial g}\right|_{u,g}^{*} \cdot \hat{v}|_{v=f|_{u,g}}^{*}(;\bullet) : \mathbb{R}^* \to G^* \text{ for given } u \in U, g \in G, \tag{5.3.114}$$

$$\hat{m}^*(;\bullet) = \left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_{m}^{*} \cdot \hat{u}|_{u=h|_m, g=k|_m}^{*}(;\bullet) + \left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_{m}^{*} \cdot \hat{g}|_{u=h|_m, g=k|_m}^{*}(;\bullet) : \mathbb{R}^* \to M^* \text{ for given } m \in M, \tag{5.3.115}$$

we get the full Jacobian for example by

$$[\hat{m}^*(; j'^* = 1)]^* \cdot \bullet = \left.\frac{\mathrm{d}[J \circ f \circ (h,k)]}{\mathrm{d}m}\right|_{m} \cdot \bullet \tag{5.3.116}$$

and the intermediate covectors adjoint variables for downstream-most seed covector $\bar{j}^*$ (which we use to calculate the full Jacobian) are

$$\bar{j}^* = I(;\bar{j}) \in \mathbb{R}^* \tag{5.3.117}$$

$$\bar{v}^* = \left.\frac{\partial J}{\partial v}\right|_v^* \cdot \bar{j}^* \in V* \text{ for given } v \in V, \tag{5.3.118}$$

$$\bar{u}^* = \left.\frac{\partial f}{\partial u}\right|_{u,g}^* \cdot \bar{v}^* \in U^* \text{ for given } u \in U, g \in G, \tag{5.3.119}$$

$$\bar{g}^* = \left.\frac{\partial f}{\partial g}\right|_{u,g}^* \cdot \bar{v}^* \in G^* \text{ for given } u \in U, g \in G, \tag{5.3.120}$$

$$\bar{m}^* = \left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_m^* \cdot \bar{u}^* + \left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_m^* \cdot \bar{g}^* \ (= \hat{m}^*(;\bar{j}^*)) \in M^* \text{ for given } m \in M. \tag{5.3.121}$$

5.3.3 Higher Derivatives

Gradient descent algorithms, which find function minima by moving in the direction of steepest gradient, are a key motivating factor for the development of AD. The standard gradient descent method for a function f with a single parameter $x \in \mathbb{R}$ is found by iteratively evaluating

$$x_{k+1} = x_k - \alpha \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_k} \tag{5.3.122}$$

with f until

$$f(x_{k+1}) \approx f(x_k) \tag{5.3.123}$$

for some sufficiently small α. Gradient descent suffers from difficulty in finding an appropriately small α (though methods exist for finding a step-specific α_k - see section 1.5.1 of Schwedes et al. [29]) and the direction of steepest gradient not necessarily pointing directly towards the minimum at each step. The latter can cause a 'zig-zagging' within the solution space which increases the number of steps needed to find the minimum.

Having access to a second derivatives of the function we wish to minimise, assuming the function is twice differentiable, gives access to many more optimisation algorithms which can be much more computationally efficient (i.e. take less time). The simplest example is to apply Newton's method to find where the first derivative of f is zero. This is done by iteratively evaluating

$$x_{k+1} = x_k - h \tag{5.3.124}$$

where

$$h = \left[\left.\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right|_{x_k}\right]^{-1} \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_{x_k} \tag{5.3.125}$$

is the so-called 'Newton direction'. This is derived by taking the Taylor expansion approximation of f

$$f(x_k + \Delta x) \approx f(x_k) + \left.\frac{\mathrm{d}f}{\mathrm{d}x}\right|_x \Delta x + \frac{1}{2!}\left.\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\right|_x \Delta x^2 \tag{5.3.126}$$

where $\Delta x = x_{k+1} - x_k$ and setting it's first derivative with respect to Δx equal to zero. If we have a quadratic function then this becomes an exact equality and we converge in a single step. Note that the above expression is for $x, f(x) \in \mathbb{R}$: in the general case the expression is more complicated, for example finding an equivalent to Δx^2. Where we cannot or do not wish to calculate the second derivative, there exist various quasi-Newton methods which approximate the second derivative such as the Broyden-Fletcher-Goldfarb-Shanno (BFGS) method (see section 1.5.4 of Schwedes et al. [29]).

Equivalents to both gradient descent and Newton's method exist for functions in vector function spaces and therefore multivariate functions. For examples see section 1.5 of Schwedes et al. [29]. Note that in general the Newton direction h is found by solving $\left[\frac{\mathrm{d}^2 f}{\mathrm{d}x^2}\Big|_{x_k}\right]^{-1} h = \frac{\mathrm{d}f}{\mathrm{d}x}\Big|_{x_k}$ rather than computing an operator inverse which is generally computationally expensive.

The general mixed second derivative for a function

$$f : U \times G \to V \text{ i.e. } f(u, g) \in V \;\forall\, u \in U, g \in G, \tag{5.3.127}$$

which we assume can be multiply differentiated, is

$$\partial\big[\partial f_u\big]_g : U \times G \times \underbrace{U}_{\text{linear}} \times \underbrace{G}_{\text{linear}} \to V \text{ i.e. } \partial\big[\partial f_u\big]_g(u, g; u', g') \in V \;\forall\, u, u' \in U, g, g' \in G \tag{5.3.128}$$

$$:= \frac{\partial^2 f}{\partial u\, \partial g}\bigg|_{u,g} \cdot u' \cdot g'. \tag{5.3.129}$$

For a given u and g but unknown u' and g' this is a linear operator of two variables (i.e. a rank 3 tensor).

Two partial derivatives with respect to the same variable (assuming our function is twice appropriately differentiable) is the operator

$$\partial\big[\partial f_u\big]_u : U \times G \times \underbrace{U}_{\text{linear}} \times \underbrace{U}_{\text{linear}} \to V \text{ i.e. } \partial\big[\partial f_u\big]_u(u, g; u', u'') \in V \;\forall\, u, u', u'' \in U, g \in G \tag{5.3.130}$$

$$:= \frac{\partial^2 f}{\partial u^2}\bigg|_{u,g} \cdot u' \cdot u'' \tag{5.3.131}$$

which is usually referred to as the *Hessian* of f with respect to u. Note that, as with the Jacobian, this should not be confused with the 'Hessian Matrix' which is once again the expansion of generalised Hessian in a particular finite dimensional basis.

We can find higher derivatives by applying tangent linear (forward) and adjoint (reverse) modes more than once giving schemes such as *forward-over-forward*, *reverse-over-reverse*, *reverse-over-forward* and *forward-over-reverse* for finding second derivatives. The naming scheme goes "second-over-first" which correspond to the order in which the schemes are applied. A note on naming: Forward-over-forward is the *second order tangent linear* mode which can be calculated alongside each step in the original calculation whilst the other three are *second order adjoint* modes which require the creation of tapes. This naming can be extended to give schemes like "forward-over-reverse-over-forward" for finding a third derivative. As with various topics in AD these schemes seem to have been discovered multiple times in different fields, for reverse-over-forward see for example the mathematics outlined by Christianson [57] in numerical analysis and Pearlmutter [58] in neural networks.

Without supplied directions a second derivative can be thought of as a Jacobian of a Jacobian. With finite computing resources and discretised vector spaces, these are typically very large data structures which are not desirable to store. Thankfully we can use AD's inherent property of performing Jacobian-vector and adjoint-Jacobian-vector products to our advantage. Forward-over-forward requires two TLM seed vectors and produces a second-derivative-vector-vector product (e.g. a Hessian-vector-vector product) such as

$$\partial\big[\partial f_u\big]_g(u, g; u', g') \tag{5.3.132}$$

Forward-over-reverse and reverse-over-forward both produce a second-derivative-vector product operator (e.g. a Hessian-vector product) such as

$$\begin{aligned}\partial\big[\partial f_u(u, g; u')\big]_u(u, g; \bullet) &= \partial\big[\partial f_u\big]_u(u, g; u', \bullet) && (5.3.133)\\ &:= \frac{\partial^2 f}{\partial u^2}\bigg|_{u,g} \cdot u' \cdot \bullet : U \to V \text{ for given } u, u' \in U, g \in G && (5.3.134)\end{aligned}$$

without ever having to store any complete Jacobians of Jacobians. Reverse-over-reverse is rarely used.

We take forward-over-reverse as our example with regards to our function f: this can be thought of as

1. taping the entire process finding the Jacobian operator df_u or df_g (depending on which second derivative we want) via the adjoint mode:
   (a) that includes taping the calculation of f for given u and g and
   (b) playing the tape in reverse, seeded with as many basis vectors as needed to calculate the Jacobian operator.
2. Then performing TLM AD on the long tape which we replay forward, seeding it with the vector u' or g' (depending on which first derivative we want).
   (a) Since the beginning of this tape is the calculation of f this calculates the initial $\partial f_u(u, g; u')$ or $\partial f_g(u, g; g')$ needed (see equation 5.3.133) whilst
   (b) at the adjoint mode section of the long tape the second derivative is found by traversing the shorter tape of f in the reverse/adjoint/upstream direction.

In practice forward-over-reverse is rarely implemented like this where one takes tapes of tapes. Instead books such as chapter 3 of Naumann [51] provide recipes for calculating higher derivatives: forward-over-reverse is described by equation 3.8. The mathematics of this is covered in more detail in sections 5.4.3 and 5.5.2.

5.4 Dolfin-Adjoint and Pyadjoint

Dolfin-adjoint [59] was a tool for solving PDE constrained optimisation problems using the FEniCS system [17] via the *adjoint method*, a generalised term for the adjoint mode of AD.[4] These are problems of the form 'given some PDE, solve an optimisation problem for a function involving the PDE solution', similar to the example given in the introduction to AD. Dolfin-adjoint limited itself to functions which map to $\mathbb{R}$ (functionals).

Much could be written about how to go about solving such problems, including the transformation of the given unconstrained problem to a constrained one, and the creation of tangent linear and adjoint systems from the original PDE discretisation. Such a discussion is lengthy and not important for the discussion here but is very interesting. For that, attention is drawn to the wide ranging briefing paper by Schwedes et al. [29], the original dolfin-adjoint paper Farrell et al. [59], and the dolfin-adjoint documentation[5].

In 2018 dolfin-adjoint was rewritten in Python as *pyadjoint* [46].[6,7] Pyadjoint can be considered an operator overloaded AD tool, as described in Naumann [51] and in Sect. 5.3, for the Python language. In pyadjoint, fundamental operations to be differentiated are referred to as *blocks* with the process of creating a block (i.e. overloading the operation) being *annotation*. Types are augmented such that they can store an operation's intermediate input vectors and TLM vectors, or intermediate output vectors and adjoint mode covectors, by encapsulating them in a new 'block variable' type. Each block variable is able to encapsulate, in a single object, each of (a) an input/output variable, (b) if calculated, a TLM variable and (c) if calculated, an adjoint variable. These correspond to the linked variables in figure 5.1 with associated equations 5.3.5 to 5.3.8 (see figure 5.4)

On top of pyadjoint sit compatibility layers which provide support for the UFL domain specific language and Firedrake. The latter is known as `firedrake.adjoint`. In `firedrake.adjoint` blocks are provided to annotate high level operations such as solving a PDE and, as implemented here, interpolating from one finite element function space to another. A similar layer exists for FEniCS but is no longer maintained.

[4]The name dolfin-adjoint refers to the dolfin finite element library, which was historically part of the FEniCS system.

[5]Available at `https://www.dolfin-adjoint.org/en/latest/documentation/maths/`

[6]One can read about the motivation behind the development of pyadjoint in Sebastian Mitusch's MSc thesis, which is available at `https://www.duo.uio.no/bitstream/handle/10852/63505/SebastianMitusch-thesis.pdf`.

[7]Note: in literature, pyadjoint is often referred to as dolfin-adjoint/pyadjoint.

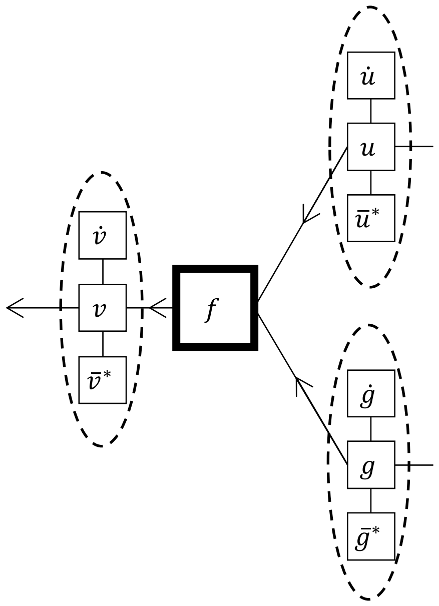


Figure 5.4: A close up of the DAG in figure 5.1 with the adjoint variables now changed to the corresponding covectors (pyadjoint uses method (2) in section 5.3.2). A block annotates the function f. Block variables contain each of (a) an input/output variable, (b) if calculated, a TLM variable (dotted) and (c) if calculated, an adjoint covector variable (barred and starred).

`firedrake.adjoint` is able to do this by finding the analytic expression for the necessary TLM and adjoint mode derivatives: since all our calculations are written in the near mathematical notation of UFL, which symbolically describes both the PDE and its discretisation, the derivative of these expressions can be found by applying the rules of differentiation to them. For a PDE in the adjoint mode this is known as finding the so-called *continuous adjoint*. Then, since we retain the information about the discretisation (symbolically in UFL and concretely in Firedrake), we can easily generate the code for the discretised derivative (in adjoint mode the *discrete adjoint*). This can be thought of as an AD approach which operates at the highest possible level of abstraction, thus avoiding most of the pitfalls found in other AD tools [60].[8]

The ability to easily generate discrete adjoints for PDE solves was a key part of the original success of dolfin-adjoint[59], alongside being able to parallelise their solution, prior to its rewrite as pyadjoint. Before this one would have to analytically find the continuous adjoint for a finite element model, rediscretise it and re-code it up.

One of AD's raisons d'être is that it avoids both the need to find a continuous solution and that it avoids so-called 'expression swell' where the number of terms in the derivative becomes very large [50]. This is true here, though it should be noted that PDE discretisations with multiple time steps can have very long tapes.

[8]Perhaps the only pitfall Hückelheim et al. [60] describe which remains applicable to our approach is the use of iterative methods for solving PDEs (section 3.4). In principal this causes inaccurate derivatives to be calculated. It is also, of course, worth remembering that we are not solving the exact mathematical problems we describe but rather our chosen numerical approximation of it.

5.4.1 Discretisation and the Impact on Covectors and Adjoints

In the finite element method, forward[9] block variables (such as u and $\dot{u}$) are vectors or covectors[10] in a Hilbert space which are represented as coefficients of basis functions or cofunctions. How many basis functions or cofunctions we have, and what these are, is the 'discretisation' of our problem. As mentioned previously, the specific discretisation depends on the mesh and choice of function space. In Firedrake and FEniCS specific instances of these variables are stored as the coefficients of the basis functions: in Firedrake this is stored in the `Function` or `Cofunction` class. For the rest of this discussion, forward block variables will only be considered as primal functions, but this is purely for clarity of the exposition as opposed to being a limitation of the software.

In section 1.4, we demonstrated that covectors to vectors in finite element function spaces can be represented as the transpose of the coefficients vector u multiplied by the appropriate inner product matrix

$$u^*_{1j} = \sum_{i}^{\dim(U)} u_{1i} M^U_{ij}. \tag{5.4.1}$$

(equation 1.4.39) which gives us a discrete way of storing covectors as row vectors, with the j^{th} entry of the row vector being the coefficient for the j^{th} dual basis function (see Sect. 2.3). At the time of writing, `firedrake.adjoint` makes use of `Function` for storing vectors and the new `Cofunction` type for storing covectors.[11]

Returning to section 1.4 again, we also showed the action of a covector is

$$u^*(;\bullet) = \sum_{ij}^{\dim(U)} u_{1i} M^U_{ij} \cdot [\bullet]_j \tag{5.4.2}$$

in equation 1.4.38 where M^U is some inner product matrix for the finite element function space U. This is useful for working out exactly what our adjoint operations are. Consider a linear operator between finite element function spaces

$$B : U \to M. \tag{5.4.3}$$

We saw from the definition of B^* (equation 5.3.65) that it is equivalent to

$$B^*(;u^*)(;m') = u^*(;B(;m'))\ \forall\, u^* \in U^* \text{ for given } m' \in M \tag{5.4.4}$$

[9]The distinction between forward and reverse block variables is outlined in section A.2.5.

[10]The output of assembling a 1-linear form is a covector as discussed in Sect. 2.3.

[11]Prior to the addition of dual spaces to UFL (chapter 2), both covectors and vectors were stored in Firedrake as `Functions`: Firedrake did not have any way of knowing if the data stored represented a vector or a covector. `firedrake.adjoint` got around this by treating Firedrake `Functions` as vectors and covectors as (confusingly) `dolfin vectors` which is the type one gets when assembling a 1-linear form in `dolfin`, the codebase of FEniCS[17] with which Firedrake maintains some compatibility.

(equation 5.3.66 for B) i.e. the application of u^* to B after its application to m'. Members of U and M can be represented as vectors of weight coefficients, so B must be representable by a matrix B_{ij}.

$$B^*(;u^*)(;m') = \sum_{l}^{\dim(V)} \sum_{j}^{\dim(U)} u^*_{1j} B_{jl} m'_l \tag{5.4.5}$$

where, from Eq. 5.4.2

$$u^*_{1j} = \sum_{i}^{\dim(U)} u_{1i} M^U_{ij}. \tag{5.4.6}$$

This is a row-vector/matrix/vector product so

$$B^*(;u^*)(;\bullet) = u^* \cdot B \cdot \bullet \tag{5.4.7}$$

as is the case with linear operators between Euclidean vector spaces. The key, then, is to make sure our covectors are correctly represented with an appropriate inner product matrix to use adjoint mode AD.

We can see now why using the definition of the adjoint as B^*, rather than $B^\dagger$ is sensible: were B our Jacobian we don't need to transform it, just have the covector act on it. Furthermore, we don't have an obvious way of finding what $B^\dagger$ is without appealing to the Riesz representation theorem which would need us to transform our vectors to covectors and do the above calculation anyway.

5.4.2 Tangent Linear Model and Adjoint Action

We assume at this point that we already have a DAG for all our operations, and therefore know the TLM DAG (the same as that for the operation we are annotating) and Adjoint DAG (which goes in the opposite direction). Pyadjoint has a core `Block` class which we subclass for each operation we wish to annotate. This class implements two fundamental operations which allow for tangent linear and adjoint mode operation. The first is *Tangent Linear Model action* or 'TLM action' and the second is *Adjoint action.*

The TLM action for a block, calculated by an `evaluate_tlm` method, performs Jacobian-vector products using the relevant upstream TLM variables given by the TLM DAG (see equations 5.3.31 to 5.3.35. It retrieves the relevant TLM variables, sums them up, then attaches the output(s) to the block variable containing the function output(s). So for the block which annotates f we do

$$\dot{v} = \partial f_u(u = h|_m, g = k|_m; \dot{u}) + \partial f_g(u = h|_m, g = k|_m; \dot{g}) \tag{5.4.8}$$

$$:= \left.\frac{\partial f}{\partial u}\right|_{u=h|_m, g=k|_m} \cdot \dot{u} + \left.\frac{\partial f}{\partial g}\right|_{u=h|_m, g=k|_m} \cdot \dot{g}. \tag{5.4.9}$$

where $\dot{v}$ is the single new output TLM variable which is attached to the block variable for v (the output of f), whilst $\dot{u}$ and $\dot{g}$ are the upstream TLM variables. These calculations have to be implemented by us for a given block subclass in an `evaluate_tlm_component` method. The `evaluate_tlm` method is predefined to be the same for all blocks and takes care of the rare case where we have multiple block outputs: it inspects the DAG then loops through block outputs calling `evaluate_tlm_component` for each.

The adjoint action, calculated by an `evaluate_adjoint` method, performs the application of each block's adjoint of its Jacobian to the incoming downstream covector and adds the output to an appropriate upstream covector. For more on this see section A.2.5.

As with the TLM action, the `evaluate_adjoint` method is predefined to be the same for all blocks. For a block which annotates f there will be two operations to perform given a downstream covector $\bar{v}^*$:

$$\bar{u}^* = \partial f_u^*(u, g; \bar{v}^*) \in U^* \text{ for given } u \in U, g \in G \tag{5.4.10}$$

(equation 5.3.85) and

$$\bar{g}^* = \partial f_g^*(u, g; \bar{v}^*) \in G^* \text{ for given } u \in U, g \in G \tag{5.4.11}$$

(equation 5.3.86) each of which is done in an `evaluate_adj_component` method which we implement ourselves for a given block subclass. These then act as inputs to the h and k blocks whose `evaluate_adjoint` method calculates our final $\bar{m}^*$ (via equation 5.3.86).

Further discussion of the mathematics of the tape, forward and adjoint mode in pyadjoint and AD in general can be found in appendix Sect. A.2.5, 'Adjoint Mode Subtlety'.

5.4.3 Hessian Action

In pyadjoint the term *Hessian* is used to refer to any second derivative (i.e. Jacobian of a Jacobian) including the mixed case given in equation 5.3.128. Pyadjoint contains an implementation of equation 3.8 in Naumann [51] which describes how forward-over-reverse mode accumulates Hessian-vector products in the reverse/adjoint/upstream direction relative to our original calculation. Here we will describe the process that the `compute_hessian` function, found in pyadjoint's `driver.py` file, uses to calculate a Hessian-vector product.

One must first perform an adjoint mode calculation on a tape with a seed of 1 to get a Jacobian operator with respect to a control variable: this gives the second (outermost) derivative of our Hessian. Unfortunately `compute_hessian` does not document this requirement and will crash if it has not been done. Rather than taping the calculation of the adjoint, the adjoint-mode calculation merely serves to populate the reverse block variables with adjoint covectors: as we will see, these are needed to calculate each block's Hessian-vector products alongside the forward block variables from what was originally taped. Next a TLM seed vector for the first (innermost) derivative, which is an argument to `compute_hessian`, is supplied: we run a TLM calculation on the tape, which we have retrieved from the adjoint mode calculation, using the seed vector to populate the forward TLM variables which we also require. We lastly run backwards through the tape (in the reverse/adjoint/upstream direction) calculating the so-called *Hessian Action* in each block's `evaluate_hessian` method.

Returning to the example of equations 5.3.5 to 5.3.8 linked by the intermediate variables shown in equations 5.3.9 to 5.3.12 (see figure 5.1) we typically aim to find the operator

$$d\big[d[J \circ f \circ (h,k)]_m\big]_m(m;\dot{m},\bullet) : M \to \mathbb{R} \text{ for given } m, \dot{m} \in M \tag{5.4.12}$$
$$:= \frac{\mathrm{d}^2[J \circ f \circ (h,k)]}{\mathrm{d}m^2}\bigg|_m \cdot \dot{m} \cdot \bullet \tag{5.4.13}$$
$$\tag{5.4.14}$$

where $\dot{m} \in M$ is the TLM seed vector.

The Hessian action is the calculation of the product of a block's Hessian and TLM direction vector. At block f, the Hessian action, given a TLM direction vector $\dot{u} \in U$ or $\dot{g} \in G$, calculates any of

$$\partial\big[\partial[J \circ f]_u\big]_u(u,g;u' = \dot{u},\bullet) : U \to X \text{ for given } u, \dot{u} \in U, g \in G \tag{5.4.15}$$
$$:= \frac{\partial^2[J \circ f]}{\partial u^2}\bigg|_{u,g} \cdot \dot{u} \cdot \bullet, \tag{5.4.16}$$

$$\partial\big[\partial[J \circ f]_g\big]_g(u,g;g' = \dot{g},\bullet) : G \to X \text{ for given } u \in U, g, \dot{g} \in G \tag{5.4.17}$$
$$:= \frac{\partial^2[J \circ f]}{\partial g^2}\bigg|_{u,g} \cdot \dot{g} \cdot \bullet, \tag{5.4.18}$$

$$\partial\big[\partial[J \circ f]_u\big]_g(u,g;u' = \dot{u},\bullet) : G \to X \text{ for given } u, \dot{u} \in U, g \in G \tag{5.4.19}$$
$$:= \frac{\partial^2[J \circ f]}{\partial g\,\partial u}\bigg|_{u,g} \cdot \dot{u} \cdot \bullet, \tag{5.4.20}$$

and

$$\partial\big[\partial[J \circ f]_g\big]_u(u,g;g' = \dot{g},\bullet) : U \to X \text{ for given } u \in U, g, \dot{g} \in G \tag{5.4.21}$$
$$:= \frac{\partial^2[J \circ f]}{\partial u\,\partial g}\bigg|_{u,g} \cdot \dot{g} \cdot \bullet \tag{5.4.22}$$

depending on which TLM direction vector we supply and which terms are necessary for the upstream Hessian calculation.

The Hessian action works almost identically to the adjoint action (which makes sense given we are going in the reverse/adjoint direction). As with the adjoint action the f block receives downstream calculations of the Hessian action operators, in this case a single Hessian action on J given a TLM direction vector $\dot{v}$

$$\partial\big[\partial J_v\big]_v(v; v' = \dot{v}, \bullet) : V \to \mathbb{R} \text{ for given } v, v' = \dot{v} \in V \tag{5.4.23}$$

$$:= \left.\frac{\partial^2 J}{\partial v^2}\right|_v \cdot \dot{v} \cdot \bullet. \tag{5.4.24}$$

We decompose our Hessian action using the chain rule (which we will do later for the interpolate block) and see that, in general, we can use this Hessian action input operator, the adjoint action input covector (for blocks which annotate nonlinear functions) and the saved tangent linear mode vectors to calculate this block's Hessian action. Once again each block has an `evaluate_hessian_component` method which we define ourselves in a block subclass that takes a necessary subset of these inputs. We then sum the outputs of `evaluate_hessian_component` as necessary in the block's `evaluate_hessian` method to create Hessian action input operators for our upstream blocks.

5.5 Differentiating Dual Evaluation: Interpolate Block Annotation

Note that the following is adapted from the documentation, written by the author, for the `InterpolateBlock` API in `firedrake.adjoint`.

We consider an interpolate block as f with forward model input $u \in U$ and output $v \in V$ (there can, in principal, be any number of inputs or tensor valued output but we stick to the single variable case for clarity):

$$f : U \to V \text{ i.e. } f(u) \in V \; \forall \, u \in U. \tag{5.5.1}$$

We henceforth consider all function spaces to be finite dimensional (i.e. discretised) unless stated otherwise. Here we have

$$\dim(U) = m \tag{5.5.2}$$

$$\dim(V) = n \tag{5.5.3}$$

The interpolate block f encompasses both the dual evaluation interpolation operator $\mathcal{I}$, which is linear in its arguments, and evaluation of some expression 'expr', which may not be linear in its arguments (for example if $\text{expr}(u) = u^2$).This can be considered as follows:

$$f = \mathcal{I} \circ \text{expr} : U \to V \text{ i.e. } f(u) = \mathcal{I}(; \text{expr}(u)) \in V \ \forall u \in U, \tag{5.5.4}$$

$$\mathcal{I} : X \to V \text{ i.e. } \mathcal{I}(; x) \in V \ \forall x \in X, \tag{5.5.5}$$

$$\text{expr} : U \to X \text{ i.e. } \text{expr}(u) \in X \ \forall u \in U. \tag{5.5.6}$$

Note that we once again adopt the notation of linear arguments following a semicolon ';'. X is any space to which dual-evaluation interpolation can be validly applied (see Sect. 3.2). We never explicitly see X, though we know that expr must represent some function of U that produces values which can be dual evaluated by $\mathcal{I}$.

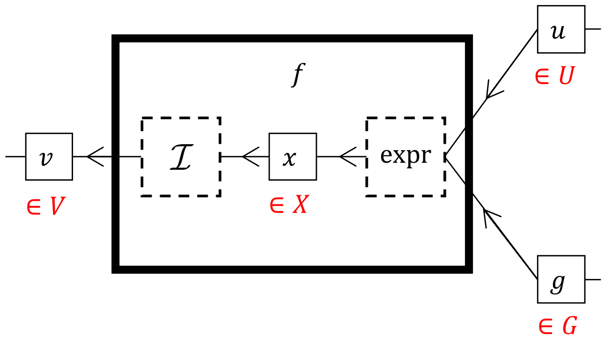


Figure 5.5: The f block can be decomposed into a linear dual evaluation interpolation operation $\mathcal{I}$ applied to an expression evaluation expr.

The interpolate block is implemented in pyadjoint as a subclass of the `Block` class named `InterpolateBlock`. Prior to this work, the `InterpolateBlock` already existed but the non-linearity of its arguments had not been fully considered and the Hessian Action had not been implemented. Adding the Hessian Action turned out to be far from trivial and required a full understanding of AD in the context of Gateaux derivatives and pyadjoint hence a chapter of this thesis being devoted to it.

5.5.1 Tangent Linear and Adjoint Action

It is first necessary to discuss the TLM and Adjoint action operations. The Gateaux derivative of the linear operator $\mathcal{I}$ in a direction $x' \in X$ is the same as $\mathcal{I}(; x')$:

$$\text{if } \mathcal{I}(; x) = Ax \tag{5.5.7}$$

$$\text{then } d\mathcal{I}_x(; x, x') = Ax' \tag{5.5.8}$$

$$\implies d\mathcal{I}_{x'}(; x|_{x'}, x') = Ax' = \mathcal{I}(; x') \tag{5.5.9}$$

where A is some linear operation. This is the case for any linear operator. In this case A is multiplication by the interpolation matrix. The TLM `evaluate_tlm_component` method for a single TLM variable $\dot{u} \in U$ is therefore

$$\begin{aligned} v'(;u' = \dot{u}) &= d[\mathcal{I} \circ \text{expr}]_u(u; u' = \dot{u}) \;\; \in V \text{ for given } u \in U && (5.5.10)\\ &= d\mathcal{I}_x(; x = \text{expr}|_u, x' = d\text{expr}_u(u; u' = \dot{u})) && (5.5.11)\\ &= \mathcal{I}\big(; d\text{expr}_u(u; u' = \dot{u})\big) && (5.5.12)\\ &:= \mathcal{I} \cdot \left.\frac{\text{dexpr}}{\text{d}u}\right|_u \cdot \dot{u} && (5.5.13) \end{aligned}$$

i.e. it is the interpolation of the directional/Gateaux derivative of our expression $d\text{expr}_u(u; u' = \dot{u}) \in X$ into our new space V.

```python
def evaluate_tlm_component(self, inputs, tlm_inputs, block_variable, idx,
                           prepared=None):
    assert len(inputs) == len(tlm_inputs)
    for i, input in enumerate(inputs):
        if tlm_inputs[i] is None:
            continue
        dJdm += self.backend.derivative(prepared, input, tlm_inputs[i])
    return self.backend.Interpolator(dJdm, self.V).interpolate()
```

Listing 8: Pyadjoint implementation of Eq. 5.5.13.

If we have multiple block inputs (as is implemented) we simply sum the interpolations of the directional derivatives of our expression with respect to each input. The corresponding code for this is shown in Listing 8. On the first line `inputs` and `tlm_inputs` are single entry lists containing u and $\dot{u}$ respectively whilst `prepared` is expr. The `idx` and `block_variable` arguments, which relate to there being multiple outputs, are surplus to requirements here and are not used.

Gateaux derivatives are calculated in UFL with the `ufl.derivative` operator. $\left.\frac{\text{dexpr}}{\text{d}u}\right|_u \cdot \dot{u}$ is calculated on the penultimate line: the first argument to `ufl.derivative` is the expression to differentiate (`prepared`, i.e. expr) the second argument is what the derivative is calculated with respect to (`input`, i.e. u) and the third argument is the direction (`tlm_inputs[0]` i.e. $\dot{u}$). The last line `self.backend.Interpolator(dJdm, self.V)` creates a linear interpolation operator from the expression for this derivative, `dJdm`, into the function space V; the `.interpolate()` method performs the interpolation which is then returned.

The adjoint `evaluate_adj_component` method for a single adjoint input covector $\overline{v}^* \in V^*$

is similarly

$$
\begin{aligned}
u'^{*}(;\bullet) &= \partial[I\circ\mathrm{expr}]^{*}_{u}(u;\bar{v}^{*}(;\bullet)) \quad \in U^{*} \text{ for given } u\in U && (5.5.14)\\
&= \bar{v}^{*}\cdot d[I\circ\mathrm{expr}]_{u}(u;\bullet) && (5.5.15)\\
&= \bar{v}^{*}\cdot\mathcal{I}\big(;d\mathrm{expr}_{u}(u;\bullet)\big) && (5.5.16)\\
&:= \bar{v}^{*}\cdot\mathcal{I}\cdot\frac{\mathrm{dexpr}}{\mathrm{d}u}\bigg|_{u}\cdot\bullet && (5.5.17)
\end{aligned}
$$

i.e. we (a) take the directional/Gateaux derivative of our expression, without supplying a direction, giving us $d\mathrm{expr}_u(u;\bullet) : U \to X$, (b) interpolate it into X giving us $\mathcal{I}\big(;d\mathrm{expr}_u(u;\bullet)\big) : U \to V$, then (c) operate on this with the covector $\bar{v}^*$. At 5.5.15 we have used the fact that our covectors are finite dimensional which allows us to represent their action as multiplication on the left (see equation 5.4.7).

```python
def evaluate_adj_component(self, inputs, adj_inputs, block_variable, idx,
                           prepared=None):
    if len(adj_inputs) > 1:
        raise(NotImplementedError("Interpolate block must have a single output"))

    dJdm = self.backend.derivative(prepared, inputs[idx])

    # make sure we have a cofunction output
    arg, = extract_arguments(dJdm)
    output = self.backend.Cofunction(arg.function_space().dual())

    return self.backend.Interpolator(dJdm, self.V).interpolate(
        adj_inputs[0], output=output, transpose=True)
```

Listing 9: Pyadjoint implementation of Eq. 5.5.17.

The corresponding code, for annotated interpolation with a single output vector space V and therefore single adjoint input covector $\bar{v}^*$, is shown in Listing 9. On the first line `inputs` and `adj_inputs` are single entry lists containing u and $\bar{v}^*$ respectively. `idx`, which tells us the index into the `inputs` list we are taking partial derivative for is 0. `inputs[idx].saved_output` is the same as `block_variable`, so `block_variable` is surplus to requirements here. `prepared` is expr. Unlike `evaluate_tlm_component` here `self.backend.derivative(prepared, inputs[idx])` has no direction specified so `dJdm` is $d\mathrm{expr}_u(u;\bullet)$. On the second-to-last line `self.backend.Interpolator(dJdm, self.V)` again creates a linear interpolation operator from our expression into the function space V then `.interpolate(adj_inputs[0], output=output, transpose=True)` left multiplies our interpolation operator (thanks to setting `transpose=True`) by the covector `adj_inputs[0]`, i.e. has the covector operate on the interpolation operator.

To ensure our interpolation operation outputs a `ufl.Cofunction`, we create one in the dual space U^*[12] and provide it as the output of the adjoint interpolation. Before performing this work this last step was not done: the analysis performed allowed this bug to be spotted and corrected.[13]

Clearly any nonlinearity in expr ought to become manifest in both the TLM and Adjoint actions due to them having $\frac{\text{dexpr}}{du}$ terms in them. If it is a linear operator it is a scaling factor and if it is a nonlinear operator it is a function of u. In most cases we do not annotate nonlinear operations and we merely deal with the scaling factor. We will go on to discuss this in the next section.

5.5.2 Hessian Action

For Hessian actions we limit the interpolate block to having one output. The interpolate block therefore has a single downstream adjoint mode covector and a single downstream Hessian vector product. In these cases no summation is required and the output of `evaluate_hessian_component` *is* our Hessian action.

For the purposes of this exposition, we limit ourselves to the non-mixed second derivative of equation 5.4.15 with downstream Hessian action input given by 5.4.23. We deal with our first derivative with the chain rule and our second derivative with the multivariate chain rule

$$\begin{aligned}
\partial\big[\partial[J\circ f]_u\big]_u(u,g;u',\bullet) &= \partial\big[\partial[J\circ f]_u(u,g;u')\big]_u(u,g;\bullet) && (5.5.18)\\
&= \partial\big[dJ_v[v=f|_{u,g};v'=\partial f_u(u,g;u')]_u\big]_u(u,g;\bullet) && (5.5.19)\\
&= \partial[dJ_v]_v\big(v=f|_{u,g};v'=\partial f_u(u,g;u'),v''=\partial f_u(u,g;\bullet)\big)\\
&\quad+\partial[dJ_v]'_v\big(v=f|_{u,g};v'=\partial f_u(u,g;u'),v''=\partial[\partial f_u(u,g;u')]_u(u,g;\bullet)\big). && (5.5.20)
\end{aligned}$$

In our second term we take the derivative of dJ_v with respect to the linear argument v' which, as in the case of the derivative of the interpolation operator (see equation 5.5.9), gives us dJ_v applied to the second derivative direction vector v'':

$$\begin{aligned}
\partial\big[\partial[J\circ f]_u\big]_u(u,g;u',\bullet) &= \partial[dJ_v]_v\big(v=f|_{u,g};v'=\partial f_u(u,g;u'),v''=\partial f_u(u,g;\bullet)\big)\\
&\quad+dJ_v\big(v=f|_{u,g};v''=\partial[\partial f_u]_u(u,g;u',\bullet)\big). && (5.5.21)
\end{aligned}$$

[12]the expression `dJdm` must contain a UFL `Argument` in U

[13]This was originally done by noticing that the interpolation result was still a `Function` so `.vector()` changed it to the necessary type before returning. This has been superseded by the addition of dual spaces to UFL and Firedrake.

In our alternative notation this is

$$\left.\frac{\partial^2[J\circ f]}{\partial u^2}\right|_{u,g}\cdot u'\cdot\bullet = \frac{\partial}{\partial u}\left(\left.\frac{\partial[J\circ f]}{\partial u}\right|_{u,g}\cdot u'\right)\Bigg|_{u,g}\cdot\bullet \tag{5.5.22}$$

$$= \frac{\partial}{\partial u}\left(\left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v=f|_{u,g}}\cdot\underbrace{\left.\frac{\partial f}{\partial u}\right|_{u,g}\cdot u'}_{=v'}\right)\Bigg|_{u,g}\cdot\bullet \tag{5.5.23}$$

$$= \left.\frac{\partial^2 J}{\partial v^2}\right|_{v=f|_{u,g}}\cdot\underbrace{\left.\frac{\partial f}{\partial u}\right|_{u,g}\cdot u'}_{=v'}\cdot\underbrace{\left.\frac{\partial f}{\partial u}\right|_{u,g}\cdot\bullet}_{=v''}$$

$$+\frac{\partial}{\partial v'}\left(\left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v=f|_{u,g}}\cdot v'\right)\Bigg|_{v'=\frac{\partial f}{\partial u}\big|_{u,g}\cdot u'}\cdot\frac{\partial}{\partial u}\left(\left.\frac{\partial f}{\partial u}\right|_{u,g}\cdot u'\right)\Bigg|_{u,g}\cdot\bullet \tag{5.5.24}$$

$$= \left.\frac{\partial^2 J}{\partial v^2}\right|_{v=f|_{u,g}}\cdot\left.\frac{\partial f}{\partial u}\right|_{u,g}\cdot u'\cdot\left.\frac{\partial f}{\partial u}\right|_{u,g}\cdot\bullet$$

$$+\left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v=f|_{u,g}}\cdot\left.\frac{\partial^2 f}{\partial u^2}\right|_{u,g}\cdot u'\cdot\bullet. \tag{5.5.25}$$

Sticking to our alternative notation for now, we investigate these two terms. We first note that $\frac{\partial f}{\partial u}\big|_{u,g}\cdot u'$ is, for some TLM variable $u' = \dot{u}$, the TLM action for the f that gives us $\dot{v}$. Therefore

$$\left.\frac{\partial^2 J}{\partial v^2}\right|_{v=f|_{u,g}}\cdot\left.\frac{\partial f}{\partial u}\right|_{u,g}\cdot\dot{u} = \left.\frac{\partial^2 J}{\partial v^2}\right|_{v=f|_{u,g}}\cdot\dot{v} \tag{5.5.26}$$

i.e. it's the downstream Hessian action input that we are given. To get our complete first term we therefore just need to left multiply it (i.e. have it operate on) $\frac{\partial f}{\partial u}\big|_{u,g}\cdot\bullet$ which is achieved by calling `evaluate_adj_component` with `adj_inputs` replaced by the downstream Hessian action input `hessian_inputs` (a single entry list). Listing 10 shows this, where `inputs` are the calculated values u and g.

The full Hessian action calculation is now equivalent to equation 3.8 in Naumann [51] which describes how forward-over-reverse mode accumulates Hessian-vector products (i.e. `hessian_inputs` values which propagate upstream). There $\nu^{(2)}_{(1)i}$ are the calculated entries in the Hessian-vector product, $\nu_{(1)i}$ is the downstream adjoint variable, $\nu^{(2)}_{k}$ in the TLM variable, and $\nu^{(2)}_{(1)j}$ are the downstream Hessian input values.

Since we operate a forward-over-reverse scheme we have also calculated and saved the downstream `adj_inputs` which, in this case, is a single entry list containing the covector/operator

$$\bar{v}^*(;\bullet) = \bar{j}^*\big(\partial J_v(v=f|_{u,g};\bullet)\big) = \partial J_v(v=f|_{u,g};\bullet) = \hat{v}(;\bullet) \tag{5.5.27}$$

$$:= \bar{j}^*\cdot\left.\frac{\partial J}{\partial v}\right|_{v=f|_{u,g}}\cdot\bullet \quad = \left.\frac{\partial J}{\partial v}\right|_{v=f|_{u,g}}\cdot\bullet \tag{5.5.28}$$

since the forward-over-reverse scheme has us calculate the complete adjoint-mode Jacobian (i.e.

```
def evaluate_hessian_component(self, inputs, hessian_inputs, adj_inputs,
                               block_variable, idx,
                               relevant_dependencies, prepared=None):

    if len(hessian_inputs) > 1 or len(adj_inputs) > 1:
            raise(NotImplementedError("Interpolate block must have a single output"))

    component = self.evaluate_adj_component(inputs, hessian_inputs,
                                            block_variable, idx, prepared)
```

Listing 10: Pyadjoint implementation of the first term in the sum of Eq. 5.5.21 (in alternative notation, Eq. 5.5.25). The implementation of the second term is of the code is shown in Listing 11.

seeded with $\bar{\jmath}^* = 1$). Note that were $J = a \circ b$ this would still work since the $\frac{\partial J}{\partial v}\big|_{v=f|_{u,g}}$ term would then be $\frac{\partial [a \circ b]}{\partial v}\big|_{v=f|_{u,g}}$.

This is needed in order to calculate the second term in equation 5.5.21 which has this operate on

$$\partial[\partial f_u]_u(u, g; u', \bullet) = \partial[\partial f_u(u, g; u')]_u(u, g; \bullet) \tag{5.5.29}$$

$$= \partial[\partial[\mathcal{I} \circ \text{expr}]_u(u, g; u')]_u(u, g; \bullet) \tag{5.5.30}$$

$$= \partial[\mathcal{I}\big(; \partial\text{expr}_u(u, g; u')\big)]_u(u, g; \bullet) \tag{5.5.31}$$

$$= \partial\mathcal{I}_x\big(; x = \partial\text{expr}(u, g; u'), x' = \partial[\partial\text{expr}_u(u, g; u')]_u(u, g; \bullet)\big) \tag{5.5.32}$$

$$= \mathcal{I}(; \partial[\partial\text{expr}_u(u, g; u')]_u(u, g; \bullet)) \tag{5.5.33}$$

where 5.5.31 makes use of the simplifications employed for finding the TLM action, 5.5.32 uses the chain rule and 5.5.33 uses the simplifications for the TLM action again. I.e. we have our covector/operator operate on the second derivative of our expression: the first derivative taken in the u' direction and the second left without a direction. We can calculate that explicitly, substituting u' with the TLM input variable $\dot{u}$ for the block.

We see here that our second term is zero whenever we interpolate from linear expressions since the second derivative of a linear expression is zero. The fact that we have nonlinear blocks indicates the need to perform an adjoint sweep of our DAG before we do our Hessian sweep.

Returning to code then, Listing 11 shows the rest of the implementation. The `relevant_dependencies` list here contain the block variable inputs u and g: when calculating the example here of $\partial\big[\partial[J \circ f]_u\big]_u(u, g; u' = \dot{u}, \bullet)$ only the block variable (`bv`) for u has a `bv.tlm_value` which isn't `None` which corresponds to $\dot{u}$. We then take the first derivative of our expression, where `bv.saved_output` is u, in the $\dot{u}$ direction. The `block_variable.saved_output` corresponds to the variable for which we want to take our second derivative, here u, which we do to get `d2exprdudu`. Next we interpolate `d2exprdudu` into our new space and, as with adjoint action, left multiply by our single adjoint input

```python
def evaluate_hessian_component(self, inputs, hessian_inputs, adj_inputs,
                               block_variable, idx,
                               relevant_dependencies, prepared=None):
    ...
    # Prepare expression
    expr = replace(self.expr, self._replace_map())

    # Calculate first derivative for each relevant block
    dexprdu = 0.
    for _, bv in relevant_dependencies:
        # Only take the derivative if there is a direction to take it in
        if bv.tlm_value is None:
            continue
        dexprdu += self.backend.derivative(expr, bv.saved_output, bv.tlm_value)

    # Calculate the second derivative leaving direction unspecified
    d2exprdudu = self.backend.derivative(dexprdu, block_variable.saved_output)

    # Make sure to have a cofunction output
    output = self.backend.Cofunction(component.function_space())

    # left multiply by dJ/dv (adj_inputs[0])
    component += self.backend.Interpolator(d2exprdudu, self.V).interpolate(
        adj_inputs[0], output=output, transpose=True
    return component
```

Listing 11: Pyadjoint implementation of the second term in the sum of Eq. 5.5.21 (in alternative notation, Eq. 5.5.25). The implementation of the first term, where `component` was given its initial value, is shown in Listing 11.

covector/operator $\bar{v}^*(;\bullet) = \hat{v}(;\bullet) = dJ_v(v = f|_{u,g};\bullet)$. All that's left to do is add this to our first term. Since we limit ourselves to having single Hessian and adjoint inputs our final operator is a covector rather than a matrix: we indicate this as before by returning a `Cofunction` by setting the `output` argument of `Interpolator.interpolate`.

5.6 Summary of Contributions

In this chapter I have produced a comprehensive description of AD using Gateaux derivatives. To my knowledge this is the first time this has been done. I have used this description to derive, for pyadjoint, the three key annotation operators of the interpolate operation: the TLM action, adjoint action and Hessian action. Whilst the TLM and Adjoint actions had already been implemented, I fixed a bug in the adjoint action. The Hessian action was implemented from scratch and required a different approach to the Hessian action found in other blocks in Firedrake's compatibility layer for pyadjoint `firedrake.adjoint`. The Hessian action is tested for correctness with a Taylor remainder convergence test: these are described in Sect. 8.4.

Having added the Hessian action I am able to use techniques such as Newton's method (see section 5.3.3) when solving PDE constrained optimisation problems that involve interpolation. This significantly increases the speed with which these problems can be solved. An example of this in action for assimilating point data is found in chapter 6.

Chapter 6

Consistent Point Data Assimilation

This chapter adapts and reuses, without further attribution, text, results, and diagrams from Nixon-Hill et al. [3] which were created by the author.

6.1 Introduction

Many disciplines face a common problem in the lack of observability of important fields and quantities. Take the geosciences as an example. In groundwater hydrology, the conductivity of an aquifer is not directly measurable at large scales; in seismology, the density of the earth; and in glaciology, the fluidity of ice. Nevertheless, these are necessary input variables to the mathematical models that are used to make predictions. Solving inverse problems or performing data assimilation, described in Sect. 5.1 is therefore extremely useful.

For example, the large scale density (an immeasurable) and displacement (a measurable) of the earth are related through the seismic wave equation (the F from Sect. 5.1). Combining measurements of displacement or wave travel time from active or passive seismic sources can then give clues to the density structure of the earth (an input to F).

Here we consider assimilating point data, as defined in Sect. 4.3. These encompass any measurements which have been performed at a location in space, at a particular time, giving some value. Returning to the geosciences, this encompasses most of the in-situ and remote sensing measurements one can imagine: satellite laser altimetry measurements, drifting weather buoys relaying weather and tidal information, geophone measurements of ground velocity and more. Derived data sets, such as the BedMachine map of Antarctic ice thickness [61] and the MEaSUREs InSAR phase-based velocity map of Antarctica [62], may appear to be field measurements, but they are ultimately derived from individual measurements or contain data that are fitted to discrete points on a grid.

So, we have some model

$$F(u, m) = 0 \tag{6.1.1}$$

where m is a set of parameters; u is our solution; and F the equation, such as a PDE, that relates m and u. The point data to assimilate are

$$u_{\text{obs}}^{i} \text{ at } X_i \text{ for } i = 0, ..., N_d - 1 \tag{6.1.2}$$

where N_d is the number of measurements. We aim to find parameters m (the immeasurable) that give us a particular u which we have discrete measurements of.

The functional we wish to minimise is

$$J = J_{\text{model-data misfit}} + J_{\text{regularisation}}. \tag{6.1.3}$$

The term $J_{\text{regularisation}}$ did not feature in the description in Sect. 5.1. It can be thought of as a prior guess at the properties our solution should have. Often this uses known properties of the physics of the model (such as some smoothness requirement) and, in general, ensures that the problem is well posed given limited, typically noisy, measurements of the true field u.

A key question to ask here is “what metric should we use for the model-data misfit?” A common approach, taken for example in glaciology by MacAyeal [63], Joughin, MacAyeal, and Tulaczyk [64], Vieli et al. [65], and Shapero et al. [66], is to perform a field reconstruction: we extrapolate from our set of observations to get an approximation of the continuous field we aimed to measure. This reconstructed field $u_{\text{interpolated}}$ is then compared with the solution field u

$$J^{\text{field}}_{\text{model-data misfit}} = \|u_{\text{interpolated}} - u\|^2_N \tag{6.1.4}$$

where $\|\cdot\|_N$ is some norm. The extrapolated reconstruction is referred to as $u_{\text{interpolated}}$ since, typically, this relies on some ‘interpolation’ regime found in a library such as SciPy [67] to find the values between measurements. As will be shown when these methods are returned to in Sect. 6.2, $J^{\text{field}}_{\text{model-data misfit}}$ is not unique since there is no unique $u_{\text{interpolated}}$ field. The method used to create $u_{\text{interpolated}}$ is up to the modeller and is not always reported: there are always only a finite number of data points, which could be concentrated in one region of the domain.

An alternative metric is to compare the point evaluations of the solution field $u(X_i)$ with the data u^i_{obs}

$$J^{\text{point}}_{\text{model-data misfit}} = \sum_{i=0}^{N_d-1} \|u^i_{\text{obs}} - u(X_i)\|^2_N. \tag{6.1.5}$$

Importantly, $J^{\text{point}}_{\text{model-data misfit}}$ is unique and independent of any assumptions made by the modeller.

It is the difference between minimising $J^{\text{field}}_{\text{model-data misfit}}$ and $J^{\text{point}}_{\text{model-data misfit}}$ which is investigated here. Previously code could be generated to minimise a functional containing $J^{\text{field}}_{\text{model-data misfit}}$ using Firedrake and dolfin-adjoint/pyadjoint. Indeed, due to their previously discussed limitations, no automated code generating finite element system can be used for this without taking a more lengthy approach such as deriving the continuous adjoint equations and discretising them manually.

The new abstractions for point data and point evaluation introduced in this work have changed this. As was described in Sect. 5.5, interpolation can be differentiated and, in Firedrake, this differentiation is integrated with dolfin-adjoint/pyadjoint through `firedrake.adjoint`. Since point evaluation is an interpolation operation this too is differentiable and integrated with `firedrake.adjoint`.

6.1.1 Review of Data Assimilation Approaches

Only the assimilation of what one might call ‘state’ into a model has been considered here: a snapshot of some measurement from an underlying probability distribution. Any reasonable measurement will come with some estimate of its uncertainty, which should be included in the assimilation process to give uncertainties in the derived quantities.

This requires that our model itself F in some way propagates probability distributions over time, usually represented by Probability Density Functions (PDFs). We assimilate into this the probability distributions of our measurements. This is a Bayesian inference problem of the form 'given prior PDFs of variables which describe the state of our model (so-called state variables, some of which may be hidden), and PDFs of some measurements, what are the resultant PDFs of the state variables?'

In this context, there are various approaches to data assimilation. The Kalman filter [68] produces an estimate of resultant state variable PDFs given Gaussian PDFs for the prior state variables and of the data. It was formulated for use on time-series data, hence it being called a 'filter'. It is a so-called 'optimal filter': assuming it operates on a linear time-invariant model[1] , with Gaussian PDFs, the resultant Gaussian PDF is exact.

Extensions to the Kalman filter for nonlinear systems exist [69, 70], but for systems of multiple variables, these, and the Kalman filter, require the calculation, storage and inversion of large covariance matrices [71]. The ensemble Kalman filter [72] is a popular approximation of the Kalman filter, where state variable PDFs are sampled from the larger, true set of state variable PDFs. This can be thought of as a dimension reduction to allow computational feasibility, or a Monte Carlo approach to solving the Bayesian inference problem. Realistic models are unlikely to be linear, time-invariant and involve only Gaussian PDFs: the ensemble Kalman filter assumes none of these to be true. There are also particle filters [73, 74], which implement sampling directly from the relevant PDFs to more directly approximate the solution to the Bayesian inference problem.

Here, a gradient based method for data assimilation is used which is commonly referred to as variational data assimilation. Alongside ensemble Kalman filters, variational approaches are widely used in numerical weather prediction [75], where huge quantities of data need to be assimilated into very large models of the earth system (see, for example, the UK Met Office [76, 77]). The method is broadly identical to that shown here with a misfit specified which is minimised. Since the model F involves PDFs, the functional for minimisation is then also stated in terms of PDFs. As an example, see Sect. 2.1 of Rihan, Collier, and Roulstone [78].

6.2 Unknown conductivity Experiment

We start with the squared L^2 norm for our 'field' model-data misfit functional

$$J^{\text{field}}_{\text{model-data misfit}} = \int_\Omega (u_{\text{interpolated}} - u)^2 dx \tag{6.2.1}$$

[1]A time-invariant model is one which does not itself change with time: a change in a model parameter at some time t will produce the same model output as if, all other things being equal, it were done at some alternative time $t + T$.

and the squared Euclidean (l_2) norm for the 'point' one

$$J^{\text{point}}_{\text{model-data misfit}} = \sum_{i=0}^{N_d-1} (u^i_{\text{obs}} - u(X_i))^2. \tag{6.2.2}$$

We write the latter as

$$J^{\text{point}}_{\text{model-data misfit}} = \int_{\Omega_v} (u_{\text{obs}} - \mathcal{I}_{\text{P0DG}(\Omega_v)}(u))^2 dx \tag{6.2.3}$$

where $u_{\text{obs}} \in \text{P0DG}(\Omega_v)$. Note that this is an expression for the L^2 norm again: since integration is equal to sums of point evaluations in P0DG(Ω_v) (Eq. 4.3.4) l_2 and L^2 norm are equal.

This is applied this to the simple model

$$-\nabla \cdot k\nabla u = f \tag{6.2.4}$$

for some solution field u and known forcing term $f = 1$ with conductivity field k under strong (Dirichlet) boundary conditions

$$u = 0 \text{ on } \Gamma \tag{6.2.5}$$

where Γ is the domain boundary. Conductivity k is asserted to be positive by

$$k = k_0 e^q \tag{6.2.6}$$

with $k_0 = 0.5$.

The inverse problem is then to infer, for the known forcing field term f, the log-conductivity field q using noisy sparse point measurements of the solution field u. This example has the advantage of being relatively simple whilst having a control term q which is nonlinear in the model.

To avoid considering model discretisation error, the log-conductivity field is generated as a finite element function q_{true} in the space of order 2 continuous Lagrange polynomials (P2CG) in 2D on a 32×32 unit-square mesh Ω with 2048 triangular cells. The model is then solved on the same mesh to get the solution field as another finite element function $u_{\text{true}} \in \text{P2CG}(\Omega)$. N_d point measurements $\{u^i_{\text{obs}}\}_0^{N_d-1}$ at coordinates $\{X_i\}_0^{N_d-1}$ are sampled from u_{true} and Gaussian random noise (representing the introduction of measurement uncertainty) with standard deviation $\{\sigma_i\}_0^{N_d-1}$ is added to each measurement.

A smoothing regularisation on the q field is used, which is weighted with a parameter α. This helps to avoid over-fitting to the errors in u_{obs} which are introduced by the Gaussian random noise. There are now two functionals to minimise

$$J[u,q] = \underbrace{\int_{\Omega_v} (u_{\text{obs}} - \mathcal{I}_{\text{P0DG}(\Omega_v)}(u))^2 dx}_{J^{\text{point}}_{\text{model-data misfit}}} + \underbrace{\alpha^2 \int_{\Omega} |\nabla q|^2 dx}_{J_{\text{regularisation}}} \tag{6.2.7}$$

and

$$J'[u,q] = \underbrace{\int_{\Omega} (u_{\text{interpolated}} - u)^2 dx}_{J^{\text{field}}_{\text{model-data misfit}}} + \underbrace{\alpha^2 \int_{\Omega} |\nabla q|^2 dx}_{J_{\text{regularisation}}} . \tag{6.2.8}$$

Each available method in SciPy's interpolation library are tested to find $u_{\text{interpolated}}$:

- $u^{\text{near.}}_{\text{interpolated}}$ using `scipy.interpolate.NearestNDInterpolator`,
- $u^{\text{lin.}}_{\text{interpolated}}$ using `scipy.interpolate.LinearNDInterpolator`,
- $u^{\text{c.t.}}_{\text{interpolated}}$ using `scipy.interpolate.CloughTocher2DInterpolator` with `fill_value = 0.0` and
- $u^{\text{gau.}}_{\text{interpolated}}$ using `scipy.interpolate.Rbf` with Gaussian radial basis function.

Note that since $u_{\text{interpolated}} \in \text{P2CG}(\Omega)$ each of 6 degrees of freedom per mesh cell has to have a value estimated given the available u_{obs}.

The estimated log-conductivity q_{est} which minimise the functionals are found using gradient decent by generating code for the adjoint of our model using dolfin-adjoint/pyadjoint then using the Newton-CG minimiser from the `scipy.optimize` library. This makes use of the ability to calculate Hessian-vector products in `firedrake.adjoint`, described in Sect. 5.5.2.

It is standard practice (see for example Shapero et al. [66]) to perform an l-curve analysis [79] to identify a value of α that balances the relative weights of the model-data misfit and regularisation terms. For fairness, this is done for both J and J'. The l-curves were gathered for $N_d = 256$ randomly chosen point measurements with the resultant plots shown in Fig. 6.1 and Fig. 6.2. For low α, $J^{\text{field}}_{\text{misfit}}$ stopped being minimised and solver divergences were seen due to the problem becoming ill formed. $\alpha = 0.02$ was therefore chosen for each 'field' method. For consistency $\alpha = 0.02$ was also used for J.

An extract of the Firedrake and dolfin-adjoint/pyadjoint code needed to minimise J is shown in Listing 12. The Firedrake expression for J in the code is the same as the mathematics in Eq. 6.2.7 given that `assemble` performs integration over the necessary mesh. The last 3 lines are all that are required to minimise our functional with respect to q, with all necessary code being generated. Note that we require a reduced functional (`firedrake.adjoint.ReducedFunctional`) since the optimisation problem depends on both q and $u(q)$: for a thorough explanation see Sect. 1.4 of [29].

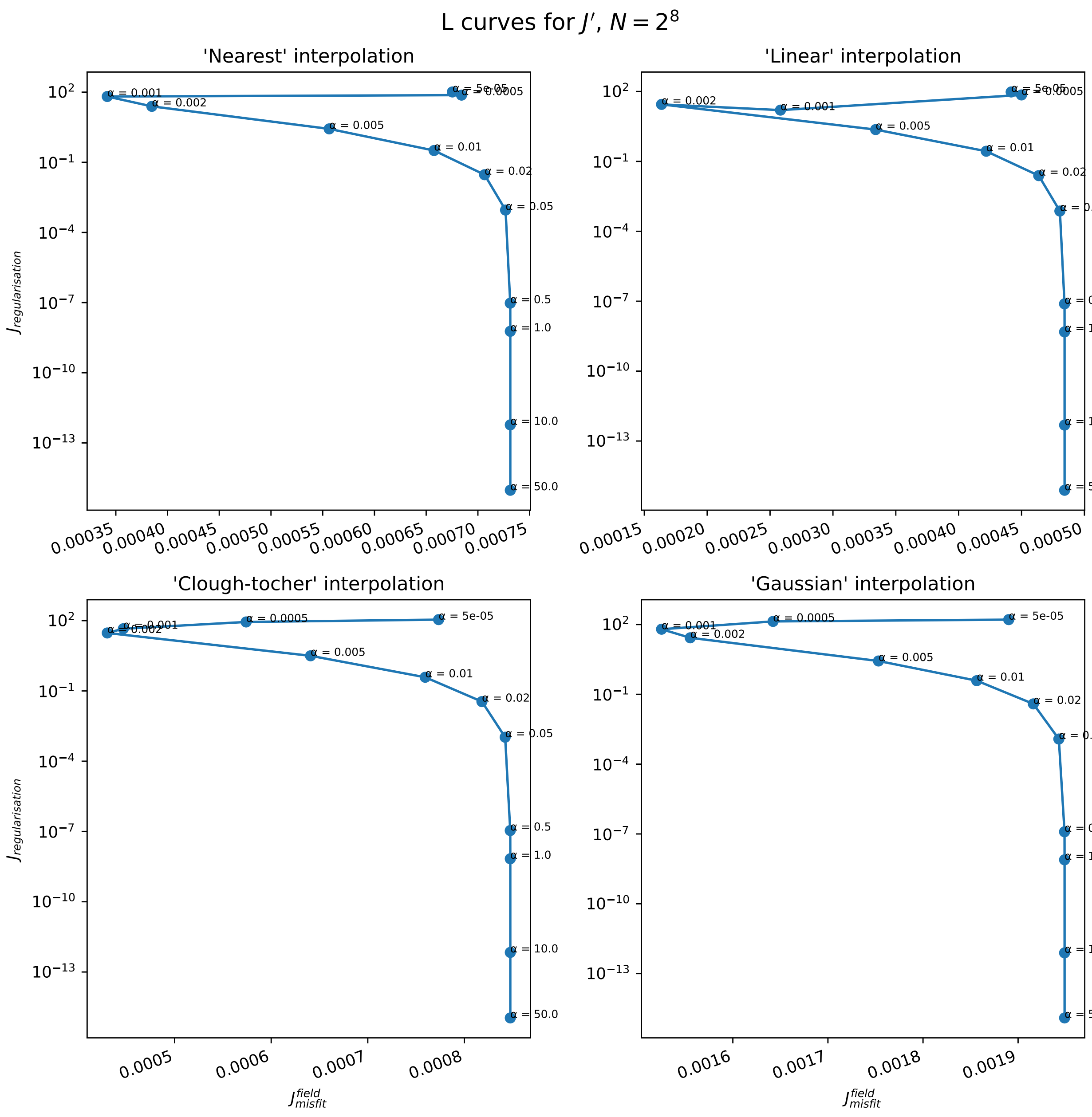


Figure 6.1: L-curves from minimising J' (Eq. 6.2.8) with different methods for estimating $u_{\text{interpolated}}$. For low α, $J^{\text{field}}_{\text{misfit}}$ stopped being minimised and solver divergences were seen for $u^{\text{gaussian}}_{\text{interpolated}}$. The problem was likely becoming overly ill formed and the characteristic 'L' shape (with sharply rising $J_{\text{regularisation}}$ for low α and a tail-off for large α) is therefore not seen. To keep the problem well formed without the regularisation parameter being too big $\alpha = 0.02$ is chosen for each method.

```python
1  from firedrake import *
2  from firedrake.adjoint import *
3  continue_annotation()
4
5  # Import our noisy samples of the true u
6  u_obs_coords = ...
7  u_obs_vals = ...
8
9  # Solve PDE with a guess for q giving an initial u
10 ...
11
12 omega_v = VertexOnlyMesh(omega, u_obs_coords)
13 P0DG = FunctionSpace(omega_v, "DG", 0)
14 u_obs = Function(P0DG)
15
16 # safely input data (covered in another chapter)
17 P0DG_input_ordering = FunctionSpace(omega_v.input_ordering, "DG", 0)
18 u_obs_input_ordering = Function(P0DG_input_ordering)
19 u_obs_input_ordering.dat.data_wo[:] = u_obs_vals
20 u_obs.interpolate(u_obs_input_ordering)
21
22 J_misfit = assemble((u_obs - interpolate(u, P0DG))**2 * dx)
23 alpha = Constant(0.02)
24 J_regularisation = assemble(alpha**2 * inner(grad(q), grad(q)) * dx)
25 J = J_misfit + J_regularisation
26
27 q_hat = firedrake.adjoint.Control(q)
28 J_hat = firedrake.adjoint.ReducedFunctional(J, q_hat)
29 q_min = firedrake.adjoint.minimize(J_hat, method="Newton-CG")
```

Listing 12: Firedrake code for expressing J (Eq. 6.2.7) and dolfin-adjoint/pyadjoint code (inside a Firedrake wrapper) for minimizing it with respect to q. The omitted PDE solve code is very similar to that in Listing 2; for more see the code itself, archived on Zenodo at [80]. The section where data is safely input (lines 6 and 7) is covered in chapter 7, Sect. 7.6.1.

6.2.1 Posterior consistency

It is expected that the error in our solution, when compared to the true solution, will always decrease as the number of points assimilated increases. This has the caveat that the underlying probability distribution that the measurements are drawn from should have unchanging variance and the measurements themselves should be independent[2]. From a Bayesian point of view, this is known as posterior consistency: under appropriate assumptions, in a well-posed Bayesian inverse problem the posterior distribution should concentrate around the true values of the estimated quantity [81]. The regularisation one chooses and the weighting it is given encodes

[2]there exists a generalisation of posterior consistency to certain kinds of correlated measurement errors which is not relevant here

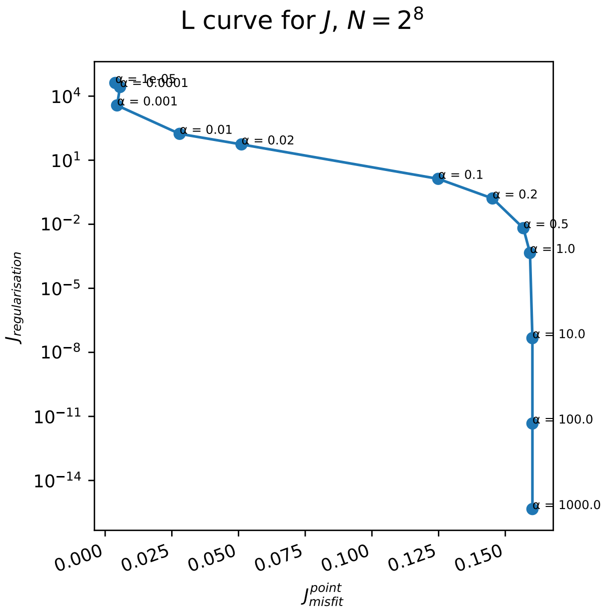


Figure 6.2: L-curve for minimising J (Eq. 6.2.7). The characteristic 'L' shape is seen and $\alpha = 0.02$ is seen to be close to the turning point and is therefore chosen for consistency with the other L-curves (Fig. 6.1).

information about the assumed prior probability distribution of q before one starts assimilating data (adding observations).

Take, for example, the regularisation used in this problem

$$\alpha^2 \int_{\Omega} |\nabla q|^2 dx. \tag{6.2.9}$$

This asserts a prior that the solution q which minimises J and J' should be smooth and gives a weighting α to the assertion. If we have posterior consistency, the contribution of increasing numbers of measurements u_{obs} should increase the weighting of our data relative to our prior and q_{est} should converge towards the true solution q_{true}. It is clear that this ought to happen for J, the point evaluation approach, since increasing N_d increases the number of terms in the model-data misfit sum (i.e. the integral over Ω_v gets bigger: see the equivalency of Eq. 6.2.3 and Eq. 6.2.2). There is no such mechanism for J', the field reconstruction approaches, since adding more data merely ought to cause $u_{\text{interpolated}}$ to approach u_{true} without increasing the relative magnitude of the misfit term.

Figure 6.3 demonstrates that the problem formulated with J, the point evaluation approach, both demonstrates posterior consistency and produces a q_{est} which is closer to q_{true} for all but the lowest N_d when compared to our formulations with J'.[3] The point evaluation approach therefore gives us *consistent* point data assimilation when compared to the particular field reconstruction approach we test: it results in posterior consistency and is therefore consistent

[3]It should be noted that α has not been optimised for minimising J so it is not unreasonable to assume that the prior is dominating the solution for low N_d.

with Bayes' Theorem.

It is possible, were an l-curve analysis repeated for each N_d, that errors in our field reconstruction approach could be reduced. The lack of convergence would not change due to there being no mechanism for growing the misfit term with number of measurements.

Example calculated fields are shown in Fig. 6.4 and Fig. 6.5. These demonstrate that the choice of interpolation method changes the field reconstruction.

One could attempt to enforce posterior consistency on the field reconstruction approach (minimising J') by introducing a term in the model-data misfit which increases with the number of measurements. When attempting to enforce posterior consistency we would also need to ensure that our field reconstruction method approaches the true field as more measurement are performed. There is no obvious way to do this which is universally applicable, particularly since measurements are always subject to noise.

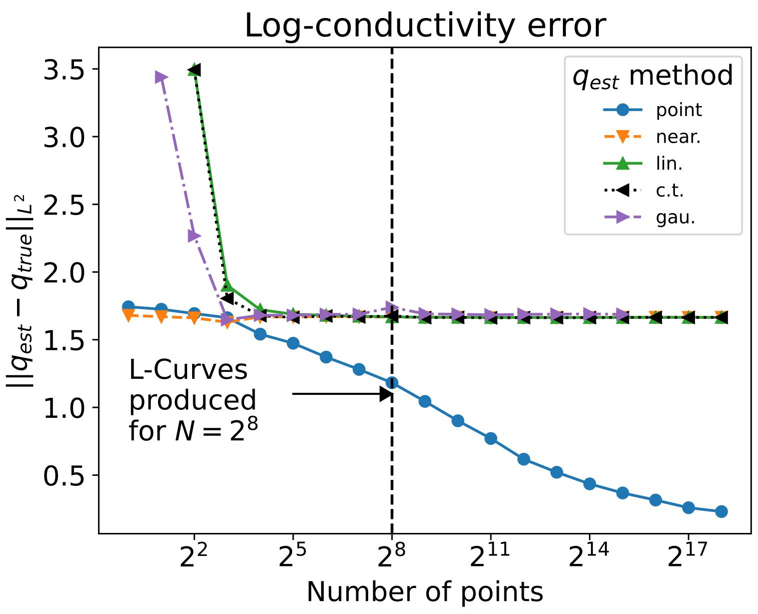


Figure 6.3: Error change as number of points N_d is increased for minimising J ($u_{\text{interpolated}}$ and q_{est} method 'point' - see Eq. 6.2.7) and J' (the other lines - see Eq. 6.2.8) with different methods for estimating $u_{\text{interpolated}}$ where $\alpha = 0.02$ throughout (see main text for justification). The l-curves for $\alpha = 0.02$ with $N_d = 256$ are shown in Fig. 6.1 and Fig. 6.2. Not all methods allowed $u_{\text{interpolated}}$ to be reconstructed either due to there being too few point measurements or the interpolator requiring more system memory than was available.

6.3 Data Assimilation in Ice Shelf and Ice Sheet Literature

Given the apparent superiority of point evaluation misfit functionals, one is left to wonder why they are not used more widely aside from limitations in software. In particular, field

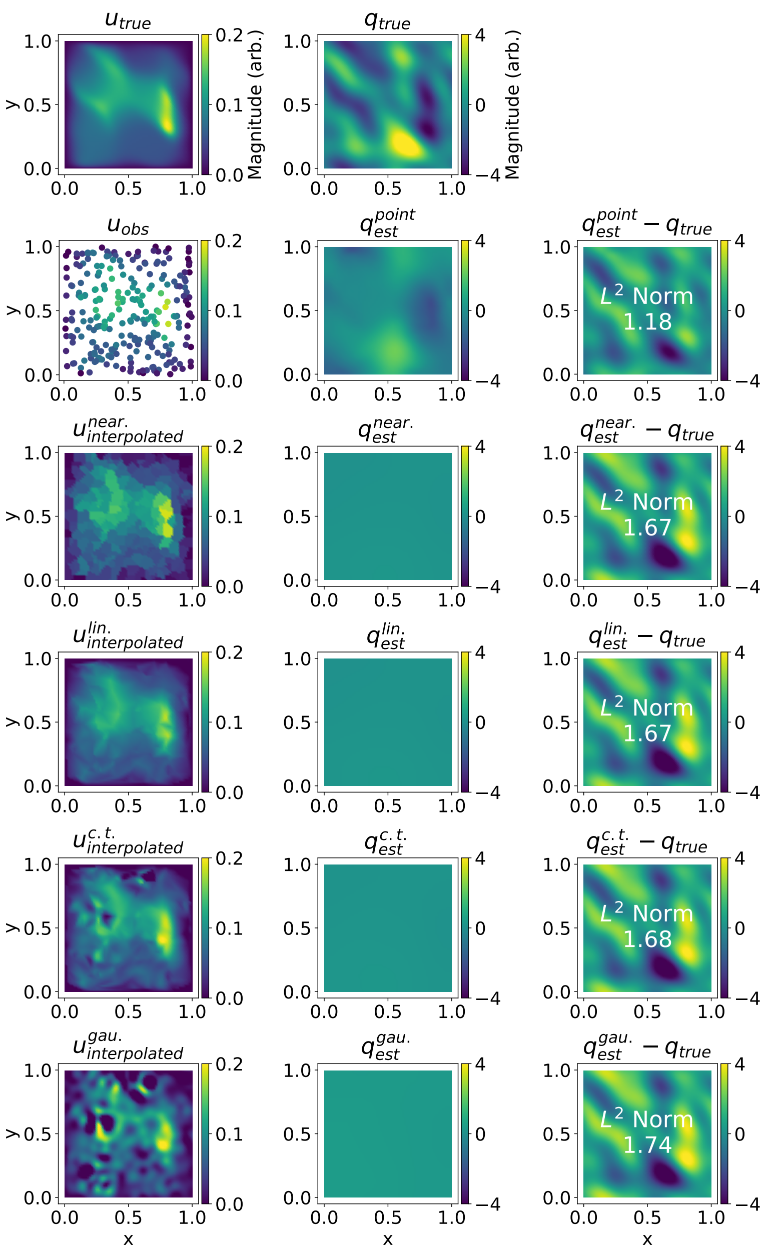


Figure 6.4: Summary plot of fields for $N_d = 256$. Rows correspond to method used where column 1 is the necessary u, column 2 is the corresponding q at the optimum solution, and column 3 is the error. Row 1 shows the true u and q. Row 2 has us minimising J (Eq. 6.2.7) whilst rows 3-6 have us minimising J' (Eq. 6.2.8). The regularisation parameter $\alpha = 0.02$ throughout. The field we get after minimising J, $q_{\text{est}}^{\text{point}}$, manages to reproduce some features of q_{true}. For minimising J' the solutions fail to reproduce any features of q_{true} and the error is therefore higher. Each of the $u_{\text{interpolated}}$ fields are also visibly different from one another. For comparison see Fig. 6.5.

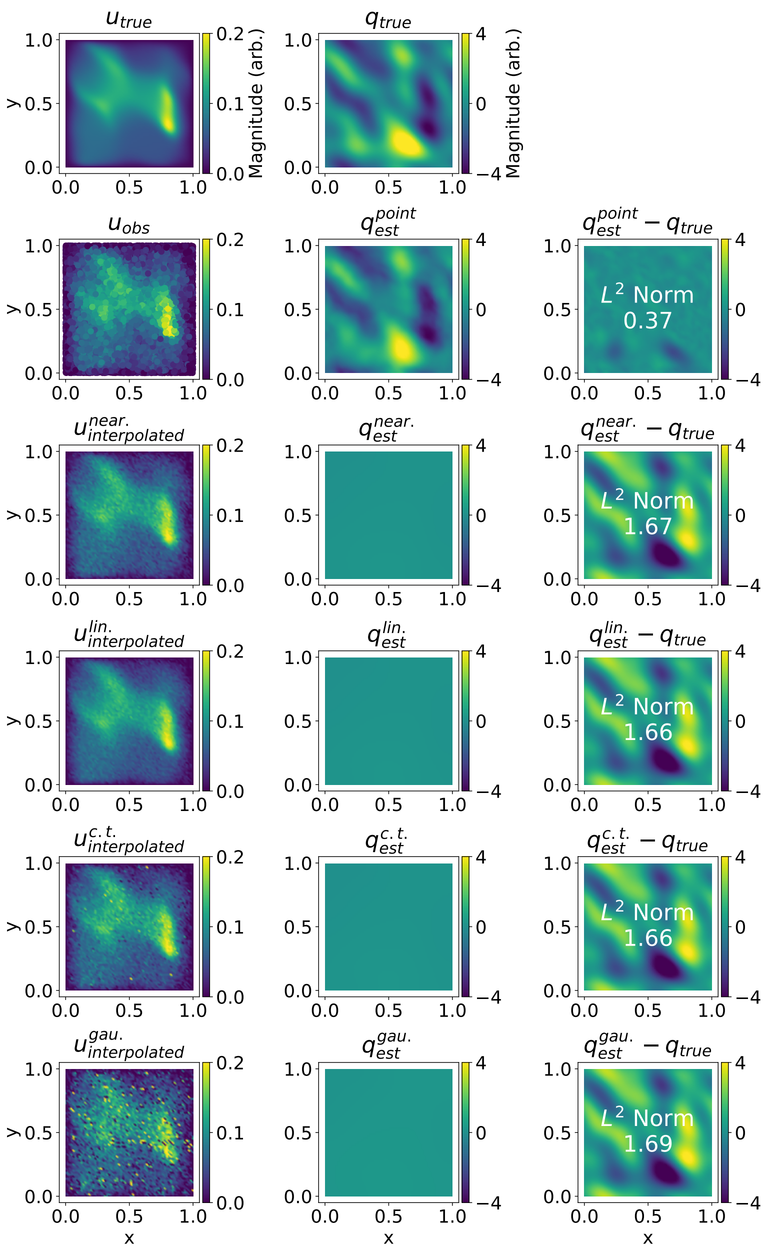


Figure 6.5: Summary plot of fields for $N_d = 32768$. Rows and columns correspond to those in Fig. 6.4. The regularisation parameter $\alpha = 0.02$ throughout. We would expect the larger number of measurements to correspondingly reduce the error: this only occurs to a significant degree when solving J. This cannot be entirely blamed on a lack of mechanism in J' for having the misfit term outgrow the regularisation term: the $u_{\text{interpolated}}$ fields do not approximate u_{true} with $u^{\text{gau.}}_{\text{interpolated}}$ being particularly poor.

reconstruction misfit functionals seem particularly prevalent in ice shelf and ice sheet data assimilation literature.

In the work where MacAyeal et al. introduce the ‘control method’ for ice sheets [63] they start by considering an analytical solution for the ice velocity field in the *Inverse Problem* section:

> As is well known, there are many ways to interpolate a field of irregularly spaced observations. Bindschadler and Scambos [1991] used a commercially available package based on kriging principles, and then adjusted the results by hand. It is thus possible for the interpolation method to artificially affect the β field derived from (4) and (5). To avoid these effects, I recommend using an interpolation scheme that is faithful to the physics which govern the ice stream velocity. This scheme is the control method which I discuss below.
>
> The direct algebraic inversion also faces several other mathematical hazards. It works only if the data, $U_d = (u_d, v_d)$, possess all of the mathematical properties required of a solution of (1) - (3). There may be velocity gradients contained in this data that cannot be explained by any distribution of β.

Without diving into the symbols and equations being referenced here, it is pointed out that (a) methods of interpolation vary and can change results and that (b), unless the observations are exact solutions to the equations (which is unlikely given the approximations being made) you won’t get a solution that makes sense anyway. Point (b) motivates setting the problem up as a constrained optimisation problem but it seems that (a) is forgotten about. The control method is introduced by forming a functional that directly compares computed and observed velocity fields and, in a section titled *Satellite Imagery Derived Velocity Data* they write

> Figure 2 shows a contour map of longitudinal and transverse components of the observed surface velocity interpolated to a 65x65 finite difference grid to be used in the modeling and inversion calculations.

The precise nature of this interpolation is not discussed despite the observed velocity data being noted for its irregularity.

This first paper was written with finite difference models which inherently give solutions on grids, nevertheless efforts could have been taken to interpolate the modelled grid data to the specified location or to simply not use all grid points. Fortunately finite element methods are not tied to grids and instead give solutions everywhere. Nevertheless MacAyeal’s approach seems to have become the defacto method for performing data assimilation with ice sheets with the subtly incorrect model-data misfit propagating through the literature, perhaps via the tutorial he published in 1993 [82].

In another paper MacAyeal, Bindschadler, and Scambos [83] adopt the finite element method but the model/observation misfit functional (Eq. 2) is once again given in terms of the comparison of two fields defined over the entire domain. Efforts are made here to try and deal with observational uncertainty in a section titled *Treatment of observational uncertainty* but these require introducing scalar terms which are defined for the entire continuous vector field of "observed" data (see Eq. 3 to Eq. 7): no attempt is made to deal with uncertainty of specific measurements.

Joughin, MacAyeal, and Tulaczyk [64] is similar: the model-data misfit (Eq. 7) uses complete field solutions despite data sparsity being mentioned when the data is introduced (Sect. 5). Efforts are made to characterise the model's sensitivity to different solutions (Sect. 3) but the manufactured data used are defined as continuous fields and are not passed through the complete input data pipeline, including any interpolation that is done. This could significantly undermine the quality of the results. Indeed the words 'interpolation' and 'extrapolation' are not found in the paper.

Things initially look better in Vieli et al. [65] where a sum is used as the model-data misfit functional (Eq. 2.4), but closer inspection reveals that this is done over all finite difference grid points with no attention given to specific measurement locations. The data used is preprocessed to give a complete velocity field and the grid is laid upon this to get the velocity components in each case (Fig. 2).

The conclusion drawn here is that a combination of a reliance on what has been done before, and no demonstration of the disadvantages of the approach are the reason, hence the value of this work.

6.4 Future Work

This work has already been extended to assimilate data in groundwater hydrology and in modelling the Larsen C ice shelf: see Nixon-Hill et al. [3]. It is not included here since it was primarily the work of my coauthor, Daniel Shapero.

It is important that our estimated u reflect the measurement uncertainty, and future work ought to integrate that with the Bayesian inference perspective used elsewhere in data assimilation (for Antarctic ice flow, see, for example Isaac et al. [84]). The point evaluation misfit approach allows raw location and value data to be directly used with measurement-specific errors being defined as necessary. So instead of our model-data misfit being

$$\int_\Omega \left(\frac{u_{interpolated} - u}{\sigma}\right)^2 dx \tag{6.4.1}$$

we can instead write

$$\sum_{i=0}^{N_d-1} \left(\frac{u_{obs}^i - u(X_i)}{\sigma_i}\right)^2 \tag{6.4.2}$$

where u_{obs}^i are specific remote-sensing data and σ_i a relative quantification of their measurement error (the width of the PDF) such as the standard deviation. Where σ_i are small, the u from minimising the functional will be closer to u^i_{obs} and the more confident we can be in whatever aspect of q is associated with that measurement. From a Bayesian inference point of view, our regularisation is the prior. The measurements u^i_{obs} are the likelihood, with PDF widths proportional to σ_i (the bigger the σ_i, the less confident we are in u^i_{obs}). The fields we estimate, q and u, are the posterior. The challenge comes in finding the associated PDF widths of them, since that would require our model F to propagate that information. Nevertheless, taking this approach would ensure that q and u appropriately reflect the confidence we have in the measurements and our prior.

We can estimate the PDFs of q and u by taking an ensemble of samples from the PDF of the measurements (with appropriate weightings), propagating them through the minimisation, and seeing how they affect the estimation of q and u. A more in-depth review of variational data assimilation methods which use PDFs could also provide some options.

An interesting question arises when we consider uncertainty in the location of our measurements. One approach is to simply solve this problem multiple times with the points in different locations, as dictated by the PDF of the measurement location, to get a sense of the sensitivity of the problem to the locations. Another approach would be to use the same misfit functional but to also solve for the point locations via a penalty term which weights the point locations as necessary given their location error

$$\sum_{i=0}^{N_d-1} A_i \left(\frac{u^i_{obs} - u(X_i)}{\sigma_i} \right)^2 \tag{6.4.3}$$

where A_i is a weighting and X_i is what we solve for. Solving for point locations via gradient methods requires the derivative of our misfit functional with respect to the position. This could be done by taking the derivative of an operator that moves points, which would be implemented alongside movable points, discussed in Sect. 10.2.2.

6.5 Concluding Remarks

This is both a demonstration of new functionality and a general call for all scientific communities who face these kinds of inverse problems to carefully consider if point evaluation misfit functionals would be appropriate for their use case. Using these instead of field reconstruction approaches in data assimilation has several benefits: it (i) ensures our inverse problem displays posterior consistency, (ii) reduces errors, (iii) avoids the need for ambiguously defined inter-measurement interpolation regimes, and (iv) allows assimilation of very sparse measurements. For data assimilation problems where this is not possible, ensuring that the model-data misfit grows with the number of measurements could also be considered.

Of course, to use model-data misfits which are evaluated at discrete points we need to be able to perform point evaluation on the fields which we compare to our discrete measurements.

These point evaluations need to integrate with whatever method we are using to solve our optimisation problem: in many cases this requires finding a first or second derivative of the point evaluation operation alongside operations such as solving a PDE. Such libraries are few and far between. Vertex-only mesh and interpolation as a point evaluation operation make this straightforward, and serve as a demonstration of the power of this abstraction. Firedrake and `firedrake.adjoint` are excellent tools to use for this purpose, and makes solving the necessary minimisation problem very straightforward.

6.6 Summary of Contributions

I have used my new abstractions to solve a PDE constrained optimisation problem which directly uses point data. I have demonstrated the advantages of this approach over field reconstruction methods which were (a) previously the only option with Firedrake or FEniCS without hand-coding derivatives and (b) widely used. This work has been published as a preprint and is currently under review for journal publication (see [3]).

The Firedrake code which generated the figures here, of which I wrote the `unknown-conductivity` directory, is archived on Zenodo at [80] and used a Firedrake version similarly archived at [85].

Chapter 7

Parallel Safe, Unlosable Point Data

This chapter adapts and reuses, without further attribution, text from Nixon-Hill et al. [3] which was created by the author.

7.1 Introduction

This chapter further documents the implementation of the vertex-only mesh whose symbolic properties were introduced in Chapter 4. When implementing a vertex-only mesh in a parallel computing system, there are considerable challenges that must be overcome: these are outlined in Sect. 7.2. The discussion then moves on to algorithms which uniquely associate vertex-only mesh vertices with specific parent mesh cells (Sect. 7.4) and the movement of point data between different parallel decompositions (Sections 7.5 and 7.6).

7.2 Motivation

Everything added to Firedrake is expected to work when run in parallel using MPI parallelism and, ideally, to use that parallelism to speed up calculations. When meshes are created, they are parallel decomposed such that different MPI ranks contain different sections of the mesh. This so-called *parallel domain decomposition* is common to many systems which run in parallel and solve problems on some domain [86]. At the edges of the domain on each rank there is a *halo* which is a section of mesh which is also present on other MPI ranks. This is illustrated in Fig. 7.1. The halo regions allow for communication between MPI ranks about, for example, the values of a PDE solution on its local mesh section via a process known as *halo exchange*.

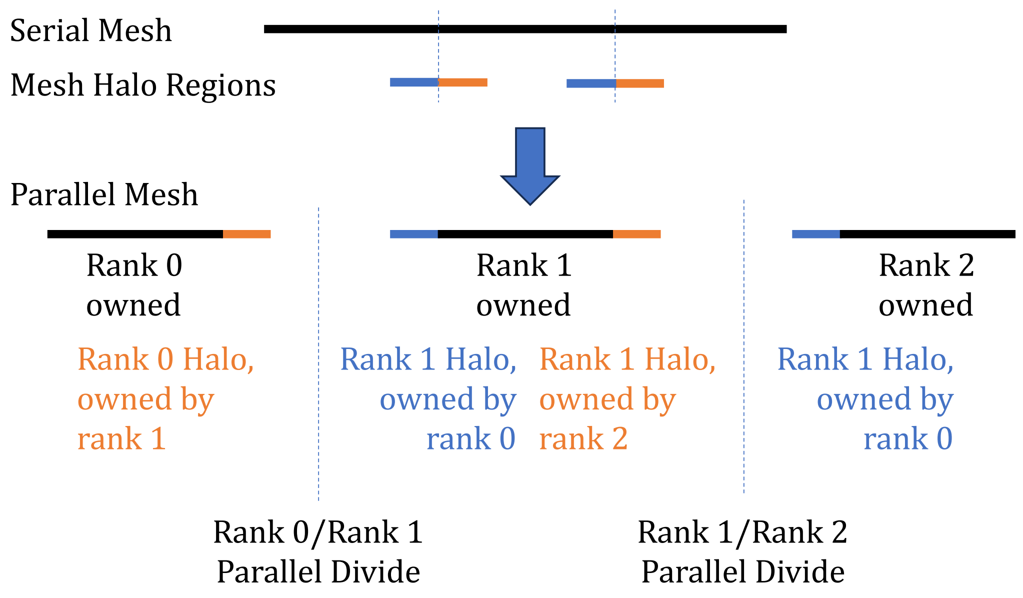

Serial Mesh
Mesh Halo Regions
Parallel Mesh
Rank 0 owned
Rank 1 owned
Rank 2 owned
Rank 0 Halo, owned by rank 1
Rank 1 Halo, owned by rank 0
Rank 1 Halo, owned by rank 2
Rank 1 Halo, owned by rank 0
Rank 0/Rank 1 Parallel Divide
Rank 1/Rank 2 Parallel Divide

Figure 7.1: Parallel domain decomposition of an interval mesh across 3 ranks. The coloured regions correspond to halos regions which are owned by other MPI ranks.

There is no guarantee that point data on one MPI rank will match the domain decomposition of the model we want to use them with. If we are coupling models, one model may produce point data with a different MPI domain decomposition than the model from which we consume the data. Similarly we may be simultaneously reading different data from various data sets on different MPI ranks.

An ideal implementation of vertex-only mesh should therefore allow each MPI rank to be supplied with a set of coordinates to embed and redistribute them to the ranks that contain the corresponding parent mesh domains. These points must be guaranteed to be embedded in the mesh if they are in its domain. Furthermore, it should be possible to send the values associated with those coordinates to the coefficients of the associated P0DG function.

Meshes are generally approximations to a domain: a curved surface may be approximated by a number of planar facets for example. It is therefore important that it be possible to embed point data which are outside the mesh domain by some specifiable tolerance, typically of the order of the size of a mesh cell. The closest cell should then be chosen. This should apply both at the domain edges and at the edges of parallel domains: where two ranks find a point outside their rank-local domain, the cell which it is nearest to should be chosen. In the unlikely case that the distances are the same, a single MPI rank should be chosen as the owner.

7.3 Existing Capability outside Firedrake

Particle methods, where the movement of some number of particles are tracked, are found throughout scientific computing. Particle-in-cell methods, which combine mesh or grid based methods for solving PDEs with particle methods, are often used for solving complex problems in areas such as plasma physics. Implementations of these methods are typically built to run in parallel on HPC platforms. Methods for linking point data with meshes and grids in parallel has therefore been tackled before.

Particle methods are found elsewhere: often these don't involve a mesh but work directly on the basis of moving particles around (these are known as 'mesh free' methods). Applications range from molecular dynamics, to computational fluid dynamics and methods which sample from probability distributions to estimate new probability distributions. These methods all involve moving particles, which is beyond the scope of this work, and since they don't necessarily involve a mesh, have more parallel distribution options available to them.[1] Nevertheless, one may wish to couple such methods to a simulation in Firedrake, which would require finding the locations of those particles on a domain decomposed mesh.

Much work has been put into tracking particles as they move through an irregular domain such as an unstructured mesh; the introduction to Kuang, Yu, and Zou [87] highlights and

[1]A quick summary of some options can be found in chapter 16 of Prof. Michael T. Heath's Parallel Numerical Algorithms course taught at the University of Illinois at Urbana-Champaign. These are available at https://courses.engr.illinois.edu/cs554/fa2011/notes/16_particle.pdf.

categorises some of the literature on this. There are plenty of highly sophisticated algorithms, many of which could prove useful in the future (see Sect. 7.8), but for now we are interested in the initial location of points. There is surprisingly little literature on this process and its parallelisation, as most articles devote their attention to computationally expensive process of relocating points.

A common approach, and that suggested by Strobl, Bannerman, and Pöschel [88], is to use specialised database structures to efficiently locate coordinates within an unstructured mesh. Such database structures and associated indexing tools are used extensively in graphical applications such as ray tracing and in simulation environments for collision detection. The specific approach used here (and indeed by other code-generating finite element libraries) is highlighted in 7.4.

Deal.II, like Firedrake, uses parallel domain decomposition. Its `Particles` class encapsulates the notion of particles in a mesh which can be thought of as vertices of a vertex-only mesh. They carry a location in real space, another within the reference cell of the cell they are in, and a unique ID. These can be given values to carry around, which is analogous to creating a P0DG function on a vertex-only mesh. Large numbers of particles are handled by a `ParticleHandler` class. Gassmöller et al. [89] used these tools to implement particle-in-cell methods in deal.II.

Legacy FEniCS and FEniCSx are also compatible with mesh parallel domain decomposition. Both contain functionality for locating points, but no overarching data structure. Like deal.II, the LEoPart add in for FEniCS [90] also has a `particles` class. This holds information about the positions of particles and, optionally, quantities such as momentum, density and concentration assigned to them. Rather than assign each particle information about the parent cell and position, LEoPart takes the approach of keeping track of the number of particles within each mesh cell. The intended use case is Lagrangian particle tracing, as Maljaars, Richardson, and Sime [90] detail. Notably, the 'point interpolation' operation for getting values from a function on a mesh in LEoPart is not differentiable.

7.4 Cell Location Algorithm: Avoiding Point Loss

A robust point location scheme ensures that each vertex or point in a vertex-only mesh is assigned a parent mesh cell. This involves (A) identifying a list of candidate cells within which the point may reside then (B) identifying the exact cell from the candidates.

We start with step (A): When meshes are created axis-aligned bounding boxes (boxes with edges aligned to the global mesh coordinate system) are placed around the extrema of each cell. These boxes necessarily overlap so a given point may be in more than one. Since they are axis-aligned, the boxes that contain a given coordinate (x, y, z) can be quickly identified by looking at the box limits. There is 1 bounding box per cell; the candidate cells are the bounding boxes containing the coordinate. A visualisation of this is shown in Fig. 7.2 (b). To include points which are close to the domain edge, each bounding box is expanded by a

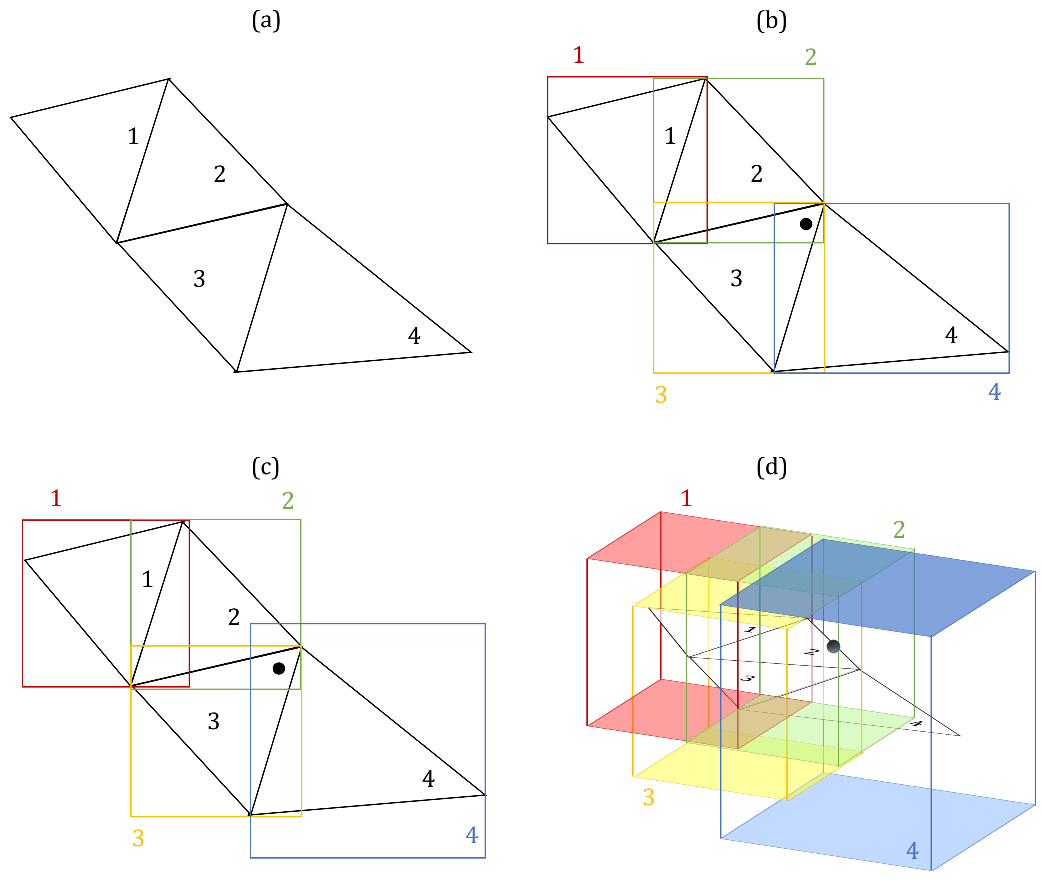


Figure 7.2: A mesh of four cells (a) with axis aligned bounding boxes. (b) shows the bounding boxes around the cell extrema: the point in cell 3 (the dot on the diagram) is inside three overlapping boxes corresponding to cells 2, 3, and 4 which are the cell candidates. (c) and (d) show the expansion of the bounding boxes to a hypercube centred on the middle of the cell to allow for point location on immersed manifolds, of which (d) is an example. In this example, the tolerance is zero: the boxes would be further expanded should a tolerance be included.

cell-size-relative tolerance.

This has obvious advantages over a brute force approach in which all cells are searched for a given point: bounding boxes can also be constructed in other code-generating finite element libraries such as deal.II[2] [33], legacy FEniCS[3] [17] and FEniCSx[4]. Maljaars, Richardson, and Sime [90] use the FEniCS bounding box search for LEoPart. Similarly, Gassmöller et al. [89] use the deal.II equivelent in their particle-in-cell implementation. More generally, axis-aligned bounding boxes are widely used for extracting spatial subdomains from unstructured spatial data [91] such as unstructured meshes.

To allow for cell location on immersed manifold meshes, each bounding box is expanded to be an axis-aligned hypercube with geometric dimension of the mesh, centred on the midpoint of the previous bounding box. The side length of the hypercube is the longest side of the previous bounding box (the L^1 diameter).[5] This appears to be novel to Firedrake. A visualisation is shown in Fig. 7.2 (c) and (d).

In practice, the bounding boxes are stored on the mesh in an R^*-tree [92] database structure (a 'spatial index') using the libspatialindex library [93]. Candidate cell identification uses lookup tools built into libspatialindex. For interval meshes all the cells are treated as candidates since libspatialindex does not support 1D, though spatial indexing methods for 1D are relatively straightforward.[6]

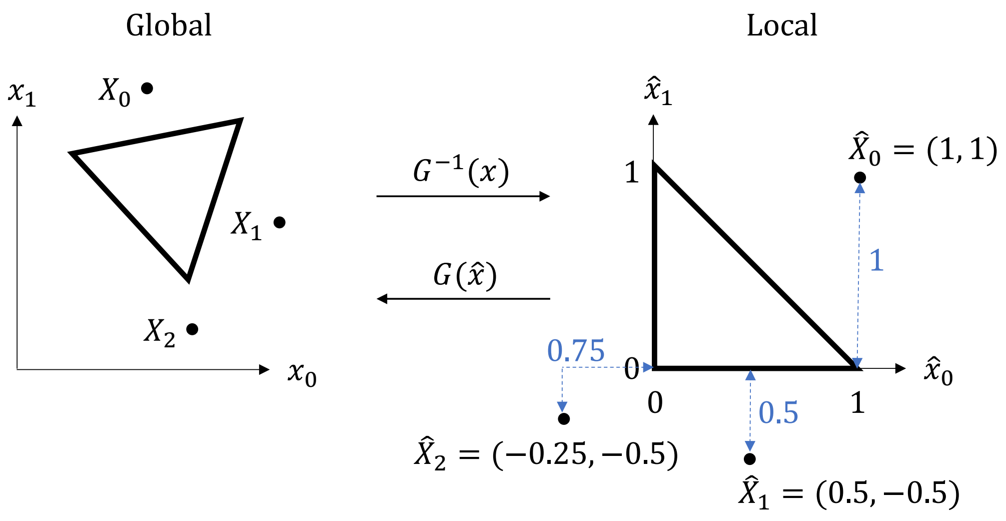


Figure 7.3: Each point X_i in the parent mesh local coordinate system $\hat{X}_i$ has the L^1 cell distance calculated and stored. For X_0 the distance is 1, for X_1 it's 0.5 and for X_2 it's 0.75.

Moving on to step (B), we (1) loop over each candidate cell, (2) find the point location

[2]see the API documentation at https://www.dealii.org/current/doxygen/deal.II/classBoundingBox.html

[3]see https://fenicsproject.org/olddocs/dolfin/1.4.0/python/programmers-reference/cpp/mesh/BoundingBoxTree.html

[4]see https://docs.fenicsproject.org/dolfinx/main/python/generated/dolfinx.geometry.html

[5]This step was the work of a masters student to whom I gave input. I tidied this and integrated it with the rest of the algorithm.

[6]For example, one can created a sorted list of the centroids of each cell, which can be quickly indexed with a simple binary (bisection) search.

in the reference coordinate space of each candidate cell (discussed in Sect. 4.5.1), then (3) choose the cell with the reference coordinate which is closest to the reference cell as measured in the L^1 norm (the 'Manhatten' distance - see Fig. 7.3). For information on how these are calculated see the appendix to this chapter (Sect. A.3.1). The L^1 distance and the cell number are stored for later use. In most cases a cell with zero L^1 distance is found, indicating a point is definitely inside, and the loop is terminated.[7] When a zero distance is not found, the lowest L^1 distance is used as long as it is below a tolerance. The relative tolerance that was specified when building the bounding boxes is used. Whilst intervals do not have bounding boxes, they require that this tolerance be set such that they appropriately drop points which are outside the mesh boundary[8].

This is far more robust than the previous cell location scheme. That had step (3) query whether the point in reference coordinates was located inside a candidate reference cell. As soon as a cell was found which contained the point, up to some user defined absolute tolerance (typically 10^{-14}), the loop was terminated.

The problems with this were manyfold. Early loop termination would cause points to be assigned to the wrong cells if the absolute tolerance was too high. A low tolerance risked the rare, but potentially catastrophic, case of floating point round-off and truncation errors at internal mesh cell boundaries causing points to not be assigned a cell. By choosing the cell that the point is closest to this is guaranteed not to happen. The tolerance parameter is now cell-size relative and exists only to ensure we capture points which are outside the domain.[9] The bounding boxes from step (A) were also not expanded to take account of any tolerance, meaning that cells at mesh or parallel domain edges that lined up with the coordinate axes could be lost.

It is difficult to compare the approach to step (B) outlined here with other libraries: in the deal.II particle-in-cell implementation article [89], for example, this is skipped over. The source code for both legacy FEniCS and FEniCSx reveal that a unique cell is found by searching for a geometric 'shortest distance'. This ought to make them resilient against point loss due to floating point arithmetic error, though it does not appear to be possible to specify a tolerance to catch points just outside mesh boundaries. In FEniCSx, at time of writing, this appears to be done by computing the distance between the point and the cell vertices via the iterative Gilbert–Johnson–Keerthi distance algorithm [94][10], though this is not clearly documented.

A method for producing the L^1 distance has been added to all cell types Firedrake uses. These are stored in the tabulation library, FIAT (discussed in Sect. 3.7). A reference coordinate is supplied and the L^1 distance from the reference cell is produced: This is carefully written to

[7]If there happen to be two cells which both have a zero L^1 distance whilst running in serial, the first to be identified is chosen.

[8]see future work (Sect. 7.8) for more on this

[9]To see how we deal with edges of parallel domains, see Sect. 7.5.

[10]see https://github.com/FEniCS/dolfinx/blob/c56d30b23f2c2ce99e8c145d3aeb43aff5d6d229/cpp/dolfinx/geometry/utils.cpp#L505-L543

ensure that a symbolic expression can be produced from FIAT's Python code using the SymPy library [95]. The point locations, reference coordinates and L^1 reference-cell distances are all calculated with generated C code.

The behaviour of the cell location algorithm and the tolerance have been tested extensively to ensure that points are indeed not lost, whilst profiles have shown no significant impact on cell location performance relative to the existing algorithm. Of course, for the purposes of point evaluation, being able to specify a data structure (the vertex-only mesh) upon which we may point evaluate more than once gives an overall performance gain.

7.5 Redistributing Point Coordinates: the 'Voting Algorithm'

As discussed in the motivation (Sect. 7.2), we must be able to redistribute point coordinates to the MPI rank containing the corresponding sections of the parent mesh domain. DMSwarm's in-built tool for this[11], which embeds a DMSwarm in a DMPlex, does not redistribute and is not compatible with many Firedrake meshes due to underlying differences between Firedrake meshes and PETSc DMPlexes. The redistribution algorithm is therefore implemented in Firedrake. The requirements of this algorithm are:

1. It must assign each unique coordinate to a mesh cell.
2. The mesh cell may be visible to more than one MPI rank in core and halo regions of the mesh; the coordinate should be assigned to all those ranks.
3. Coordinates should not be able to fall between the gaps of mesh partitions.
4. Coordinates should be assigned to a single cell, even if that cell appears on more than one mesh partition.
5. It should be clear after running the algorithm which coordinates are not in the domain.

Firstly we tackle the uniqueness of coordinates (requirement 1). Upon vertex-only mesh construction, there are two distinct situations which the user must be asked about:

A The same set of coordinates is being provided to all MPI ranks. The coordinates on all but one rank are therefore considered to be *redundant* and can be redistributed from a single rank.

B Different coordinates are supplied to each MPI rank. There is no implied redundancy and every coordinate supplied must be assumed to require embedding.

[11]`DMSwarmSetPointCoordinates` see https://www.mcs.anl.gov/petsc/petsc-3.9/docs/manualpages/DMSWARM/DMSwarmSetPointCoordinates.html

In both cases, there are n coordinates $\{X_i\}_0^{n-1}$. The index i is referred to as the *global index*: this is used as a unique identification number for a given point. In the redundant case (A), MPI rank 0 is assumed to have all n points and they are numbered sequentially. In the non-redundant case (B), all points are numbered in rank order: if rank 0 has 10 points, rank 1 has 20 points, and rank 3 has 5 points ($n = 35$), then rank 0's points have global indices 0-9, rank 1's points have global index 10-29, and rank 3's points have global indices 30-34.

Next, it is ensured that all MPI ranks have the same set of coordinates $\{X_i\}_0^{n-1}$: in the redundant case all MPI ranks are given the set of coordinates on rank 0 (an MPI broadcast operation from rank 0), whilst in the non-redundant case all ranks gather all the coordinates from each rank in rank order (an MPI all-gather operation).[12] All ranks now have the same set of points so the redundant and non-redundant cases no longer need to be treated separately.

For all coordinates $\{X_i\}_0^{n-1}$ the parent cell number (which is unique within an MPI rank), the reference coordinates, the L^1 distance from the reference cell and the MPI rank ownership now need to be identified. The cell location algorithm (Sect. 7.4) produces the first three of these, and the rank ownership can be found using the cell number.[13]

Now all coordinates which are within the meshed domain (including tolerance) will have been claimed by one or more MPI ranks. Where there are coordinates which lie near the boundaries of mesh partitions, cells on both sides of the partition may have laid claim to them thanks to the tolerance behaviour of the cell location algorithm. To decide between them the cell that has the lowest L^1 distance from the reference cell is chosen: in most cases this will be zero for one side of the partition and above zero for any others, allowing the points with larger L^1 distances to be discarded.

In most cases minimising the L^1 distance will resolve any rank conflicts. In the case of a tied L^1 distance there are two key scenarios to consider:

(i) Multiple ranks claim that a parent mesh cell on their partition owns the point. Other ranks may see the point, but not claim that they own it.

(ii) Each rank claims that a parent cell on a different rank owns the point.

Both of these cases involve a point exactly on the boundary between owned and halo regions. To resolve (i), the highest numbered rank is chosen as the owner: the other ranks that claimed the point are made to search again for the cell ownership (with the previously chosen cell excluded from the search), alongside ranks which didn't claim the point but did disagree about

[12] For very large numbers of points this step may limit performance, but it allows for the cell location algorithm being collective at present (i.e. having to be run once on each rank) and provides a working implementation that satisfies the requirements that were set out in the motivation. For a proposed improvement, see Future Work (Sect. 7.8).

[13] A Firedrake `Function` on the parent mesh Ω is created which contains one value per cell (P0DG(Ω)). The parent mesh rank number is then interpolated onto it. After a halo exchange is performed between parallel decompositions, to each MPI rank this `Function` contains the rank ownership information of all cells in the local mesh partition: the cell number of each coordinate indexes this to tell us the rank ownership of that coordinate's parent mesh cell.

the owner. All ranks which had claimed the point as their own are guaranteed to find a cell in the halo, with the same L^1 distance, that corresponds to the chosen rank since, on those ranks, the point was on a boundary between owned and halo regions.

For case (ii) to occur, the point must be on an owned/halo boundary on all ranks that claim to be able to see the point. To resolve (ii), the lowest numbered rank is chosen. Now the chosen rank, alongside any other disagreeing ranks, are also made to search again for cell ownership. In this case, all ranks are guaranteed to find a cell with the same L^1 distance which is owned by the chosen rank.

This cell-decision process is why we call this a 'voting algorithm': the cell a point is closest to wins, with ties broken depending on whether the point fits scenario (i) or (ii). The algorithm is summarised in Fig. 7.4.

This is implemented as follows: Coordinates which were not located by the cell location algorithm on a given rank are labelled as 'not locally visible'. The L^1 distance and the rank ownership of these 'not locally visible' coordinates are set to infinity. Ranks which claim to have identified a point as owned by themselves have the rank ownership made negative. The L^1 distances and rank ownerships are each arrays of length n which are stacked next to each other to create an $n \times 2$ shaped array. The lexicographic row-wise minimum[14] of this array is taken across all MPI ranks.[15] After returning all ranks to be positive, there is an agreed set of L^1 distances and rank owners for each coordinate on every MPI rank. Any coordinates which still have their L^1 distances and ranks as infinity must not be in the mesh: these can be counted for output. Where the L^1 distance has changed locally, the coordinate must have been claimed by another rank so it is marked as 'not locally visible'.

If the L^1 distance has not changed but the rank has, a process of cell re-identification is started: any previously identified cells are supplied to the cell location algorithm as cells-to-ignore, and a new cell given. When a cell is found that matches both the agreed L^1 distance and rank, the identification process stops. If no matches are found (which only happens for cells within halo boundaries), the coordinate is again marked as 'not locally visible'[16]. Such points are not included in the eventual DMSwarm embedding.

This satisfies all our requirements: Global indices ensure uniqueness (requirement 1). Taking the approach of marking points as 'not locally visible' ensures that coordinates are assigned to both core and halo cells as necessary (requirement 2). The voting algorithm taking the minimum L^1 distance ensures points cannot be lost between ranks (requirement 3). The voting algorithm tie breaker ensures assignment is unique (requirement 4). The algorithm also tells us how many coordinates are not found on any parallel partition of the mesh (requirement 5)

[14]i.e. first comparing the two arrays by their first column, returning the element-wise minimum, with ties broken by comparing the second column element wise

[15]Combining values from all MPI ranks, performing an operation on them, and redistributing them back to all ranks is known as an MPI allreduce operation: this is performed with a newly implemented lexicographic row-wise minimum as the MPI operator.

[16]this may be susceptible to floating point errors - see Future Work, Sect. 7.8

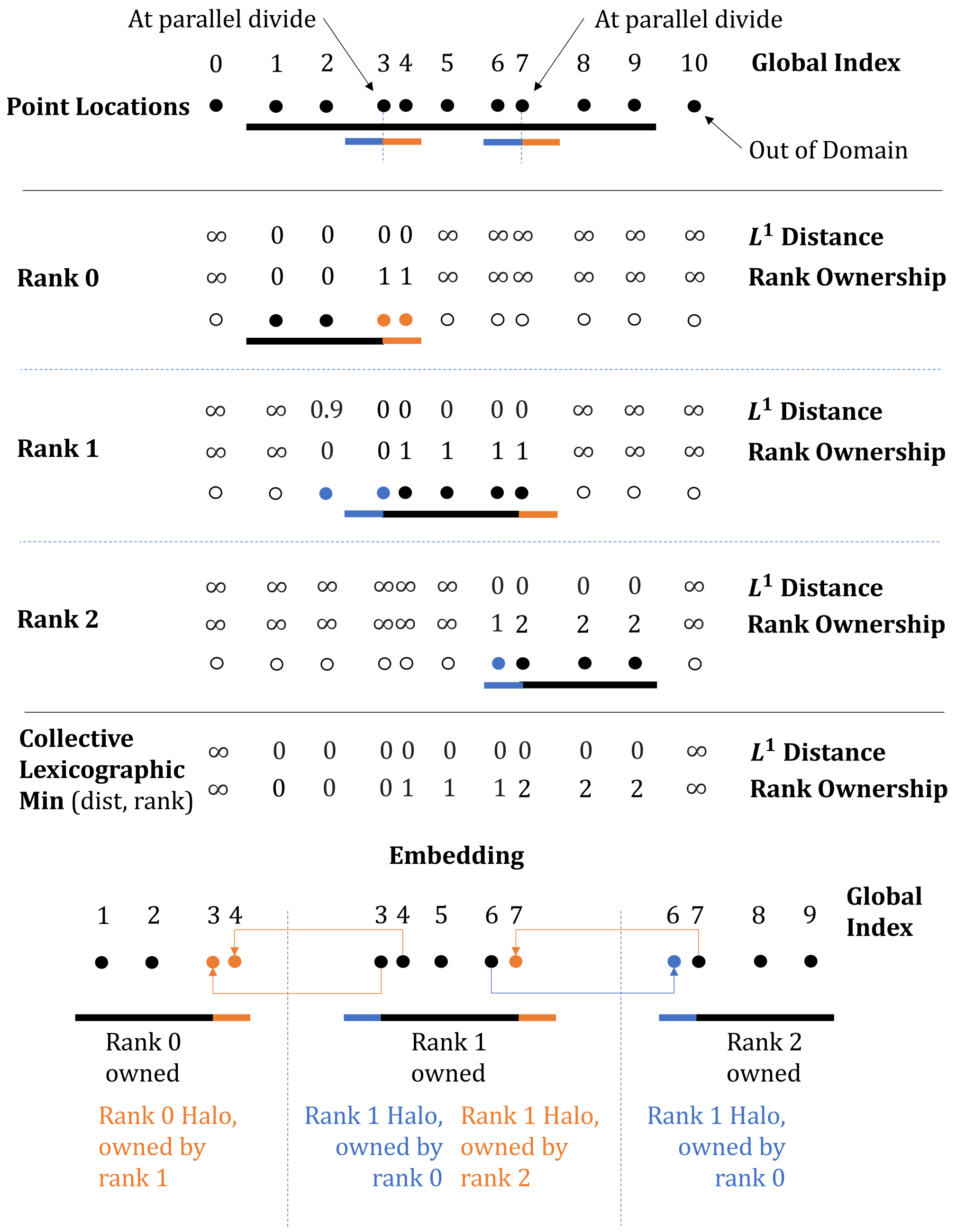


Figure 7.4: Embedding 10 point locations into a parallel domain decomposed interval mesh. Points 0 and 10 are outside of the domain. Point 2 is identified as being inside a cell on rank 0 (it has 0 L^1 distance) but is also identified as being inside a cell on rank 1, albeit with nonzero L^1 distance. The collective lexicographic minimum operation chooses the lower distance so assigns it to a cell on rank 0. Points 3 and 7 are in the unusual situation of being exactly on a parallel decomposition boundary. Point 7 fits scenario (i) and point 3 scenario (ii). For point 7, both ranks are set to be negative and the collective lexicographic minimum operation breaks the tie by choosing the most negative rank (-2). After restoring all ranks to be positive, the result is the higher rank. Rank 1 is made to search for the cell again until it finds another cell with zero L^1 distance on rank 1. For point 3, the lowest rank is chosen. The cell is searched for again until the point is identified as being on rank 2, still with zero L^1 distance. In the final embedding one can see how points have been uniquely assigned: duplicate global indices occur for points in the halos.

which allows us to easily raise a warning or error.

Both FEniCS and FEniCSx do not deal with this aspect of point location directly, though they contain tools that let the user implement a parallel safe point location algorithm themselves: After performing a cell search the user either finds a cell or not, where multiple cells are found one must be chosen. Maljaars, Richardson, and Sime [90] do not explicitly document how they decide on ownership where there are disagreements between MPI ranks in LEoPart - the published source code[17] does not appear to deal with this case. Deal.II has a `distributed_compute_point_locations` function which the `Particles` library uses for point redistribution. This, presumably, relies on some kind of voting algorithm to deal with disagreements at both cell boundaries and parallel domain boundaries, but this is not documented.

Nevertheless, the algorithm presented here is most likely not entirely novel. Notably, appendix A.3 of Maddison [96] contains a very similar algorithm to that presented here, though does not deal explicitly with domain containment.

7.6 Redistributing Point Data

Having redistributed our coordinates we now need a method of redistributing the data associated with them. When a vertex-only mesh is created some list of coordinates is specified: the values associated with them will in all likelihood be in the same order. We therefore need

1. a way to associate the data with the input coordinates and

2. a way to map those to the coordinates of our vertex-only mesh.

This is a challenge which other libraries do not face, due to the novel nature of the distinction made here between the coordinates (the vertex-only mesh) and the data (the P0DG function space on the vertex-only mesh).

7.6.1 The Input-Ordering Vertex-Only Mesh

We start with (1.). The input coordinates can be thought of as a vertex-only mesh in their own right. Rather than having coordinates redistributed to appropriate parent-mesh cell locations, this vertex-only mesh maintains the order and MPI rank of the input coordinates. Crucially, data associated with this mesh, such as Firedrake `Functions`, are stored in the order and on the MPI rank of the input coordinates, allowing their direct access or modification. The mesh is referred to as the input-ordering vertex-only mesh or $\Omega_v^{\text{input-ordering}}$ associated with a vertex-only mesh Ω_v.

[17]https://bitbucket.org/jakob_maljaars/leopart

7.6.2 Redistribution as Interpolation

Moving on to (2.) then, we need to move data to and from $\text{P0DG}(\Omega_v^{\text{input-ordering}})$ and $\text{P0DG}(\Omega_v)$. The input-ordering vertex-only mesh $\Omega_v^{\text{input-ordering}}$ may or may not have the same number of vertices as the vertex-only mesh immersed in the parent mesh Ω_v since some coordinates may have been deemed to be outside the parent mesh

$$\Omega_v \subseteq \Omega_v^{\text{input-ordering}}. \tag{7.6.1}$$

The bases and dual bases of these will be similar

$$\text{P0DG}(\Omega_v) \subseteq \text{P0DG}(\Omega_v^{\text{input-ordering}}) \text{ and} \tag{7.6.2}$$

$$\text{P0DG}(\Omega_v)^* \subseteq \text{P0DG}(\Omega_v^{\text{input-ordering}})^* \tag{7.6.3}$$

so, despite the vertex-only meshes only being defined at the vertex locations, the movement between the function spaces can be thought of as a dual evaluation interpolation operation

$$\mathcal{I}_{\text{P0DG}(\Omega_v)} : \text{P0DG}(\Omega_v^{\text{input-ordering}}) \to \text{P0DG}(\Omega_v). \tag{7.6.4}$$

To be precise, this interpolation operation which performs our redistribution is a map from the input ranks and input order of our coordinates (represented by $\Omega_v^{\text{input-ordering}}$) to the ranks assigned by the voting algorithm and the rank-local data layout of Ω_v.

$\mathcal{I}_{\text{P0DG}(\Omega_v)}$ can be considered a permutation that may operate on a subset of its domain. Say we have

$$\Omega_v^{\text{input-ordering}} = \begin{pmatrix} X_0^{\text{input-ordering}} \\ X_1^{\text{input-ordering}} \\ X_2^{\text{input-ordering}} \\ X_3^{\text{input-ordering}} \end{pmatrix} \tag{7.6.5}$$

$$\Omega_v = \begin{pmatrix} X_0 \\ X_1 \\ X_2 \end{pmatrix} = \begin{pmatrix} X_1^{\text{input-ordering}} \\ X_0^{\text{input-ordering}} \\ X_2^{\text{input-ordering}} \end{pmatrix}. \tag{7.6.6}$$

Here, $X_3^{\text{input-ordering}}$ is outside the domain of the parent mesh of Ω_v. A function $r \in \text{P0DG}(\Omega_v^{\text{input-ordering}})$ has four coefficients $\{r_i\}_{i=0}^3$ on each $\{X_i^{\text{input-ordering}}\}_{i=0}^3$ which $\mathcal{I}_{\text{P0DG}(\Omega_v)}$maps to a function $l \in \text{P0DG}(\Omega_v)$ with coefficients $\{l_i\}_{i=0}^2$ on each $\{X_i\}_{i=0}^2$. We then have

$$\underbrace{\begin{pmatrix}0 & 1 & 0 & 0\\ 1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\end{pmatrix}}_{\mathcal{I}_{\text{P0DG}(\Omega_v)}}\begin{pmatrix}r_0\\ r_1\\ r_2\\ r_3\end{pmatrix} = \begin{pmatrix}r_1\\ r_0\\ r_2\end{pmatrix} = \begin{pmatrix}l_0\\ l_1\\ l_2\end{pmatrix} \tag{7.6.7}$$

$$= \underbrace{\begin{pmatrix}0 & 1 & 0\\ 1 & 0 & 0\\ 0 & 0 & 1\end{pmatrix}}_{\boldsymbol{A}}\begin{pmatrix}r_0\\ r_1\\ r_2\end{pmatrix} \tag{7.6.8}$$

Here, $\mathcal{I}_{\text{P0DG}(\Omega_v)}$ has no inverse since the matrix is non-square and we cannot recover r_3. The inverse of $\boldsymbol{A}$ is its transpose $\boldsymbol{A}^T$ since permutations are orthogonal. In general, if all vertices of $\Omega_v^{\text{input-ordering}}$ are present in Ω_v, $\mathcal{I}_{\text{P0DG}(\Omega_v)}$ is a permutation and its inverse is the transpose. The interpolation operation in the other direction

$$\mathcal{I}_{\text{P0DG}(\Omega_v^{\text{input-ordering}})} : \text{P0DG}(\Omega_v) \to \text{P0DG}(\Omega_v^{\text{input-ordering}}) \tag{7.6.9}$$

is therefore only strictly defined where all points in $\Omega_v^{\text{input-ordering}}$ are present in Ω_v. This makes sense; one cannot evaluate a dual basis function that uses a point which is not present in its domain! In such cases

$$\mathcal{I}_{\text{P0DG}(\Omega_v^{\text{input-ordering}})} \equiv [\mathcal{I}_{\text{P0DG}(\Omega_v)}]^T \tag{7.6.10}$$

and vice-versa.

The adjoint to our interpolation operations are the left multiplication of the matrix by the row vector of cofunction coefficients (this is discussed in detail in Sect. 5.4.1). For

$$\mathcal{I}^*_{\text{P0DG}(\Omega_v)} : \text{P0DG}(\Omega_v)^* \to \text{P0DG}(\Omega_v^{\text{input-ordering}})^* \tag{7.6.11}$$

with $l^* \in \text{P0DG}(\Omega_v)^*$ that has coefficients $\{l_i^*\}_{i=0}^2$ this is

$$\begin{pmatrix}l_0^* & l_1^* & l_2^*\end{pmatrix}\underbrace{\begin{pmatrix}0 & 1 & 0 & 0\\ 1 & 0 & 0 & 0\\ 0 & 0 & 1 & 0\end{pmatrix}}_{\mathcal{I}_{\text{P0DG}(\Omega_v)}} = \begin{pmatrix}l_1^* & l_0^* & l_2^* & 0\end{pmatrix} = \begin{pmatrix}r_0^* & r_1^* & r_2^* & r_3^*\end{pmatrix}. \tag{7.6.12}$$

Equivalently, we can start with a column vector of coefficients

$$\underbrace{\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \\ 0 & 0 & 0 \end{pmatrix}}_{[\mathcal{I}_{\mathrm{P0DG}(\Omega_v)}]^T} \begin{pmatrix} l_0^* \\ l_1^* \\ l_2^* \end{pmatrix} = \begin{pmatrix} l_1^* \\ l_0^* \\ l_2^* \\ 0 \end{pmatrix} = \begin{pmatrix} r_0^* \\ r_1^* \\ r_2^* \\ r_3^* \end{pmatrix}. \tag{7.6.13}$$

7.6.3 Implementation

Input-ordering Vertex-Only Mesh

The input-ordering vertex-only mesh $\Omega_v^{\text{input-ordering}}$ is created directly from the input coordinates and is accessed via an `.input_ordering` property of a given vertex-only mesh Ω_v. This still sits on top of a DMSwarm and all DMSwarm 'fields' present in Ω_v are reordered to correspond to entries in $\Omega_v^{\text{input-ordering}}$. It doesn't necessarily make sense that $\Omega_v^{\text{input-ordering}}$ have a parent mesh, since it is not necessarily immersed in another mesh. Nevertheless, to avoid modifying the vertex-only mesh constructor, we make Ω_v the parent.

Star Forests for Interpolation

The PETScSF or 'Star Forest' communication layer found inside PETSc [97] is designed to deal with exactly the kind of parallel communication challenge we face. This provides an API which allows one set of MPI ranks and data layouts to be associated with another set of MPI ranks and data layouts such that data can be sent between them.

A rank with a local index into that rank is called a *root*. Each root can be associated with zero or more other tuples of rank and local index: each of these is called a *leaf*. The tree of links between a root and any associated leaves is called a *star*, whilst the union of all the stars is called a *star forest*. For an illustration of stars see Fig. 1 in Zhang et al. [97] whilst a star forest is shown in Fig. 2.

The star forest is a complete description of connectivity between two parallel data structures. Sending all root data to associated leaves is called an SF broadcast, whilst sending all leaf data to associated roots is an SF reduce. For more see Sect. 3.2 in Zhang et al. [97]. In many cases these can be thought of as parallel distributed binary matrix operations. The $\mathcal{I}_{\mathrm{P0DG}(\Omega_v)}$ in Eq. 7.6.8 is an example of an SF broadcast. The roots are $\{r_i\}_{i=0}^3$ and the leaves are $\{l_i\}_{i=0}^2$. r_3 is a root with no associated leaf. The reduce operation here is $\boldsymbol{A}^T$

$$\underbrace{\begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \\ 0 & 0 & 1 \end{pmatrix}}_{\boldsymbol{A}^T} \begin{pmatrix} l_0 \\ l_1 \\ l_2 \end{pmatrix} = \begin{pmatrix} l_1 \\ l_0 \\ l_2 \end{pmatrix} = \begin{pmatrix} r_0 \\ r_1 \\ r_2 \end{pmatrix} \tag{7.6.14}$$

which only assigns values to the roots which have associated leaves. SF reduce can therefore be used to represent adjoint interpolation where cofunction coefficients are taken as vectors to right multiply (Eq. 7.6.13), so long as the output cofunction has its coefficients set to zero where the matrix-vector product does not provide a value. We get a permutation matrix when each root has 1 leaf: the reduce operation in this case is both the transpose and inverse of broadcast. SF reduce can then also be used to represent interpolation in the other direction (Eq. 7.6.9) where it is defined.

The data layout of P0DG functions on vertex-only meshes corresponds to the layout of the meshes themselves. All the information needed to implement the interpolation operation

$$\mathcal{I}_{\text{P0DG}(\Omega_v)} : \text{P0DG}(\Omega_v^{\text{input-ordering}}) \to \text{P0DG}(\Omega_v) \tag{7.6.15}$$

is therefore contained in the data ordering and rank information of $\Omega_v^{\text{input-ordering}}$ and Ω_v. $\Omega_v^{\text{input-ordering}}$ is already arranged as necessary: for vertices specified on rank i in order $X_j^{\text{input-ordering}}$, the roots are tuples (i, j). The leaves are the non-halo vertices of Ω_v: for vertices now on rank k in order X_l, the leaves are tuples (k, l).

Why non-halo vertices? As a rule, dual evaluation interpolation operations in Firedrake do not affect halo regions. This makes sense: halos are an implementation detail, so we do not need to worry about Ω_v in parallel having the same coordinate X_i on different processes. Mathematically, each X_i is unique so the permutation-like operation in Eq. 7.6.8 remains valid. Data associated with any X_i in halo regions are updated after a halo exchange.

As mentioned in Sect. 7.6.2, where the input-ordering vertex-only mesh $\Omega_v^{\text{input-ordering}}$ contains points which are not in Ω_v (they were not found in the parent mesh) interpolation is strictly not defined. Nevertheless, the interpolation operator for points which are in both vertex-only meshes is created and the values at missing points are left unmodified. This is an SF reduce acting as a permutation. The adjoint interpolation, an SF broadcast, similarly ignores missing points. This is used for cross-mesh interpolation, discussed in Chapter 8.

The star forest graph[18] is embedded in Firedrake's interpolation routines: Where a vertex-only mesh and its input-ordering are both found as either source or target meshes, the input-ordering star forest is requested, and an SF broadcast (for $\mathcal{I}_{\text{P0DG}(\Omega_v)}$ or $\mathcal{I}^*_{\text{P0DG}(\Omega_v^{\text{input-ordering}})}$) or SF reduce (for $\mathcal{I}^*_{\text{P0DG}(\Omega_v)}$ or $\mathcal{I}_{\text{P0DG}(\Omega_v^{\text{input-ordering}})}$) operation chosen. This is either executed on a given `Function` or `Cofunction`, or held as a partially applied function in a Firedrake `Interpolator` for execution when one is specified. This is a 'matrix-free' approach - we never assemble the binary interpolation matrix itself.

[18]accessed via an `.input_ordering_sf` property of a vertex-only mesh

7.7 Other Parallel Considerations

7.7.1 Constructing Halos

Star Forest

Star forests are used in Firedrake for halo exchange. This involves moving data stored upon mesh topological entities (vertices, edges, facets and cells) from the edges of one parallel partition to another. All the topological entities in a partition's halo are leaves. In each case the leaf tuple is the current rank and local index in memory of the entity being considered. The root tuples are, for each halo entity, the remote rank which owns the entity and the local index at which it is found on that remote rank. To update the values stored on a mesh halo, typically in a Firedrake `Function`, an SF broadcast operation is performed.[19] This is described in more detail in Lange et al. [47].

This SF is created by default for most Firedrake meshes when the DMPlex is constructed and is attached to the DMPlex: it is referred to as the *point SF* since topological entities in a DMPlex are referred to as 'points'. When constructing a vertex-only mesh, the point SF for a DMSwarm is not created automatically: this is instead done manually at mesh construction to ensure halo exchange works.

For a vertex-only mesh each entity is a vertex. The halo regions consist of vertices which are in the halos of the parent mesh. The voting algorithm (Sect. 7.5) says which ranks own these vertices: on a given MPI rank, these are the remote ranks which are the first entries in the root tuple of each vertex. The local index of each vertex on its remote ranks (i.e. the index in memory of the coordinate with the same global index) is the second entry in each root tuple. These are identified and the point SF set.

Assigning PyOP2 Labels

Halo topological entities in a DMPlex or DMSwarm are given labels by PyOP2 to identify them. This aids the execution of parallel loops of a given 'kernel', such as interpolation, over the whole parallel distributed mesh. The kernel for the point evaluation dual evaluation operation is discussed in Sect. 4.5.3. On a given rank the PyOP2 labels are:

- `core`: Data stored on these entities do not need to wait for halo exchange before the PyOP2 kernel performs calculations with them. These are 'owned' by the rank, i.e. they are a root (with or without leaves) in the DM point SF.

- `owned`: Data stored on these entities do need to wait for halo exchange before the PyOP2 kernel performs calculations with them. These are also 'owned' by the rank, as per the definition above.

[19]Non-halo entities are roots without leaves - the broadcast operation does not affect them.

- `ghost`: Data stored on these entities are 'owned' by another rank: these can be labelled as either `core` or `owned` on that rank.

All entities receive a unique label. When Firedrake is run in serial, all mesh entities are labelled as PyOP2 `core` since no halo exchange will occur.

For a DMPlex, these labels are derived from the point SF via the connectivity information contained in the DMPlex. Since DMSwarm represents the topology of a mesh of *disconnected* vertices, the existing algorithm does not identify any vertices as `owned`. This is a problem: when we loop over the vertices of a vertex-only mesh to interpolate from a parent mesh, we need to ensure that the vertex-only mesh PyOP2 cell label, stored on the DMSwarm, matches that of the parent mesh DMPlex. If we don't, the PyOP2 kernel will make incorrect assumptions about halo exchange and will not execute correctly. Fortunately, the simple solution is to inherit the DMSwarm PyOP2 label from the DMPlex: this is done after cell location.

7.8 Future Work

The cell-location algorithm (Sect. 7.4) can be made more flexible. As yet, there is no accommodation for users who have a very coarse mesh and want to ensure all points outside it are dropped (a bounding box tolerance in step (A) of zero), but still want to guarantee points within the mesh cannot be lost (no tolerance for choosing the L^1 bounding box in step (B)). As stated in Sect. 7.4, the second tolerance is only needed because bounding boxes cannot yet be constructed for intervals in 1D. Available options are to either only use the L^1 bounding box tolerance for intervals (perhaps the easiest option) or to allow bounding boxes to be drawn for them. The latter option would either require a library other than libspatialindex or manual implementation of a 1D spatial index. Should a different library be sought, the libraries investigated by Lawson, Gropp, and Lofstead [91] could provide a starting point. As is noted there, changing library may also mean changing the database-lookup structure (spatial index): the pros and cons of such a change would need to be considered given the relative ease of implementing a simple 1D spatial index. A method for marking the outer boundaries of meshes prior to parallel decomposition would also be needed to avoid point loss between parallel boundaries.

The cell location algorithm's bounding boxes also currently preclude a safe cell-search algorithm for high-order meshes. This is because the cell extrema, around which the bounding boxes are created, are assumed to be at the vertices. A high order mesh may have a bend between two vertices which extends outside the bounding box. Put another way, when the mesh coordinates function is in a higher degree polynomial finite element function space, the values saved as the global coefficients do not necessarily encapsulate the extrema of the function. Projecting the coordinates function to a basis of Bernstein polynomials, the coefficients for which will always contain the extrema, could provide a solution.

More pressingly, the cell-location algorithm's performance is limited to the speed of a Python loop. This is because the generated C code includes a call to the Python interpreter. The exact cause of this has yet to be identified, though it could be the SymPy code generation. This has not yet inhibited uptake of the vertex-only mesh since its performance is comparible to the existing `.at` method of Firedrake `Functions` and enables plenty of new use cases. Nevertheless, moving points (discussed in Sect. 10.2.2) will likely need significant performance improvements when using large numbers of particles.

Where a point is outside a mesh boundary, but is included due to the tolerance a relevant question is "what value should we give the point when we perform point evaluation?" At present, the value of the basis functions of the assigned mesh cell, extrapolated to the point location, is used. So, for a ramp function $f(x) = x$ on a line mesh $0 \leq x \leq 1$, with a sufficiently high tolerance that $x = -0.1$ is considered to be on the line, the point evaluation will be $f(-0.1) = -0.1$. This may not be the correct behaviour for some use cases: it might be better to project the point onto the closest on-mesh location and return the value there. In the case of $x = -0.1$ we would project this onto $x = 0$ then evaluate $f(-0.1) = 0$. The simplest approach would be to do this after transforming to reference coordinates: where the L^1 distance is above 0, the output reference coordinates would be the projection of the given reference coordinate onto the closest point on the reference cell.

The implementation of the voting algorithm (Sect. 7.5) may be susceptible to floating point arithmetic error causing points to either be lost or halos to not accurately reflect the cell locations. The algorithm relies on being able to find an exact match for rank and L^1 distance when points are on owned/halo boundaries in order to re-identify the owning cell. Of course, floating point arithmetic error could, in rare scenarios, result in no exact match being found, causing points to be lost. To avoid this, the implementation should specifically identify points which are exactly on owned/halo boundaries (as opposed to halo/halo boundaries), then enforce and communicate cell ownership.

The voting algorithm's performance can also be improved. At present, the voting algorithm requires all-to-all rank communication in the non-redundant case such that all MPI ranks have the same set of points. Where this is found to limit performance, a more intelligent approach could be used: Consider each rank as having a pool of coordinates to distribute. Firstly we can remove from the pool any coordinates which are known to be on the core section of the mesh and definitely not in any mesh halos by querying the rank-local bounding boxes (see Sect. 7.4). This would require more bounding box querying routines to be exposed at the level of vertex-only mesh construction than is currently the case.

We can then redistribute the remaining points: this can take a similar approach to cell location identification (see Sect. 7.4) this time applying axis-aligned bounding boxes around each MPI rank's decomposed mesh section. The coordinates of each point would provide candidate ranks which could then vote on the point amongst themselves. For this to provide significant performance improvements the cell location algorithm would probably need to be made non-collective. This approach is similar to that taken in LEoPart [90] and the NESO-Particles library [98].

As a result of the matrix-free approach to interpolation from and to the input-ordering vertex-only mesh, the interpolation matrix itself is inaccessible. The matrix free approach has advantages of course, interpolation matrices may be very large given the number of degrees of freedom involved. Indeed, when otherwise performing adjoint interpolation in Firedrake, we assemble the whole interpolation matrix and left multiply it by the given row-vector of cofunction coefficients. That is a problem which needs fixing! To fix the problem at hand, some way of turning a star forest graph into an equivalent logical matrix would need to be found.

7.9 Summary of Contributions

I have implemented vertex-only meshes in Firedrake such that they satisfy my ideal requirements for parallel domain decomposition (see the motivation, Sect. 7.2). A voting algorithm, outlined in Sect. 7.5, ensures that points can be specified on one rank and be correctly redistributed to another rank. I guarantee that points are embedded in the domain if they are within it using a robust cell location algorithm, described in 7.4. The voting algorithm ensures that the assigned cell is unique within the complete parallel distributed mesh.

In comparison to other similar libraries, this is beyond the present capability of FEniCS and FEniCSx, and makes Firedrake at least comparable to deal.II. The additional flexibility of Firedrake's cell-location and voting algorithm allow points just outside the mesh but inside the true domain to be captured.

To cope with my split of point data coordinates (the vertex-only mesh) and data at those coordinates (the P0DG function space), I have introduced an input-ordering vertex-only mesh alongside an interpolation operator to move from one to the other. I have implemented this in Firedrake using PETSc Star Forests. This satisfies the requirement of being able to unambiguously read and write P0DG functions on the vertex-only mesh in parallel. The interpolation operation is differentiable; this allows for differentiable cross-mesh interpolation, discussed in Chapter 8.

I have also ensured that Firedrake vertex-only meshes have halos for data exchange across parallel partitions. This makes them comparable to other Firedrake meshes and allows PDEs to be solved on them in parallel, as demonstrated for point forcing (Sect. 4.6.1).

Chapter 8

Arbitrary, Parallel Safe, Differentiable, Mesh to Mesh Interpolation

The following text and code examples reuse, in places, work I contributed to the Firedrake manual [5].

8.1 Motivation

Interpolation across meshes is useful wherever we want to move a finite element function from a function space on one mesh, onto another function space on another mesh.

Prior to this work, this had been implemented in Firedrake for the special case of *geometric multigrid methods* where PDE solutions are found by solving on a mixture of coarse and fine meshes of the same domain. Solutions would be moved from a coarse mesh to a fine mesh using dual evaluation interpolation (referred to as a *prolongation* operation). The adjoint to this interpolation operation is referred to as the *restriction* operation and is used to down-sample cofunctions on the fine mesh to the coarse mesh. Lastly there is an *injection* operation which dual evaluates functions on fine meshes back into coarse meshes. Firedrake's multigrid infrastructure relies on well defined refinement relationships between the meshes. Generally a hierarchy of mesh refinements is automatically generated: all the vertices of a coarse mesh appear at the same locations in the refined mesh and the parallel domain decomposition is maintained (see Fig. 8.1). This reliance on automatic generation is very limited. If one wanted

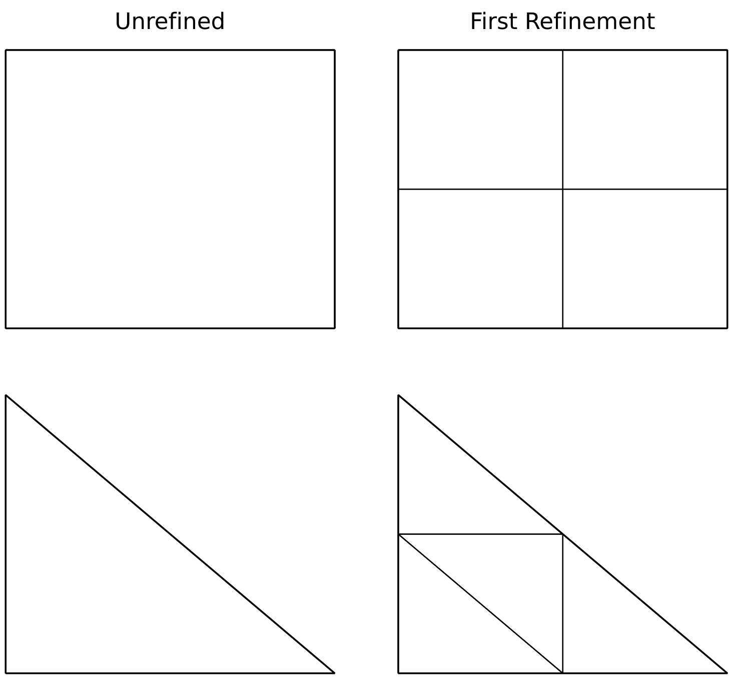


Figure 8.1: The geometric effect of a single mesh refinement on quadrilaterals and triangles in Firedrake. All vertices prior to refinement are present after refinement.

to move solutions between the meshes in Fig. 8.2 (a so-called 'nonnested' hierarchy of meshes), one has to provide the links between coarse and fine cells manually and ensure that the parallel domain decompositions match. Nonnested multigrid is known to be optimal for certain classes of problems (see for example Scott and Zhang [99]) so providing a straightforward method of creating the necessary operators would make the use of these methods in Firedrake much more straightforward.

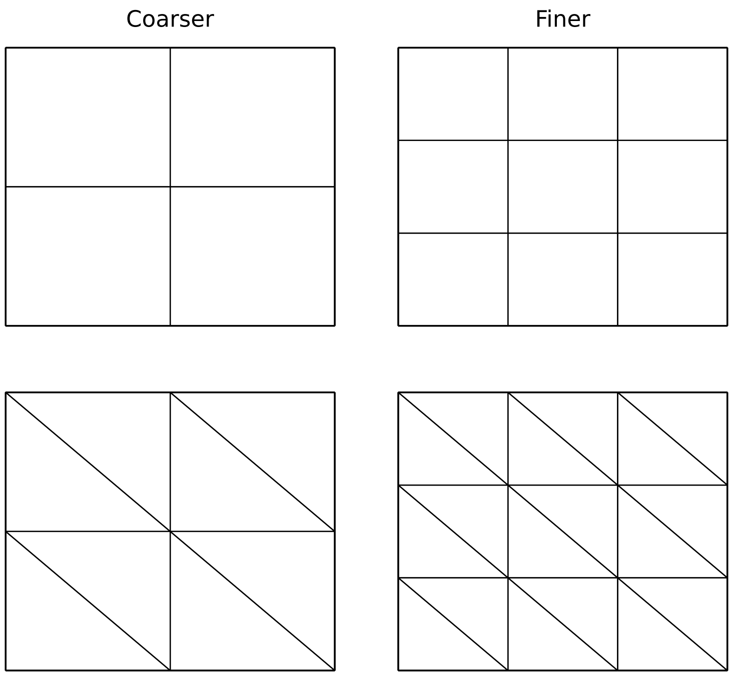


Figure 8.2: Four different ways of meshing the same square domain. Not all vertices in the coarser meshes are present in the finer meshes, so, prior to this work, there was no straightforward way to interpolate between them in Firedrake.

Other, as yet unrealised, uses of mesh-to-mesh interpolation abound: one might wish to mesh a domain with different cell shapes and examine the effects on the solution by interpolating from one mesh to the other or to a third, finer mesh which has a well defined mathematical relationship to both meshes. If meshes change in response to some metric, such as a changing solution with time, one ideally should be able to interpolate from the mesh at one time step to another to, for example, advance a time stepping algorithm.[1]

Being able to evaluate the dual basis of a mesh along an arbitrary domain, such as an embedded line, surface or volume, would allow diagnostics to be calculated on a function. Take, for example, the function

$$f(x, y) = x \times y \tag{8.1.1}$$

represented as a finite element function on a 2D mesh, with a line

$$L = (0, 0) \text{ to } (1, 1) \tag{8.1.2}$$

[1]The stability and conservation behaviour of such an algorithm would depend on the user's implementation: whilst dual evaluation interpolation avoids introducing new maxima and minima in finite element functions (which Galerkin projection does not), conservation of quantities generally requires the use of 'supermeshes' (see [100]).

along it (see Fig. 8.3). The line integral is

$$\int_L f(x,y)\ ds = \sqrt{2}/3. \tag{8.1.3}$$

If we represent the 1D line in 2D space (making it an immersed manifold) we ought to be able to (a) find the values of the function f along the line by performing interpolation onto an appropriately defined function space then (b) evaluate the line integral by assembling the appropriate expression.

```
# (a)
f_line = interpolate(f, V_line)
# (b)
assemble(f_line * dx)   # = sqrt(2) / 3
```

This could be extended to appropriately defined surfaces and volumes.

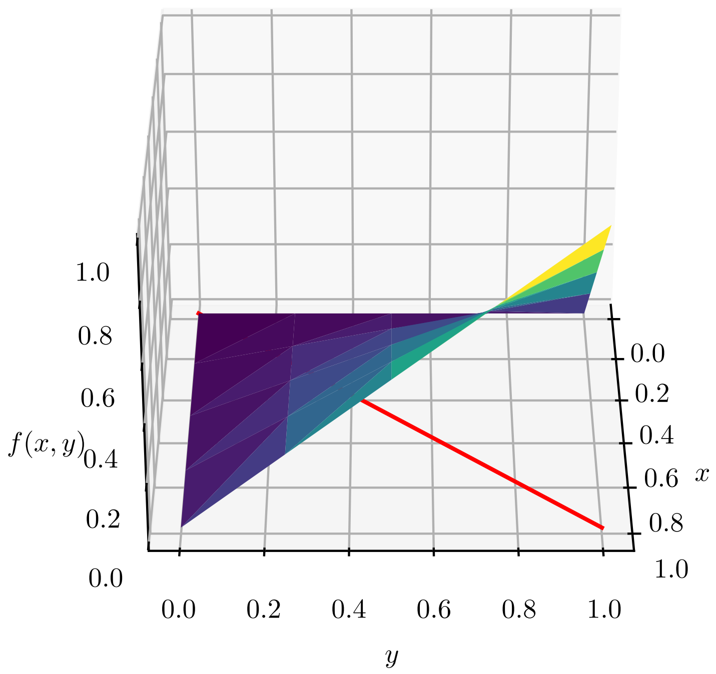


Figure 8.3: The function $f(x,y) = x \times y$ on a meshed square domain with the line $L = (0,0)$ to $(1,1)$ to integrate along.

Generic cross-mesh interpolation, particularly with the ability to interpolate over domains that only partially match, would allow simulations to be coupled together. For fluid structure interactions, one could solve the appropriate approximation to the Navier Stokes equations for the fluid in the fluid domain, interpolate the solution from the fluid mesh onto a function on the structure mesh (which would update the values of coefficients on boundaries on the structure), use the interpolated function as a boundary condition on the structure along the interface, then use those values to solve a PDE on the structure. Strong (Dirichlet) boundary conditions will enforce a 1-way coupling by setting the value at the interface. Weak boundary conditions, which are enforced as part of the problem formulation (the variational form expressed in UFL) such as Neumann boundary conditions, allow for two way coupling: in a most basic sense, one can iteratively interpolate a solution on the structure back into the fluid domain and repeat[2]. Once interpolation can be expressed symbolically (see Chapter 9) the interpolation of UFL arguments will be expressible between the two domains: this would allow the system to be solved monolithically. Since all operations are compatible with automated adjoint generation via pyadjoint (see Sect. 5.1, specifically Sect. 5.4) it will be possible to easily express and solve optimisation problems involving multiple interacting problems.

It should be noted that, in limited circumstances, Galerkin projection can already be performed across meshes. This, however, produces a different result to interpolation (see Sect. 3.3) and requires a global solve so is not always appropriate.

8.2 Existing Capability Outside Firedrake

Parallel compatible cross mesh interpolation with a differentiable interpolation operator, within the domain of code generating Finite Element Method frameworks, has not, to the author's knowledge, been implemented before. The closest would be cross-mesh interpolation in FEniCS[17]: when it was under active development it maintained compatibility with pyadjoint (see Sect. 5.4) to allow differentiation of these operations. However, it only worked in serial[3] which limited its usefulness to small problems. The FEniCS rewrite, FEniCSx recently added support for cross-mesh interpolation in parallel[4], though, at the time of writing, FEniCSx does not support automated adjoint generation for these operations.

Other FEM code generation libraries also support cross-mesh interpolation, deal.II[33] for example has a `SolutionTransfer` tool which works in parallel. The lack of explicit high-level differentiation operations in the libraries means that finding derivatives of these operations requires the use of traditional AD tools[5] which, though powerful, have downsides [60] which pyadjoint mostly avoids (see Sect. 5.4). MFEM [31, 32] also allows cross mesh interpolation, seemingly aimed at grid transfer operations in multigrid[6] and has its own AD tool[7] which, in principal, could work with the cross-mesh interpolation tools though this is not documented. The DUNE framework also supports multigrid [101] though is not documented as being compatible with any form of AD. NGSolve's [22] dual basis evaluation functionality is not documented as supporting multiple domains, but they do support multigrid methods; meanwhile the au-

[2]this can be thought of as a sort of home-made Additive Schwartz method

[3]see the FEniCS documentation

[4]see https://github.com/FEniCS/dolfinx/pull/2245

[5]for Deal.II see https://www.dealii.org/current/doxygen/deal.II/group__auto__symb__diff.html

[6]see the API documentation for the finite element class https://docs.mfem.org/2.0/classFiniteElement.html and various tutorials referencing multigrid

[7]see https://mfem.org/autodiff/

tomatic differentiation they document is seemingly limited to symbolic differentiation of their UFL-like expressions: in principal this could be used to build something like pyadjoint though no one has done this. FREEFEM [21] has `MatPtAP` and `transferMat` which implements a 'fast interpolation algorithm' between non-matching meshes, though this does not use dual evaluation, rather it is a home-baked interpolation operator documented on their website[8].

8.3 Mathematics and Firedrake Implementation

Internally, all Firedrake basis function coefficients are constructed by performing one or more point evaluations as described in Sect. 3.7. Dual evaluation interpolation operations always boil down to sums of point evaluations (see Eq. 3.7.2). If we find the locations of these points in the global coordinate system of the source mesh, we merely have to perform the point evaluations on the source function to reconstruct our basis function coefficients. By using the global coordinate system, we avoid needing to worry about finding a valid extension to each local dual basis functional in the source mesh reference cells (see Sect. 3.2).

Whilst this sounds straightforward, it has not been implemented for several reasons:

1. Meshes have different cell numbering and different parallel domain decomposition. This means that when given the same list of point locations tied to cells of two different meshes, one finds that the list is split differently across MPI ranks depending on which mesh is being considered. See Fig. 8.4. When point evaluating on the source mesh, a mechanism is then needed to send that to the destination mesh across the parallel domain decomposition. Parallel domain decomposition of meshes and grids are found throughout scientific computing software, so this is not a Firedrake specific problem.

2. An appropriate data structure is required which ties points to their locations in each mesh. This data structure needs to be able to cope with the parallel domain decomposition.

3. In order to differentiate the interpolation operation, the underlying operations need to be ones which we already know how to differentiate.

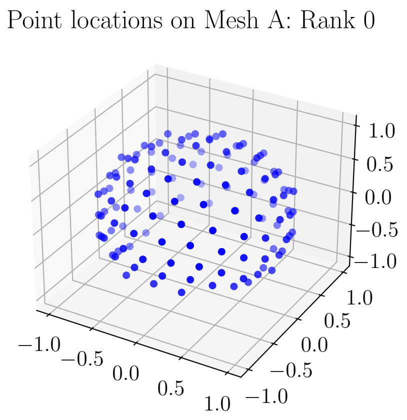
All these can be addressed by using a vertex-only mesh as the intermediary data structure.

We start with a simplified case of one point evaluation per basis function: the Lagrange polynomials (see Fig. 3.1 and Eq. 3.2.23). As discussed in chapter 3, each global basis function coefficient is now one point evaluation (we have point evaluation nodes).

The recipe for mesh-to-mesh dual evaluation interpolation

$$\mathcal{I}_V : U \to V \; \forall u \in U \tag{8.3.1}$$

[8]see https://doc.freefem.org/documentation/finite-element.html#a-fast-finite-element-interpolator



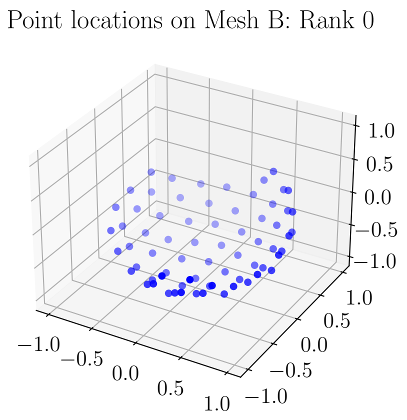


Figure 8.4: We have two different meshes of a sphere, A and B, with the same set of points on that sphere. After the meshes have been domain decomposed across MPI ranks, a given MPI rank will have a different subset of the points depending on which mesh is being considered.

where

$$U = \mathrm{FS}_{\mathrm{source}}(\Omega_{\mathrm{source}}) \tag{8.3.2}$$

$$V = \mathrm{FS}_{\mathrm{destination}}(\Omega_{\mathrm{destination}}) \tag{8.3.3}$$

is now the following:

1. Find the global point evaluation node coordinates of the *destination function space* V. This has the destination mesh $\Omega_{\mathrm{destination}}$ domain decomposition.

2. Create a vertex-only mesh Ω_v, *immersed in the source mesh* Ω_{source}, at those locations. In Firedrake, vertex-only meshes inherit their parent mesh parallel domain decomposition (here Ω_{source}). The point evaluation node coordinates have the parallel domain decomposition of the destination mesh $\Omega_{\mathrm{destination}}$ but are automatically redistributed to the source mesh's parallel domain decomposition by the voting algorithm. The input-ordering vertex-only mesh $\Omega_v^{\text{input-ordering}}$ maintains the ordering and MPI rank of the point evaluation node coordinates as supplied. Whilst the parent mesh of $\Omega_v^{\text{input-ordering}}$ is not $\Omega_{\mathrm{destination}}$, we nevertheless see that

$$\Omega_v^{\text{input-ordering}} \subseteq \Omega_{\mathrm{destination}}. \tag{8.3.4}$$

3. Point evaluate the expression or function on the source mesh U at the destination mesh point evaluation node coordinates Ω_v. This is the dual evaluation interpolation operation

$$\mathcal{I}_{\mathrm{P0DG}(\Omega_v)} : U \to \mathrm{P0DG}(\Omega_v). \tag{8.3.5}$$

4. Interpolate the point evaluations from vertex-only mesh Ω_v to a P0DG function space on

the input-ordering vertex-only mesh $\Omega_v^{\text{input-ordering}}$

$$\mathcal{I}_{\text{P0DG}(\Omega_v^{\text{input-ordering}})} : \text{P0DG}(\Omega_v) \to \text{P0DG}(\Omega_v^{\text{input-ordering}}) \tag{8.3.6}$$

Under the hood, this is a PETSc SF reduce operation[9] which moves the point evaluations from the parallel domain decomposition and ordering of Ω_v to $\Omega_v^{\text{input-ordering}}$ (a binary matrix multiplication - see Sect. 7.6.3).

5. The Firedrake function in P0DG($\Omega_v^{\text{input-ordering}}$) this yields has point evaluations from the source mesh function space at the destination mesh function space node coordinates, with the corresponding ordering and domain decomposition. These are the coefficients of the function in the destination function space V which can now be directly assigned.

$$C : \text{P0DG}(\Omega_v^{\text{input-ordering}}) \to V. \tag{8.3.7}$$

These steps are summarised in Fig. 8.5. More concisely, our mesh-to-mesh interpolation is the composition of three linear operations

$$\mathcal{I}_V(;u) = C(;\mathcal{I}_{\text{P0DG}(\Omega_v^{\text{input-ordering}})}(;\mathcal{I}_{\text{P0DG}(\Omega_v)}(;u))) \in V \quad \forall\, u \in U. \tag{8.3.8}$$

We wrap the recipe up in a new subclass of `firedrake.Interpolator` which we automatically create when interpolating across meshes. This recipe has the advantage of requiring very little special case code since we use Firedrake functionality which we have already implemented. This addresses the difficulty of differentiating this operation: we already know how to perform forward, adjoint and higher derivative modes of AD on linear interpolation operations and the linear assignment operation C (see sections 5.5.1 and 5.5.2).

Explicitly, the adjoint interpolation from cofunction to cofunction

$$\mathcal{I}_V^* : V^* \to U^*, \tag{8.3.9}$$

which we require for adjoint mode AD is the composition of the adjoint of each of the linear operations

$$\mathcal{I}_V^*(;v^*) = \mathcal{I}_{\text{P0DG}(\Omega_v)}^*(;\mathcal{I}_{\text{P0DG}(\Omega_v^{\text{input-ordering}})}^*(;C^*(;v^*)) \in U^* \quad \forall\, v^* \in V^*. \tag{8.3.10}$$

1. The first operation

$$C^* : V^* \to \text{P0DG}(\Omega_v^{\text{input-ordering}})^* \tag{8.3.11}$$

[9]see section 3.2 of Zhang et al. [97]

[10]This example uses immersed manifold sphere meshes: we therefore assume that $\Omega_v \subseteq \Omega_{\text{source}}$ up to the mesh discretisation error. This is specified for each mesh with a cell-size-relative `tolerance` parameter. This example was chosen because it clearly illustrates the node locations of V as identified in both Ω_{source} and $\Omega_{\text{destination}}$.

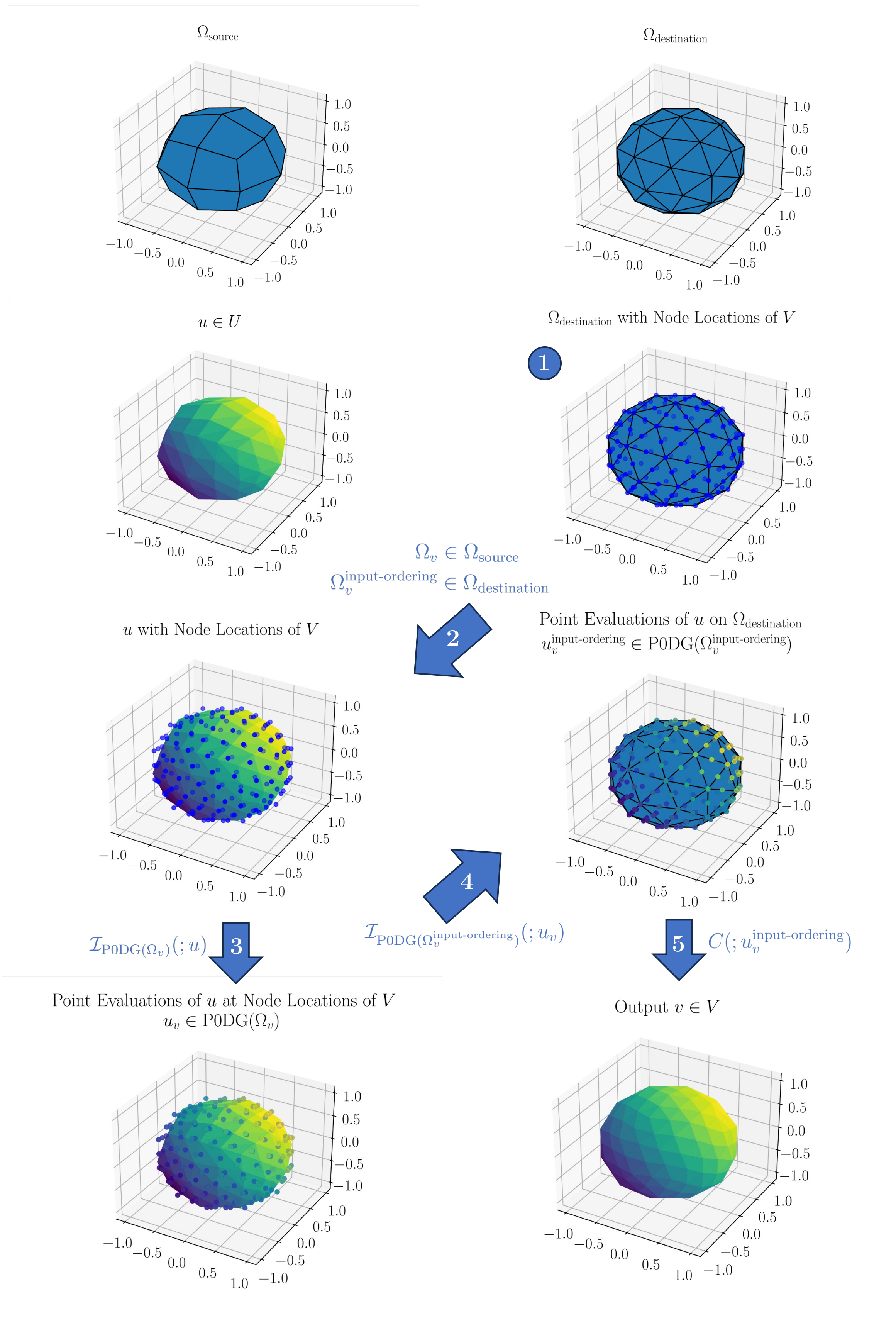

Ω_{source}
$\Omega_{\text{destination}}$
$u \in U$
$\Omega_{\text{destination}}$ with Node Locations of V
1
$\Omega_v \in \Omega_{\text{source}}$
$\Omega_v^{\text{input-ordering}} \in \Omega_{\text{destination}}$
2
u with Node Locations of V
Point Evaluations of u on $\Omega_{\text{destination}}$
$u_v^{\text{input-ordering}} \in \text{P0DG}(\Omega_v^{\text{input-ordering}})$
4
$\mathcal{I}_{\text{P0DG}(\Omega_v^{\text{input-ordering}})}(;u_v)$
3
$\mathcal{I}_{\text{P0DG}(\Omega_v)}(;u)$
5
$C(;u_v^{\text{input-ordering}})$
Point Evaluations of u at Node Locations of V
$u_v \in \text{P0DG}(\Omega_v)$
Output $v \in V$

Figure 8.5: Summary of the steps needed for cross-mesh interpolation where $U = \text{FS}_{\text{source}}(\Omega_{\text{source}})$ and $V = \text{FS}_{\text{destination}}(\Omega_{\text{destination}})$.[10]

is the assignment of the basis cofunction coefficients of V^* to an identically discretised cofunction in $\mathrm{P0DG}(\Omega_v^{\text{input-ordering}})^*$. We again use the input-ordering vertex-only mesh $\Omega_v^{\text{input-ordering}}$, exactly as it was set up for interpolating primal functions (i.e. it has point evaluation node coordinates in the order and rank domain decomposition of $\Omega_{\text{destination}}$).

2. The next operation

$$\mathcal{I}^*_{\mathrm{P0DG}(\Omega_v^{\text{input-ordering}})} : \mathrm{P0DG}(\Omega_v^{\text{input-ordering}})^* \to \mathrm{P0DG}(\Omega_v)^* \tag{8.3.12}$$

is the PETSc SF broadcast operation[11] from the vertices of $\Omega_v^{\text{input-ordering}}$ to the equivalent point evaluation node coordinate locations $\Omega_v \subseteq \Omega_{\text{source}}$.

3. Lastly,

$$\mathcal{I}^*_{\mathrm{P0DG}(\Omega_v)} : \mathrm{P0DG}(\Omega_v)^* \to U^* \tag{8.3.13}$$

is the adjoint to the point evaluation interpolation operation. At present, this involves Firedrake assembling the primal interpolation operator as a matrix and left multiplying it by the cofunction coefficients.

Returning to the original recipe: Where a target mesh does not fully overlap with the source mesh, we default to raising an exception, but allow this to be overridden with a keyword argument when creating the cross-mesh `Interpolator` object. This allows for applications such as simulation coupling. The intermediary vertex-only Ω_v mesh drops any points it can't locate in the source mesh but keeps those points in the input-ordering vertex-only mesh $\Omega_v^{\text{input-ordering}}$. We allow a default to be specified which we seed an intermediary `Function` on $\Omega_v^{\text{input-ordering}}$ prior to interpolation onto it with $\mathcal{I}_{\mathrm{P0DG}(\Omega_v)}$. The interpolation itself then doesn't assign a value to the input-ordering vertex-only mesh, and the node value assignment operation C therefore uses the default value.

For cases where we already have a `Function` which we want to interpolate onto, we similarly supply NaNs[12] to the intermediary `Function` on the $\Omega_v^{\text{input-ordering}}$ prior to interpolation. These are then identified and removed such that the assignment operation C only writes to the nodes it has values for. Regarding adjoint operations in this case: rather than explicitly filtering out values to skip over (which would be strictly correct for C^* as we've defined it here) we assign all the cofunction basis-function values on $\Omega_{\text{destination}}$ to the cofunction on $\Omega_v^{\text{input-ordering}}$, then let the adjoint interpolation $\mathcal{I}^*_{\mathrm{P0DG}(\Omega_v)}$ from $\Omega_v^{\text{input-ordering}}$ to Ω_v remove the values for us.

[11]as before, see section 3.2 of Zhang et al. [97]

[12]Not a Number - a special value that is part of the IEEE specfification for floating point numbers

8.4 Discussion and Demonstrations

The new interpolation operations allow us to do everything we wanted in the motivation. Fig. 8.5 was created using Firedrake code which runs in parallel and has been integrated with `master` Firedrake: see Listing 13. Similarly, we can evaluate the line integral in Fig. 8.3 by interpolating onto a function space on the line and assembling the corresponding expression (Listing 14).

```python
from firedrake import *

# f = x + y + z on source mesh
src_mesh = UnitCubedSphereMesh(1)  # quadrilateral cells
U = FunctionSpace(src_mesh, "S", 2)  # "S" needs quadrilateral cells
x, y, z = SpatialCoordinate(src_mesh)
u = Function(U).interpolate(x + y + z)

# interpolate across mesh and function space
dest_mesh = UnitIcosahedralSphereMesh(1)  # triangle cells
V = FunctionSpace(dest_mesh, "CG", 2)
v = Function(V).interpolate(u)
```

Listing 13: Interpolating from one mesh and function space to another. Fig. 8.5 contains plots of the meshes (`src_mesh` is Ω_{source} and `dest_mesh` is $\Omega_{\text{destination}}$), and Firedrake `Functions` u and v.

```python
from firedrake import *
import numpy as np

m = UnitSquareMesh(2, 2)
V = FunctionSpace(m, "CG", 2)
x, y = SpatialCoordinate(m)
f = Function(V).interpolate(x * y)  # Exactly representable in CG2

cells = np.asarray([[0, 1]])  # Only 1 cell
vertex_coords = np.asarray([[0.0, 0.0], [1.0, 1.0]])
tdim = 1  # topological dimension
gdim = 2  # geometric dimension
dmplex = mesh.plex_from_cell_list(tdim, cells, vertex_coords, m.comm)
line = mesh.Mesh(dmplex, dim=gdim)

V_line = FunctionSpace(line, "CG", 2)
f_line = interpolate(f, V_line)
line_integral = assemble(f_line * dx)
assert np.isclose(line_integral, np.sqrt(2) / 3)
```

Listing 14: Evaluating a line integral in Firedrake using mesh-to-mesh interpolation. See also Fig. 8.3

Since these operations are differentiable with pyadjoint, we can create functionals to minimise which involve lines, surfaces and volumes within a domain. For example, one might have an ocean model, some pre-existing values of tidal fluxes through a surface (perhaps from a diagnostic on another model), and want to optimise one's own model to fit those values. The code for forming the necessary misfit functional would involve creating the corresponding surface for the flux immersed in the model domain, interpolating onto the surface, then assembling a UFL expression for the flux on the surface. The general approach is shown in Listing 15. When optimising for multiple diagnostics these misfits can be summed together alongside an appropriate regularisation.

```python
from firedrake import *

...
point_mesh = VertexOnlyMesh(mesh, point_location)
V_point = FunctionSpace(point_mesh, "DG", 0)
f_point = interpolate(f, V_point)
point_eval = assemble(f_point * dx)
J_misfit_point_eval = (point_eval - data_point_eval)**2
...
# line_mesh = ...
# V_line = FunctionSpace(line_mesh, ...)
f_line = interpolate(f, V_line)
line_integral = assemble(f_line * dx)
J_misfit_line_integral = (line_integral - data_line_integral)**2
...
# surface_mesh = ...
# V_surface = FunctionSpace(surface_mesh, ...)
f_surface = interpolate(f, V_surface)
surface_integral = assemble(f_surface * dx)
J_misfit_surface_integral = (surface_integral - data_surface_integral)**2
...
# V_vector_suface = VectorFunctionSpace(surface_mesh, ...)
f_vector_surface = interpolate(f_vector, V_vec_surface)
flux = assemble(inner(f_vector_surface, CellNormal(surface_mesh)) * dx)
J_misfit_flux = (flux - data_flux)**2
...
# volume_mesh = ...
# V_volume = FunctionSpace(volume_mesh, ...)
f_volume = interpolate(f, V_volume)
volume_integral = assemble(f_volume * dx)
J_misfit_volume_integral = (volume_integral - data_volume_integral)**2
```

Listing 15: How we could assemble misfit functionals for PDE constrained optimisation given some PDE solution f and a value of an equivalent diagnostic. In the most simple case (the top of the code snippet), this is optimising to match with a single point evaluation which is done using a vertex-only mesh as shown.

Where we have data that are only available in point, line, surface or volume data, for example when coupling to a model which is supplying data from outside of Firedrake, we can directly create functionals to minimise using this data much as we do for point data: see listing 16. A mesh of 1 or more lines, surfaces or volumes which represent the data would be created, an appropriate function space used and the PDE solution interpolated onto it to form the functional. These misfits could be combined together, alongside point data misfit functionals (see other chapters) and an appropriate regularisation to create an overall functional to minimise.

```python
from firedrake import *

...
points_mesh = VertexOnlyMesh(mesh, point_locations)
V_points = FunctionSpace(points_mesh, "DG", 0)
f_points = interpolate(f, V_points)
# data_points is a function in V_points, alpha is a scalar
J_misfit_points = assemble(alpha * (f_points - data_points)**2 * dx)
...
# lines_mesh = ...
# V_lines = ...
f_lines = interpolate(f, V_lines)
# data_lines is a function in V_lines, beta is a scalar
J_misfit_lines = assemble(beta * (f_lines - data_lines)**2 * dx)
...
# surfaces_mesh = ...
# V_surfaces = ...
f_surfaces = interpolate(f, V_surfaces)
# data_surfaces is a function in V_surfaces, gamma is a scalar
J_misfit_surfaces = assemble(gamma * (f_lines - data_surfaces)**2 * dx)
...
# volumes_mesh = ...
# V_volumes = ...
f_volumes = interpolate(f, V_volumes)
# data_volumes is a function in V_volumes, delta is a scalar
J_misfit_volumes = assemble(delta * (f_volumes - data_volumes)**2 * dx)

J_misfit = (
    J_mistfit_points
    + J_mistfit_lines
    + J_misfit_surfaces
    + J_misfit_volumes
)
```

Listing 16: How we could assemble misfit functionals for point, line, surface and volume data for PDE contrained optimisation given some PDE solution f. Here we use the L^2 norm for each misfit functional, but that need not be the case.

The in-built integration with pyadjoint via `firedrake.adjoint` (see Sect. 5.5) is tested with a Taylor remainder convergence test. If we create a functional $J(m)$, where m is the control parameter we will take the derivative with respect to, then we expect from Taylor's theorem[13] that

$$\|J(m + h\delta m) - J(m)\| \to 0 \text{ at } \mathcal{O}(h) \tag{8.4.1}$$

for some small scalar h and some small perturbation of the control δm. The Gateaux derivative of J with respect to m in the m' direction is $dJ_m(m; m')$. Applying Taylor's theorem again to find the next term in the expansion, we now expect that

$$\|J(m + h\delta m) - J(m) - hdJ_m(m; \delta m)\| \to 0 \text{ at } \mathcal{O}(h^2). \tag{8.4.2}$$

If the Jacobian $dJ_m(m; \bullet)$ `firedrake.adjoint` gives us is correct, we expect to observe this rate of convergence. Such tests already exist for `firedrake.adjoint`'s annotation of interpolation, we extend these to cross-mesh interpolation with the functional

$$J(m) = \int_{\Omega_{\text{destination}}} \mathcal{I}_{\text{P1CG}(\Omega_{\text{destination}})}(u^2 + mu)^2 \; dx \tag{8.4.3}$$

where Ω_{source} and $\Omega_{\text{destination}}$ are unit-square meshes with differing numbers of square and quadrilateral cells respectively[14],

$$\mathcal{I}_{\text{P1CG}(\Omega_{\text{destination}})} : \text{P2CG}(\Omega_{\text{source}}) \to \text{P1CG}(\Omega_{\text{destination}}) \tag{8.4.4}$$

is the cross-mesh interpolator from 2nd order continuous polynomials on the source mesh to 1st order continuous polynomials of the destination mesh,

$$u \in \text{P2CG}(\Omega_{\text{source}}) \tag{8.4.5}$$

is a ramp function which takes on the values of the x-axis of the domain and

$$m \in \mathbb{R} \tag{8.4.6}$$

such that

$$mu \in \text{P2CG}(\Omega_{\text{source}}). \tag{8.4.7}$$

A similar test for convergence of the Hessian action, which relies on the next term in the Taylor expansion, has also been added. For more on Taylor tests, see section 1.4.6 of Schwedes et al. [29].

[13] expanding $J(m + h\delta m)$ about m

[14] to be precise, they are the bottom left and top right meshes in Fig. 8.2, respectively

8.4.1 Model Coupling

Models can now be coupled together exactly as described in Sect. 8.1. As a proof of concept, we use two square meshes (Fig. 8.6) and couple a Helmholtz problem to a Poisson problem along the $y = 1$ boundary. The Helmholtz problem on the top mesh Ω_{top} with boundary $\partial\Omega_{\text{top}}$ is taken from the Firedrake manual[5]:

$$-\nabla^2 u + u = f \tag{8.4.8}$$

$$\nabla u \cdot \boldsymbol{n} = 0 \text{ on } \partial\Omega_{\text{top}} \tag{8.4.9}$$

with

$$f(x, y) = (1 + 8\pi^2)\cos 2\pi x \cos 2\pi y. \tag{8.4.10}$$

The variational form of the Helmholtz problem is to find $u \in V_{\text{top}}$ such that

$$\int_{\Omega_{\text{top}}} \nabla u \cdot \nabla v + uv \; dx - \int_{\Omega_{\text{top}}} fv \; dx - \int_{d\Omega_{\text{top}}} v\nabla u \cdot \boldsymbol{n} \; ds = 0 \quad \forall v \in V_{\text{top}} \tag{8.4.11}$$

The $\int_{d\Omega_{\text{top}}} \nabla u \cdot \boldsymbol{n} \; ds$ term disappears where we set $\nabla u \cdot \boldsymbol{n} = 0$.

The Poisson problem on the bottom mesh Ω_{bottom} with boundary $\partial\Omega_{\text{bottom}}$ is the same as that used in sections 1.3 and 4.6:

$$-\nabla^2 u = f \tag{8.4.12}$$

$$u = 0 \text{ on } \partial\Omega_{\text{bottom}} \tag{8.4.13}$$

with $f(x, y)$ being a randomly generated forcing term. The variational form of the Poisson problem is to find $u \in V_{\text{bottom}}$ such that

$$\int_{\Omega_{\text{bottom}}} \nabla u \cdot \nabla v \; dx - \int_{\Omega_{\text{bottom}}} fv \; dx - \int_{d\Omega_{\text{bottom}}} v\nabla u \cdot \boldsymbol{n} \; ds = 0 \quad \forall v \in V_{\text{bottom}}. \tag{8.4.14}$$

The $\int_{d\Omega_{\text{bottom}}} \nabla u \cdot \boldsymbol{n} \; ds$ term disappears where we specify a value on the boundary.

We initially use strong (Dirichlet) boundary conditions to couple Helmholtz to Poisson: this changes Eq. 8.4.13 to be

$$u = u_{\text{interpolated from } V_{\text{top}}} \text{ on } \partial\Omega_{\text{bottom}} \tag{8.4.15}$$

where $u_{\text{interpolated from } V_{\text{top}}}$ is solution to the Helmholtz problem interpolated into V_{bottom} with initial values set to zero. The effect of applying this Dirichlet boundary condition and the solution Firedrake gives can be seen in Fig. 8.7. The code used is in Listing 17.

We achieve two way coupling iteratively as described in Sect. 8.1. The weak boundary condition we apply is a penalty term added to the right hand side of each PDE:

$$w \int_{\Gamma_{\text{interface}}} (u_{\text{interpolated}} - u)v \; d\Gamma_{\text{interface}} \tag{8.4.16}$$

where $u_{\text{interpolated}}$ is the solution to the other problem at each iteration (on the first instance this is a zero function) and v is the appropriate test function. This gets smaller as the solution to each PDE at the interface approaches the interpolated interface solution from the other mesh. The overall effect of the boundary condition is controlled by a positive scalar weighting w. When the solutions to both PDEs stop changing we stop iterating. We now only apply the strong (Dirichlet) boundary conditions to the Poisson problem (Eq. 8.4.13) away from the interface: we therefore have to include the surface term $\int_{d\Omega_{\text{bottom}}} \nabla u \cdot \boldsymbol{n}\, ds$ along the interface in the variational form since we leave the value at the interface unspecified. Similarly we have to include the surface term along the interface in the Helmholtz variational form $\int_{d\Omega_{\text{top}}} \nabla u \cdot \boldsymbol{n}\, ds$ since we no longer fix $\nabla u \cdot \boldsymbol{n} = 0$ along the interface.

The code for the two can be found in the appendix to this chapter (Sect. A.4, Listing 19) with the results shown in Fig. 8.8. Whilst the solutions do not perfectly match across the domains, the Poisson problem's solution clearly now effects the solution to the Helmholtz problem, pulling $u(x, y)$ down.

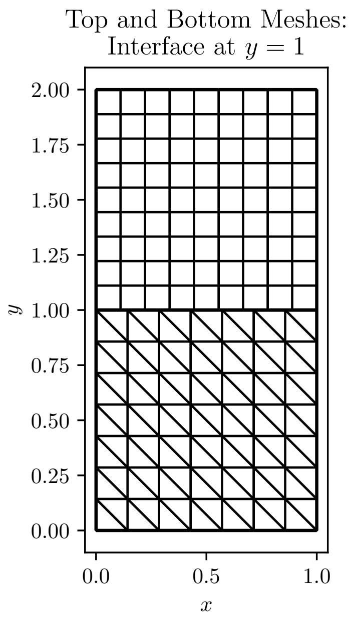
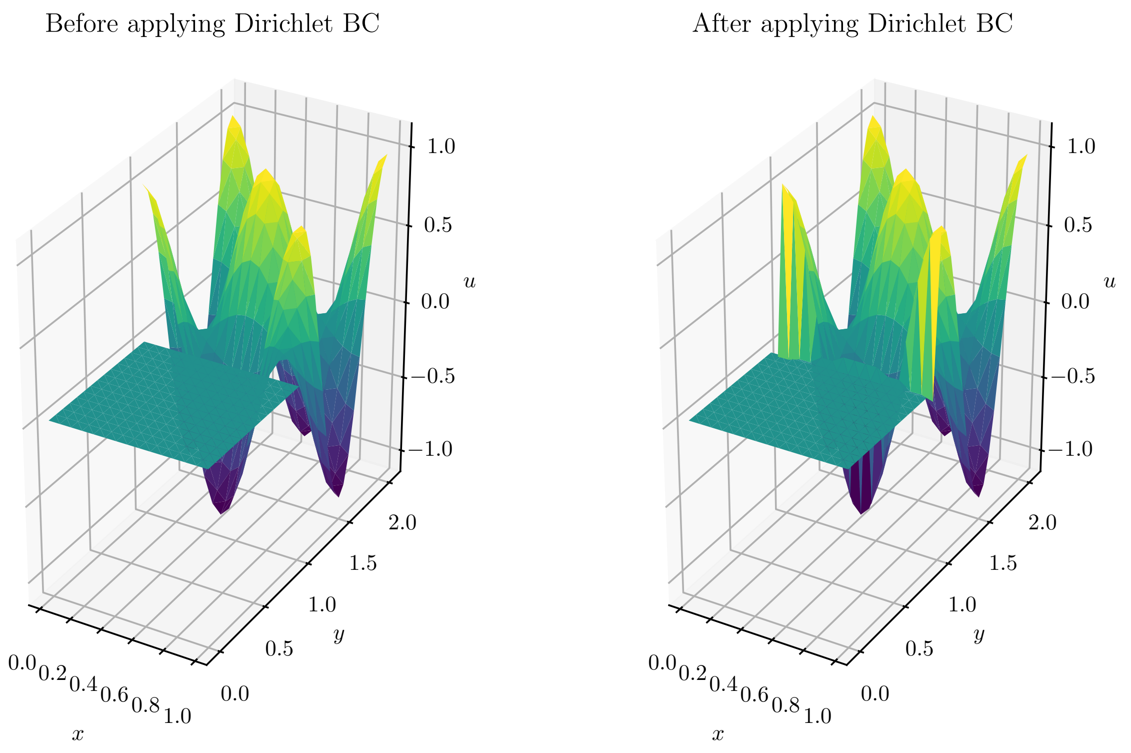


Figure 8.6: The meshes we solve our coupled problems on. Helmholtz is solved on the top mesh and Poisson on the bottom.

8.4.2 Generic External Field Point-Evaluation Data Input

Field data that one might wish to import from an external source are typically specified as a set of point evaluations of the underlying field. The BedMachine map of Antarctic ice thickness [61] and the MEaSUREs InSAR phase-based velocity map [62] mentioned in chapter 6 are both, in principal, point evaluation measurements of the underlying thickness and velocity fields, in each case aligned to a grid.



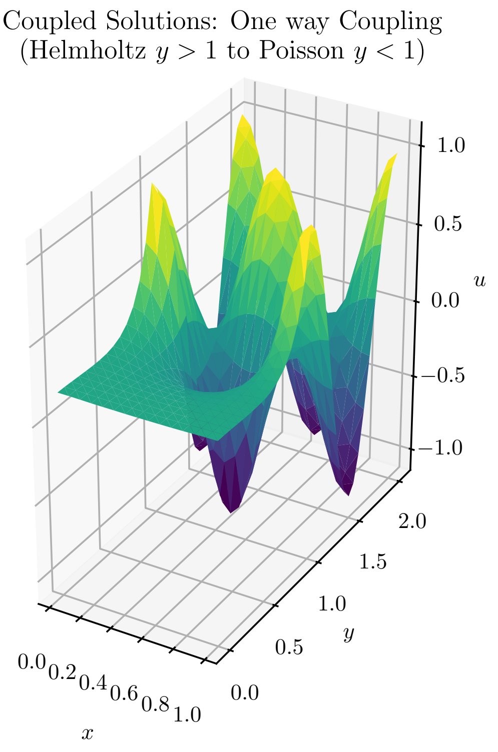
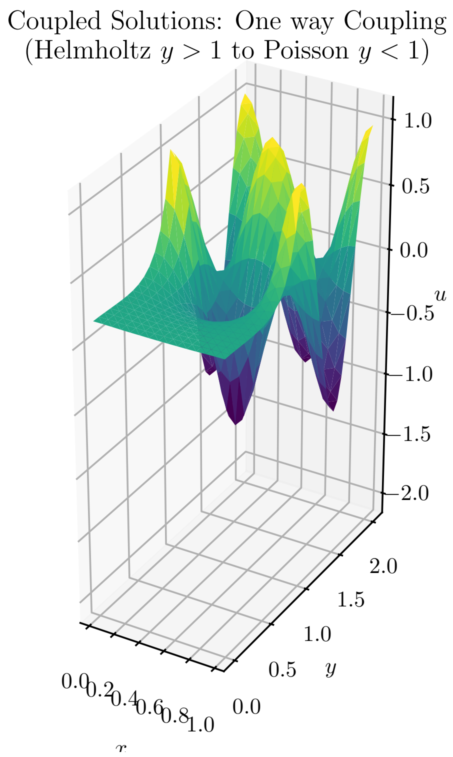


Figure 8.7: One way coupling in action, from the code in Listing 17. We solve a Helmholtz problem on the top mesh ($y > 1$), then interpolate the solution onto the bottom mesh ($y < 1$), giving values on the bottom mesh along the interface. The top two plots show the effect of applying the Dirichlet (strong) boundary conditions using the interpolated `Function`. We then solve the Poisson problem on the bottom mesh.



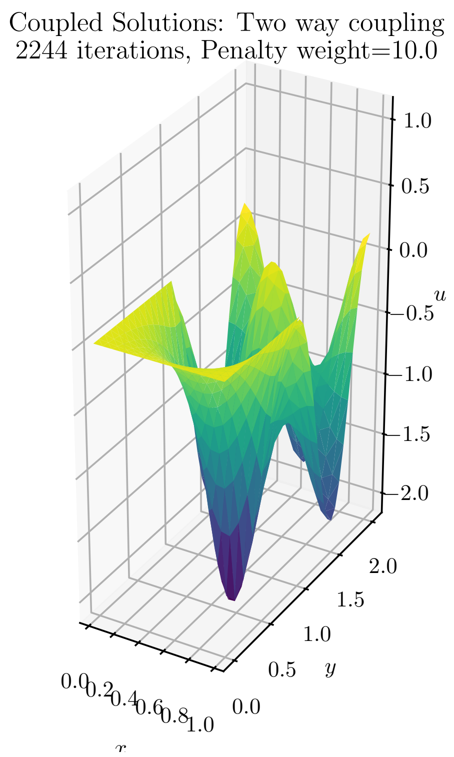


Figure 8.8: One way coupling (left) versus two way coupling (right) of a Helmholtz problem and a Poisson problem. The two way coupling uses a weakly enforced boundary condition 8.4.16 enforced as a penalty term in the variational form. The meshes used are shown in Fig. 8.6 and the code is in Listing 19 in the appendix to this chapter (Sect. A.4).

```python
from firedrake import *

m_top = UnitSquareMesh(9, 9, quadrilateral=True)
m_top.coordinates.dat.data_wo[:, 1] += 1.0
m_bot = UnitSquareMesh(7, 7)

V_top = FunctionSpace(m_top, "CG", 2)
V_bot = FunctionSpace(m_bot, "CG", 3)

# Helmholtz problem on the top mesh
u_top = Function(V_top)
v = TestFunction(V_top)
f_top = Function(V_top)
x, y = SpatialCoordinate(m_top)
f_top.interpolate((1 + 8 * pi * pi) * cos(x * pi * 2) * cos(y * pi * 2))
F = (inner(grad(u_top), grad(v)) + inner(u_top, v)) * dx - inner(f_top, v) * dx
solve(F == 0, u_top)

# Couple Helmholtz to Poisson with interpolate
u_top_in_V_bot = Function(V_bot).interpolate(u_top, allow_missing_dofs=True)
bc = DirichletBC(V_bot, u_top_in_V_bot, 4)

# Poisson problem on the bottom mesh, using the BCs from the top mesh
u_bot = Function(V_bot)
v = TestFunction(V_bot)
# Random forcing Function with values in [1, 2].
f_bot = RandomGenerator(PCG64(seed=0)).beta(V_bot, 1.0, 2.0)
F = (inner(grad(u_bot), grad(v)) - f_bot * v) * dx
solve(F == 0, u_bot, bc)
```

Listing 17: One-way coupling of a Helmholtz problem to a Poisson problem. The Helmholtz problem is adapted from the 'Simple Helmholtz equation' demo in the Firedrake manual[5], whilst the Poisson problem is adapted from Listing 2 (note again that the forcing term is mesh dependent so this should only be used for demonstration). The meshes are shown in Fig. 8.6 and solutions in Fig. 8.7.

The input-ordering vertex-only mesh already gives us a parallel safe way to input point data into a vertex-only mesh (see Sect. 7.6). More generically, we can use these, alongside mesh-to-mesh interpolation, to input field point-evaluation data onto any mesh. These data can be distributed across multiple MPI ranks, which would prove very useful for automating diagnostics calculations on such data. For more, see Sect. 10.2.5.

This works as follows:

1. We have point evaluation data with values at various locations distributed across various MPI ranks.
2. We create a Firedrake mesh Ω_{data} which has vertices at the data coordinates. This mesh does not need to respect the parallel decomposition of the data.

3. We create a vertex-only mesh Ω_v immersed in Ω_{data} with point locations at the data coordinates: these are redistributed to have the parallel decomposition of the mesh Ω_{data}.

4. Our input-ordering vertex-only mesh $\Omega_v^{\text{input-ordering}}$ retains the original decomposition: we use this to assign data values to a P0DG function space on Ω_v (see Sect. 7.6). Our data are now the point data function $f_v \in \text{P0DG}(\Omega_v)$.

5. Our vertex-only mesh Ω_v has the same number of points as vertices of the mesh Ω_{data}. If we now create a space of linear continuous polynomials on the mesh $\text{P1CG}(\Omega_{\text{data}})$ we have point evaluation nodes at each of our point locations. We therefore need to find the dual evaluation interpolation operation which takes us from $\text{P0DG}(\Omega_v)$ to $\text{P1CG}(\Omega_{\text{data}})$. The data in the point data function $f_v \in \text{P0DG}(\Omega_v)$ are not necessarily in the same order as data in functions in $\text{P1CG}(\Omega_{\text{data}})$: we need to find the correct permutation from the former to the latter. Going the other way, the parallel safe permutation matrix from $\text{P1CG}(\Omega_{\text{data}})$ to $\text{P0DG}(\Omega_v)$ is the point-evaluation interpolation matrix, itself a permutation matrix. We want the permutation in the other direction - i.e. it's inverse. Fortunately permutations are orthogonal matrices, so the transpose is the inverse. The transpose of the point-evaluation interpolation matrix performs the correct permutation. We therefore apply this interpolation to get $f_{\text{data}} \in \text{P1CG}(\Omega_{\text{data}})$.

6. We now have a finite element function on Ω_{data} in Firedrake which is a linear interpolation between the originally supplied data points representing the field. We can interpolate this onto other meshes as we see fit, perhaps to calculate some derived quantity such as a line integral.

A simple example of this in action is shown in the appendix to this chapter (Sect. A.4, Listing 20) and a plot of the data and function produced is in Fig. 8.9.

To be clear, this produces a linear interpolation as Fig. 8.9 shows: the data will not be smooth between vertices - we have not solved any kind of optimisation problem, which might impose a smoothing constraint, to find it. This, however, does give us a parallel safe way of bringing this kind of data into Firedrake to work with as we wish. If one does want to solve an optimisation problem with such data, they should treat it as the point data it is. A vertex-only mesh should be used directly, with data input using the input-ordering vertex-only mesh for parallel safety, and formulate an appropriate misfit functional (see Chapter 6).

8.4.3 Line, Plane and Volume Sources and Sinks

In Sect. 4.6.1, it was demonstrated how the adjoint to the dual evaluation interpolation operation which performs point evaluation can be used to create point source and sink terms. If $v \in V$ is a test function on a parent mesh Ω and $v_v \in \text{P0DG}(\Omega_v)$ is the corresponding test

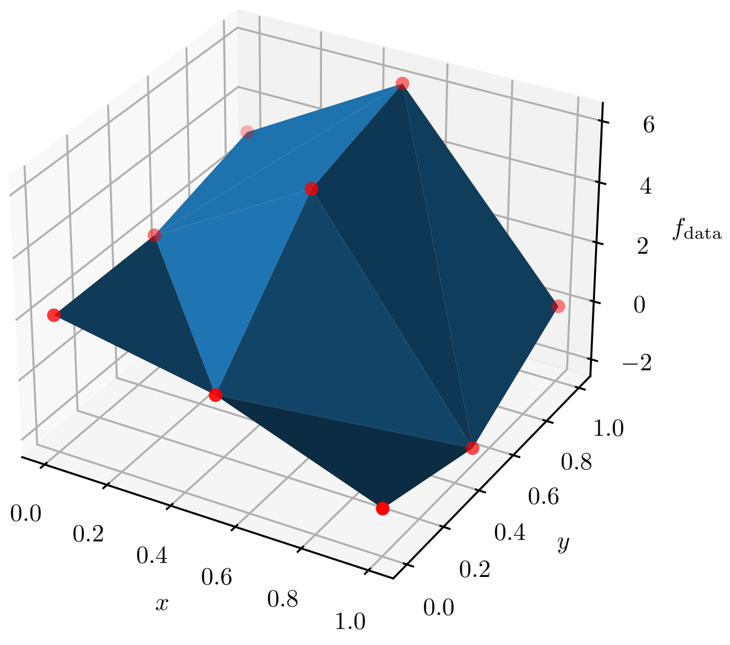


Figure 8.9: The function produced after inputting 9 data points using the process outlined in 8.4.2 using the code in Listing 20 in the appendix to this chapter (Sect. A.4). The data themselves are shown in red. The full function is a linear interpolation between each of the input field data points - i.e. we have triangular planes linking the points.

function on an immersed vertex-only mesh $\Omega_v \subseteq \Omega$ with N vertices at X_i, then

$$\sum_{i=0}^{N-1} \int_\Omega f(x)v(x)\delta(x - X_i)\ dx = \mathcal{I}^*_{\mathrm{P0DG}(\Omega_v)}\left(;\int_{\Omega_v} f_v v_v\ dx\right) \in V^* \tag{8.4.17}$$

where $f_v(X_i) = f(X_i)$ is a function in $\mathrm{P0DG}(\Omega_v)$ for all N vertices $\{X_i\}_{i=0}^{N-1}$ of Ω_v. We can simplify this by leaving out f, to just the equivalent ways of expressing point forcing at the point X_i:

$$\sum_{i=0}^{N-1} \int_\Omega v(x)\delta(x - X_i)\ dx = \mathcal{I}^*_{\mathrm{P0DG}(\Omega_v)}\left(;\int_{\Omega_v} v_v\ dx\right) \in V^*. \tag{8.4.18}$$

A reasonable question to ask is whether we can do the same for line, surface and volume sources and sinks via the adjoint cross-mesh dual evaluation interpolation operation $\mathcal{I}^*_V$. We start with some cofunction

$$v^* = \int_{\Omega_{\text{destination}}} f_{\text{destination}} v_{\text{destination}}\ dx \in V^*, \tag{8.4.19}$$

where $v_{\text{destination}} \in V$ is a test function on the target mesh $\Omega_{\text{destination}}$, and $f_{\text{destination}}$ is some function on $\Omega_{\text{destination}}$. Operating on this with

$$\mathcal{I}^*_V(;v^*) \in U^* \tag{8.4.20}$$

to get a cofunction in U^*, should allow, in some way, the introduction of a source term to a linear variational problem with a left hand side primarily defined over the source mesh.

The question to answer is exactly *how* the two cofunctions are linked, since we want to introduce the test function on the target mesh, so that we have a source term there. To answer this, we need to look at the composition of the adjoint cross-mesh dual evaluation interpolation operation $\mathcal{I}^*_V : V^* \to U^*$

$$\mathcal{I}^*_V(;v^*) = \mathcal{I}^*_{\mathrm{P0DG}(\Omega_v)}(;\mathcal{I}^*_{\mathrm{P0DG}(\Omega_v^{\text{input-ordering}})}(;C^*(;v^*)) \in U^* \quad \forall\, v^* \in V^* \tag{8.4.21}$$

(Eq. 8.3.10). The inner operations

$$\mathcal{I}^*_{\mathrm{P0DG}(\Omega_v^{\text{input-ordering}})} \circ C^* : V^* \to \mathrm{P0DG}(\Omega_v)^* \tag{8.4.22}$$

take the basis cofunction coefficients of some $v^*_{\text{destination}} \in V^*$ and assign them to $\mathrm{P0DG}(\Omega_v)^*$ where Ω_v has vertices X_i at the point evaluation node locations of the target space V. The outer operator, $\mathcal{I}^*_{\mathrm{P0DG}(\Omega_v)}$ is the same as that used in Eq. 8.4.18 to introduce point source terms, though in this case the points are at the point evaluation node locations. So if

$$v^* = \int_{\Omega_{\text{destination}}} v_{\text{destination}}(x)\ dx \in V^* \tag{8.4.23}$$

then, by analogy with Eq. 8.4.18, $\mathcal{I}_V^*(;v^*)$ is equivalent to

$$\mathcal{I}_V^*\left(;\int_{\Omega_{\text{destination}}} v_{\text{destination}}(x)\ dx\right) = \sum_{i=0}^{N-1}\int_{\Omega_{\text{source}}} v_{\text{source}}(x)\delta(x-X_i)\ dx \tag{8.4.24}$$

where X_i are the N locations of the point evaluation nodes of the target space V.

What about where we have some $f_{\text{destination}}$ in the cofunction v^*? If we introduce some unknown f_{source} on the source mesh, then, from Eq. 8.4.17 we have

$$\sum_{i=0}^{N-1}\int_{\Omega_{\text{source}}} f_{\text{source}} v_{\text{source}}(x)\delta(x-X_i)\ dx = \mathcal{I}_V^*\left(;\int_{\Omega_{\text{destination}}} f_{\text{destination}} v_{\text{destination}}(x)\ dx\right) \tag{8.4.25}$$

$$= \mathcal{I}_V^*\left(;\int_{\Omega_{\text{destination}}} \mathcal{I}_V(;f_{\text{source}}) v_{\text{destination}}(x)\ dx\right) \tag{8.4.26}$$

i.e.

$$f_{\text{source}} = \mathcal{I}_V^{-1}(;f_{\text{destination}}) \tag{8.4.27}$$

which may or may not exist depending on the function spaces involved. We can, however, always introduce a source term by starting from a function f_{source} on the source mesh: the source term will be that function evaluated at the point evaluation node locations of the target space V.

So we see that we can introduce source terms using the adjoint cross-mesh dual evaluation interpolation operation $\mathcal{I}_V^*$ but it will always be a sum of point sources. Depending on the target application, as long as one has enough point evaluation nodes, this may be sufficient.

As an example, the Poisson point source example in Sect. 4.6.1 is adapted to use a line mesh from $(0,0)$ to $(1,1)$ as its source terms. The linear variational problem is 'find $u \in U$ such that $a(;u,v) = L(;v)$ for all $v \in U$' where

$$a(;u,v) = \int_\Omega \nabla u \cdot \nabla v - \int_{d\Omega} v\nabla u \cdot \boldsymbol{n}\ ds, \tag{8.4.28}$$

$$L(;v(x)) = \sum_{i=0}^{\dim(V)-1}\int_\Omega f(x)v(x)\delta(x-X_i)\ dx \text{ and} \tag{8.4.29}$$

$$f(x) = x \times y, \tag{8.4.30}$$

where X_i are the point evaluation node locations of the target function space V, on the line mesh, of which there is one per dimension of V. Boundary conditions are $u = 0$ on $x = 0$ and $y = 0$, and $\nabla u \cdot \boldsymbol{n} = 0$ on the $x = 1$ and $y = 1$ boundaries, so the surface term is always zero and need not be calculated. The code for this is in the appendix to this chapter (Listing 21 in Sect. A.4) and the results shown in Fig. 8.10. The more point evaluation nodes there are in the target function space, the smoother the solution is.

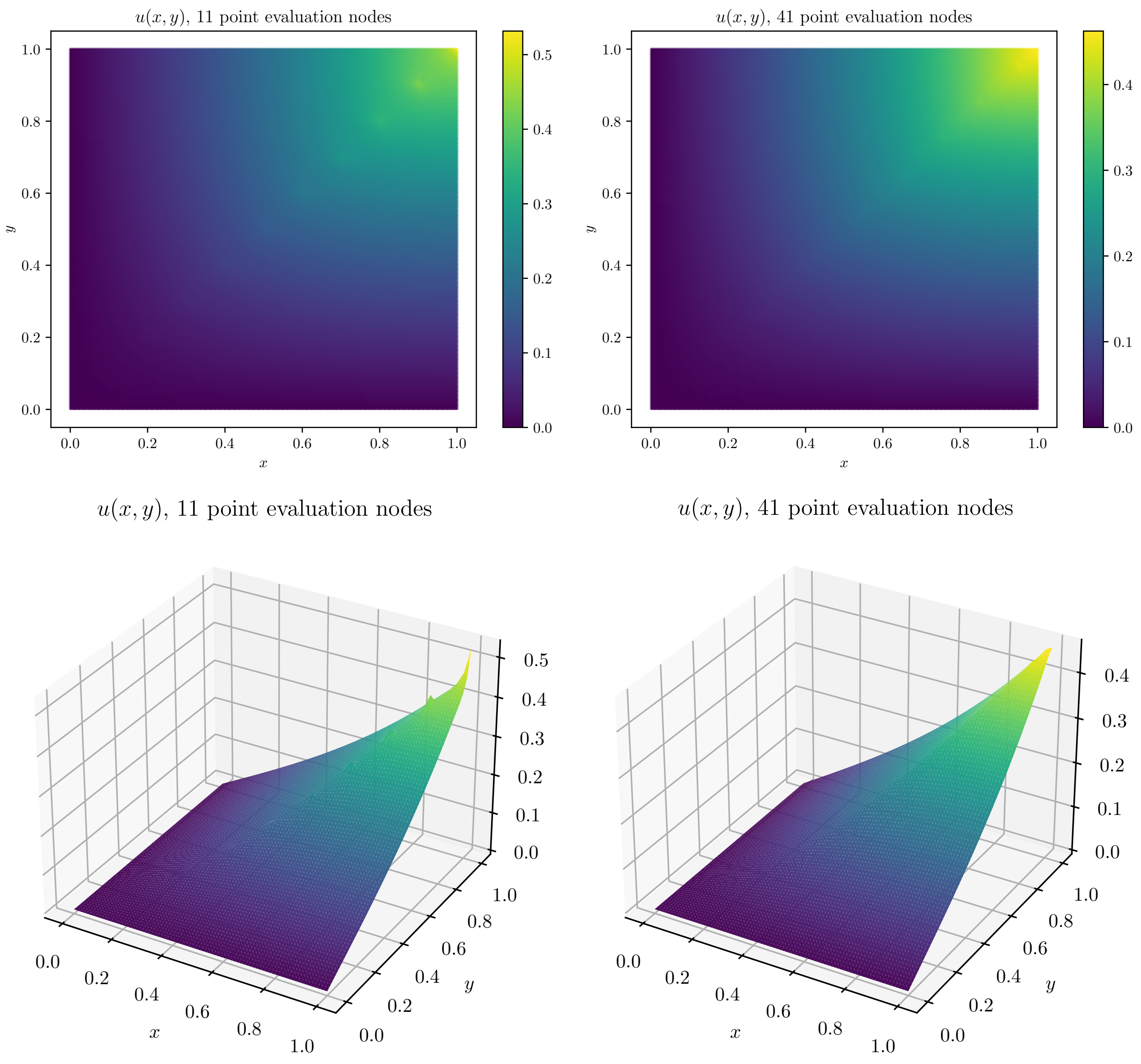


Figure 8.10: The solution to the Poisson problem with point-forcing source terms on a line mesh from $(0, 0)$ to $(1, 1)$, making use of the adjoint cross-mesh dual evaluation interpolation operation $\mathcal{I}_V^*$, as described in Sect. 8.4.3. The more point evaluation nodes we have, the smoother the solution is. The code for this is in the appendix to this chapter (Sect. A.4, Listing 21).

8.5 Future Work

The current implementation has limitations: foremost is only being able to interpolate into spaces with point evaluation nodes. This could be addressed in a number of ways: to continue using vertex-only meshes as the intermediate data-structure, we could specify the points to sum for each node (Eq. 3.7.2) in global coordinates with weights attached, perhaps as a DMSwarm 'field'. We would then need to introduce some routine that summed over the necessary points to build each of the dual basis functionals (nodes). Other limitations are those of vertex-only meshes as implemented in Firedrake: we cannot yet immerse them in high-order meshes[15] or extruded meshes with variable layers. This means we cannot cross-mesh interpolate from these meshes but can interpolate onto them. There is also the underlying issue of certain function spaces, such as those built from Enriched Elements, not having an implemented dual evaluation (see Sect. 3.8.3). These are often found in weather, ocean and climate model simulations (for example in the Gusto dynamical core [102] which is built on Firedrake). Implementation would therefore allow cross-mesh interpolation to be used there and bring all the benefits highlighted in Sect. 8.1.

The vertex-only mesh is a significant performance bottleneck when creating a cross-mesh interpolator, due to the factors highlighted in Sect. 7.8. Cross-mesh interpolation operations via point evaluation node locations necessarily involve very large numbers of points so both serial and parallel performance and scalability are important. Where one wishes to perform multiple interpolations with a static interpolator (for example via two way model coupling) this may be acceptable but otherwise this will limit the possible use cases of the cross-mesh interpolation as it is presently implemented.

The mathematical description at the beginning of Sect. 8.3 jumps straight into using global coordinates. This is only necessary where we cannot perform dual evaluation on the reference cell, which would become an issue when we have dual basis functionals with more than one point evaluation (see the original description of dual evaluation in Sect. 3.2). Where a target reference cell's coordinates are some subdomain of the source reference cell, this is not an issue and performing the summation in reference space on the reference cell could be more performant and could make direct use of existing dual evaluation routines. However, making this work with vertex-only meshes, which have solved so many issues with cross-mesh interpolation, would be a challenge.

The missing piece of the puzzle for evaluating line, surface and volume integrals, either for direct use (Listing 14) or for optimisation (Listing 15) is an easy to use utility function for creating immersed manifold meshes in Firedrake or for linking existing meshes to the locations in a domain at which they should be evaluated. For example, the line mesh created in Listing 14 uses the internal Firedrake function `plex_from_cell_list` rather than a visible part of the

[15]these are meshes with coordinate fields which are not in the degree-1 Lagrange function space: these are often referred to as 'bendy meshes' because their edges are generally not straight

API. So long as we are able to get cell connectivity information and vertex coordinates in the geometric dimension of the domain (the contents of `cells` and `vertex_coords` in Listing 14, respectively) this ought to be achievable using similar code to Listing 14.

When calculating the flux on an immersed manifold, as in Listing 15, the assembly operation will only work after the reference cells have been given so-called "cell orientation" information. This is a vector-valued UFL expression for a vector field which always points the same way through the manifold: the cell normal is defined using this direction.[16] This is not an implementation detail, it is vital information that must be supplied if we want to calculate anything which involves a surface normal (a Möbius strip, for example, is not an orientable manifold and cannot have a flux through it calculated). For simple cases where surfaces are planar, this can be a straightforward expression of one of the basis vector directions, depending on which direction the normal should point. Similarly for ellipsoids, the expression needs to merely be a vector field pointing radially outward from some point within the ellipsoid.[17] As soon as lines and surfaces become more complicated, such as a line or curve in an S shape, this becomes much more difficult. Careful consideration about how to make this a user friendly experience would be needed.

For generic external data input (Sect. 8.4.2), we are missing a generic tool for creating a triangulation given a list of points, ideally implemented directly in Firedrake. This is a solved problem, a Delaunay triangulation, for example, does exactly this. Ideally the final Firedrake implementation shouldn't need to gather point locations from all MPI ranks (assuming each rank contains contiguous locations), instead it should be able to build a mesh using the specified locations on each rank. Very recently, the NETGEN mesh generator [103] has been integrated with Firedrake: this may, therefore, already be possible. In the given recipe, step 5, dual evaluation interpolation from P0DG(Ω_v) onto P1CG(Ω_{data}), has not yet been explicitly implemented in Firedrake as a single call to interpolate. Indeed, for completeness and consistency, we should allow interpolation from vertex-only meshes in all cases: if we have point evaluation nodes which are at locations where we have vertices, we should let them be set. Lastly, if data are not stored as field point evaluations, perhaps instead as some set of quadrature point values in a given function space, we would require the extension of cross-mesh interpolation to non-point evaluation nodes: the machinery needed to specify quadrature points and sum them up (described earlier in this section) would prove useful at this point.

The coupling included here is a proof of concept: more mathematical and testing work needs to be done to verify that coupling works without issues. A starting point would be to solve the same problem on two meshes and two function spaces which are identical in all but their position relative to one another, such that the boundary between them is artificial. The solution on these meshes versus an equivalent single mesh (and single function space) could be compared.

[16] The specific Firedrake method to call on a mesh is `.init_cell_orientations(ufl_expression)`.

[17] For an ellipsoid around the origin called `mesh`, this would be `mesh.init_cell_orientations(SpatialCoordinate(mesh))` since `SpatialCoordinate(mesh)` is the vector field of mesh coordinates.

For a more thorough study, one could use the Method of Manufactured Solutions to check that, as the meshes are refined, the true solution is approached at the expected convergence rate. Once symbolic expression of interpolation is available, the full system of PDEs ought to be expressible in a single variational form which can be assembled: i.e. terms like the coupling penalty (Eq. 8.4.16) will be expressible without having to first calculate the solution to one of the PDEs.

A demonstration of PDE constrained optimisation using cross-mesh interpolation should be made. This could use the example of coupling shown here: some samples of parameters in both the Poisson and Helmholtz problems could be chosen as controls and the coupled problems optimised to give the best fit.

Uses for line, plane, surface and volume sources and sinks should be explored, given the mathematical description described in Sect. 8.4.3. As long as one has enough point evaluation nodes, these can be used as forcing terms to represent a wide range of phenomena: for example, a heat source in the heat equation. Recent work that uses Graph Neural Networks to aid weather forecasting [104] requires an 'encoding' step, which could be an interpolation onto a low resolution mesh of graph points at the point evaluation node locations, followed by a 'decoding' step which could be either interpolating back or, for a hybrid machine-learning and numerical model method, using those node locations as forcing terms. One could also import data from an external data set, which does not match the mesh, using the process identified in Sect. 8.4.2 and then use them as source or sink terms. For example, aerosol or chemical processes in a climate model are generally solved on different meshes to those where fluid flow occur. These could now be introduced as forcing terms S in the transport equation

$$\frac{\partial \phi}{\partial t} + \nabla \cdot \boldsymbol{u}\phi = S \tag{8.5.1}$$

for the fluid flow where ϕ is the density of the species (aerosol or chemical) being transported, and u is the velocity.

8.6 Summary of Contributions

I have implemented, for the first time, fully parallel compatible mesh-to-mesh interpolation in Firedrake. The meshed domains can have fully or only partially overlapping domains and the adjoint (transpose) interpolation operation has been implemented. A full test suite has been written to test all these features.

I have demonstrated here that this can be used for moving functions across meshes and function spaces, for calculating diagnostics by interpolating onto immersed domains, and for coupling models together. Alongside the input-ordering vertex-only mesh, I have shown how this can be used for generic external data input.

As a result of my implementation of both the forward and adjoint interpolation operations, my work is fully compatible with pyadjoint via `firedrake.adjoint` thanks to the work detailed in Sect. 5.5. I have added Taylor convergence tests to ensure this works as expected. This is the first time, to my knowledge, that parallel compatible mesh-to-mesh interpolation capability has been integrated with a high-level automatic differentiation system.

Chapter 9

Symbolic Interpolation

This chapter as a whole draws on ideas from [30].

9.1 Motivation

Recall from 1.3 that solving nonlinear variational problems 'find $u \in U$ such that

$$F(u; v) = 0 \;\; \forall\, v \in V, \text{'} \tag{9.1.1}$$

is done in Firedrake with Newton's method: For some small perturbation to the solution u, which we call u', the residual, F, can be approximated by the first two terms in its Taylor expansion

$$F(u + u'; v) \approx F(u; v) + \partial F_u(u; v, u') \tag{9.1.2}$$

where $\partial F_u(u; v, u')$ is the Gateaux derivative of F with respect to u in the perturbation direction u'. Newton's method has us starting with some some guess of the solution u^n, and finding the perturbation direction u', that sets these two terms equal to zero: 'For $u^n \in U$ find $u' \in U$ such that

$$F(u^n; v) + \partial F_u(u^n; v, u') = 0 \;\; \forall\, v \in V.\text{'} \tag{9.1.3}$$

The improved guess, u^{n+1}, is then calculated from the identified perturbation direction

$$u^{n+1} = u^n + u' \tag{9.1.4}$$

which is then the new u^n. Equation 9.1.3 is a linear variational problem with

$$a = \partial F_u(u^n; v, u') \text{ and} \tag{9.1.5}$$

$$L = -F(u^n; v). \tag{9.1.6}$$

This iterative set of linear solves, using both F and its first derivative ∂F_u, happens after the statement of the problem in UFL, using the symbolic information UFL provides when the problem is assembled, for example when calling `Firedrake.solve`.

At the moment when we request a dual evaluation interpolation be done with Firedrake, we immediately perform the interpolation, and therefore lose any symbolic information about whatever we interpolated. If the problem we want to solve requires us to interpolate the PDE solution, we therefore cannot solve it if it is nonlinear, since we cannot re-evaluate F and ∂F_u using the new solution at each iteration. To solve such a problem we need to find a way to express interpolation symbolically in UFL, and provide that information to our assembly and solver routines.

The problem outlined could potentially be solved by supplying an operation to apply to the solution between each Newton iteration. Such a change would not be consistent with Fire-

[0]Department of Mathematics, Imperial College London

drake's approach to solving problems: Where possible, operations are represented by symbolic mathematics in UFL and specific calculations are performed at assembly. This is the design philosophy that gives pyadjoint/dolfin-adjoint so many advantages, as laid out in Sect. 5.4. Dual evaluation interpolation clearly can be expressed using symbolic maths so there is no reason to jump ahead to calculation early.

As a motivating example, consider solving Burgers' equation for advection and diffusion of momentum

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u - \nu\nabla^2 u = 0 \tag{9.1.7}$$

on a 2D domain Ω with boundary condition

$$(\boldsymbol{n} \cdot \nabla)u = 0 \ \in \Omega_\Gamma. \tag{9.1.8}$$

u is a velocity field, $\boldsymbol{n}$ is an outward facing normal to the edge of the domain Ω_Γ and ν is a viscosity term. This is a nonlinear PDE which is one of the first demos in the Firedrake documentation [5].

In that demo, ν is a constant. However, we could be solving it for some ν derived from an implicit nonlinear constitutive law

$$f(u, \nu;) = 0 \tag{9.1.9}$$

where ν depends on u and we do not have an explicit relationship between them (i.e. there is no explicit $\nu(u;)$ such as $\nu = 0.5(\nabla(u) + \nabla(u)^T)$). These appear in continuum mechanics (see, for example, [105] where the relation is between viscosity and pressure). If this relation involved derivatives of u it is a differential equation, if not it is an algebraic equation. In either case we have to solve for both fluid velocity and viscosity both here and in Burgers' equation.

Burgers equation can be solved with the nonlinear variational problem 'find $u \in V_{\text{velocity}}$ such that

$$\begin{aligned} F^{\text{velocity}}(u, \nu; v) &= \int_\Omega \frac{\partial u}{\partial t} \cdot v + ((u \cdot \nabla)u) \cdot v + \nu \nabla u \cdot \nabla v \; dx \\ &= 0 \ \ \forall v \in V_{\text{velocity}}.' \end{aligned} \tag{9.1.10}$$

Assuming the nonlinear constitutive law is algebraic, we can choose to evaluate it at the quadrature points of a finite element function space $V_{\text{viscosity}}$ by interpolating it into that space[1]

$$\mathcal{I}_{V_{\text{viscosity}}}(; f(u, \nu;)) = 0. \tag{9.1.11}$$

[1]This is not intended to be a full analysis of the validity of such a discretisation, but gives us a useful motivating example.

From the definition of global interpolation (Eq. 3.2.15) this is

$$\sum_{i=0}^{\dim(V_{\text{viscosity}})-1} \bar{\phi}_i^*(; f(u,\nu;))\phi_i = 0 \tag{9.1.12}$$

where ϕ_i are the basis functions of $V_{\text{viscosity}}$ and $\bar{\phi}_i^*$ the extended dual basis of $V_{\text{viscosity}}^*$ where the extension is in the form of a quadrature rule (as in Firedrake, see Eq. 3.7.3). The basis functions are linearly independent so to sum to zero their coefficients must all be zero:

$$\bar{\phi}_i^*(; f(u,\nu;)) = 0 \quad \forall\, i. \tag{9.1.13}$$

Each of these can be multiplied by any set of i scalar weights to make some arbitrary member of the space of extended cofunctions

$$\bar{V}_{\text{viscosity}}^* : X \to \mathbb{R} \tag{9.1.14}$$

where, as in Sect. 3.2.2, X is the union of all function spaces to which each $\bar{\phi}_i^*$ can be applied. This gives us a second, nonlinear problem to solve 'find $\nu \in V_{\text{viscosity}}$ such that

$$\begin{aligned} F^{\text{viscosity}}(u,\nu;\bar{\gamma}^*) &= \bar{\gamma}^*(; f(u,\nu;)) \\ &= 0 \quad \forall\, \bar{\gamma}^* \in \bar{V}_{\text{viscosity}}^*\text{'} \end{aligned} \tag{9.1.15}$$

If we consider this without being coupled to Burgers equation, each Newton loop iteration will have us solving the linear problem 'for some $\nu^n \in V_{\text{viscosity}}$, find $\nu' \in V_{\text{viscosity}}$ such that

$$\begin{aligned} \partial F_\nu^{\text{viscosity}}(u,\nu^n;\bar{\gamma}^*,\nu') &= -F_\nu^{\text{viscosity}}(u,\nu^n;\bar{\gamma}^*) \\ = \bar{\gamma}^*(; \partial f_\nu(u,\nu^n;\nu')) &= -\bar{\gamma}^*(; f(u,\nu^n;)) \quad \forall\, \bar{\gamma}^* \in \bar{V}_{\text{viscosity}}^*\text{.'} \end{aligned} \tag{9.1.16}$$

In order to solve this we need to be able to represent the dual evaluation interpolation of $f(u,\nu^n;)$ and $\partial f_\nu(u,\nu^n;\nu')$ into $V_{\text{viscosity}}$ symbolically.

9.2 Existing Capability Outside Firedrake

Due to the previously described different design decisions and limitations of most other finite element libraries, there is no equivalent to symbolic interpolation elsewhere. One can instead, however, look at how the motivating problem of operating on a solution between Newton iterations are solved.

None of the libraries which use UFL (legacy FEniCS, its rewrite, FEniCSx, and DUNE-Fem) can solve these problems without manually implementing the Newton solver to apply operations between iterations. Other libraries which operate at a lower level of abstraction also require direct implementation of the Newton solver: This is the case for deal.II[2] and NGSolve[3]. As well as losing the level of flexibility and abstraction UFL provides, one generally has to operate within the context of compiled programming languages in order to ensure speedy operation. It should be noted that MFEM has nonlinear form types but it is not clear from their documentation how MFEM would cope with such a problem.

9.3 Multilinear Forms as Linear Operations Between Hilbert Spaces

This section as a whole draws on ideas from Bouziani and Ham [106] and Ham [30].

In section 1.4 we considered a 2-linear (bilinear) form

$$M^{\text{previous}} : V_1 \times V_0 \to \mathbb{R} \tag{9.3.1}$$

which we Curried by fixing one of the arguments to find the inverse Riesz map M_1, which is equal to the covector M_0.

$$M^{\text{previous}}(;v_1, \bullet) = M_1^{\text{previous}}(;v_1)(;\bullet) = M_0^{\text{previous}}(;\bullet) \in V_0^*. \tag{9.3.2}$$

Let's look at what happens if we start with a 2-linear form which contains an argument from a dual space:

$$M : U \times V^* \to \mathbb{R} \text{ i.e. } M(;u, v^*) \in K \;\forall\, u \in U, v^* \in V^*. \tag{9.3.3}$$

We start by Currying it into two new functions M_1 and M_0

$$M(;u, v^*) = \underbrace{M_1(;u)}_{M_0}(;v^*) = M_0(;v^*) \tag{9.3.4}$$

where

$$M_1 : U \to (\underbrace{V^* \to \mathbb{R}}_{V^{**}}) \tag{9.3.5}$$

only evaluates u, leaving v^* as a free argument such that

$$M_1(;u)(;\bullet) \in V^{**} \quad \forall\, u \in U \tag{9.3.6}$$

[2]see the introductory example of solving a nonlinear PDE `https://www.dealii.org/current/doxygen/deal.II/step_15.html`

[3]see the nonlinear problems tutorial `https://docu.ngsolve.org/v6.2.1910/i-tutorials/unit-3.7-nonlinear/nonlinear.html`

and

$$M_0 : \underbrace{V^* \to \mathbb{R}}_{V^{**}} \tag{9.3.7}$$

already includes the evaluation of u such that

$$M_0(;v^*) \in K \quad \forall\, v^* \in V^*. \tag{9.3.8}$$

Since our space of primal vectors V is a Hilbert space, our space of covectors V^* is also a Hilbert space[4]. This implies that for each covector in V^* you can get a co-covector in V^{**} using the Riesz representation theorem. Hilbert spaces have the property that they are *reflexive* meaning that they are naturally isomorphic under the canonical evaluation map given by

$$v^{**}(;v^*) = v^*(;v) \in K \tag{9.3.9}$$

for all $v \in V$ with covector $v^* \in V^*$ and co-covector $v^{**} \in V^{**}$.[5] The isomorphism is 'natural' in the sense that it comes immediately from the properties of V and V^{**}: there is no dependency on other choices such as a particular basis. All vector space operations and norms are preserved through the evaluation map. We can therefore use V and V^{**} interchangeably

$$V^{**} \cong V \tag{9.3.10}$$

with members of V operating on members of V^* such that

$$\begin{aligned} v(;v^*) &\equiv v^{**}(;v^*) \\ &= v^*(;v) \in K \quad \forall\, v^* \in V^*. \end{aligned} \tag{9.3.11}$$

Note that if our vector space is a space of functions this **does not imply the evaluation of the functions**.

[4]This can be proved with the Riesz representation theorem.

[5]A note on this: The inverse Riesz representers in both these cases are

$$v^*(;\bullet) = \langle v, \bullet \rangle_V : V \to \mathbb{R}\ \forall\, v \in V \tag{9.3.12}$$

$$v^{**}(;\bullet) = \langle v^*, \bullet \rangle_{V^*} : V^* \to \mathbb{R}\ \forall\, v^* \in V^* \tag{9.3.13}$$

What is this inner product on V^*? In spaces with a finite number of basis functions we can represent the inner product as

$$\langle v, \nu \rangle_V = \sum_{i,j}^{\dim(V)} v_{1i} M_{ij} \nu_j \tag{9.3.14}$$

and

$$\langle v^*, \nu^* \rangle_{V^*} = \sum_{i,j}^{\dim(V^*)} v^*_{1i} M^{\text{mystery}}_{ij} \nu^*_j \tag{9.3.15}$$

Eq. 9.3.9 requires that

$$M^{\text{mystery}} = M^{-1}. \tag{9.3.16}$$

Returning to our newly Curried forms, the operation

$$M_1 : U \to V^{**} \tag{9.3.17}$$

is, through the isometric isomorphism, indistinguishable from

$$\tilde{M}_1 : U \to V. \tag{9.3.18}$$

This has two impacts: firstly it tells us that the 1-linear form

$$M_0(;\bullet) = M_1(;u)(;\bullet) = M(;u,\bullet) \in V^{**} \cong V. \tag{9.3.19}$$

where the assembled output is a co-covector (the missing argument in M is a covector $v^* \in V^*$) can be treated as a primal vector (a function). Compare this with the 1-linear form example in section 2.4.2: there after assembly we got a covector (in UFL a new `Cofunction`), now we get a vector (a `Coefficient`).

More broadly, we now have an equivalence for the linear operator between Hilbert spaces

$$\tilde{M}_1 : U \to V \tag{9.3.20}$$

and the uncurried 2-linear (bilinear) form

$$M : U \times V^* \to \mathbb{R} \tag{9.3.21}$$

since we can replace $\tilde{M}_1$ with M_1. The action of supplying an operand to the linear operator $\tilde{M}_1$ gives the same result, via the isomorphism, as supplying that operand to the 2-linear form giving a 1-linear form:

$$\tilde{M}_1(;u)(;\bullet) \in V \cong V^{**} \quad \forall\, u \in U \tag{9.3.22}$$

$$= M(;u,\bullet) \in V^{**} \cong V \quad \forall\, u \in U. \tag{9.3.23}$$

If we supply some $v^* \in V^*$ to the 2-linear form we have

$$M(;u,v^*) = M_1(;u)(;v^*) = v^*(;M_1(;u)) \tag{9.3.24}$$

from the canonical evaluation map (Eq. 9.3.9). If we supply neither operand we have

$$M(;\bullet,\bullet) = M_1(;\bullet)(;\bullet) \tag{9.3.25}$$

which is the raw operator.

V and V^{**} are henceforth treated as indistinguishable so any distinction between $\tilde{M_1}$ and M_1 is dropped.

Now we can represent linear operators as forms, we ought to be able to express them in UFL and assemble them with Firedrake. For a given 2-linear form M, the UFL types supplied dictate the symbolic meaning of the operation performed and the assembled output:

1. If we supply both arguments to M, i.e.

   - $u \in U$ is a known `ufl.Coefficient` and
   - $v^* \in V^*$ is a known `ufl.Cofunction`

   implying UFL code

   ```
   u = Coefficient(U)
   v_star = Cofunction(V.dual())
   zero_form = M(u, v_star)
   ```

   Symbolically, `zero_form` is a 0-linear form which symbolically corresponds to

   $$M(;u, v^*) = v^*(;M_1(;u)) = M_1^*(;v^*)(;u) \in K. \tag{9.3.26}$$

   If we add data to our symbolic types and assemble this with Firedrake we get a scalar $\in K$, concretely a Python `float` or `complex`.

   The second equality in Eq. 9.3.26 comes from the definition of an adjoint operator (equation 2.1.2) we have that

   $$v^*(;M_1(;u)) = M_1^*(;v^*)(;u) \; \forall \, v^* \in V^*, u \in U. \tag{9.3.27}$$

   where

   $$M_1^* : V^* \to U^*. \tag{9.3.28}$$

   For finite element spaces, M_1^* is evaluated by having the covector $v^* \in V^*$ operate on M_1 (see Sect. 5.4.1, in particular Eq. 5.4.7).

2. If we supply only the primal argument to M, i.e.

   - $u \in U$ is a known `ufl.Coefficient` and
   - $v^* \in V^*$ is an unknown `ufl.Coargument` (argument slot 0, a `ufl.TestFunction` on a dual function space)

   then we have UFL code

```
u = Coefficient(U)
v_star = Coargument(V.dual(), 0)  # Or TestFunction(V.dual())
one_form_A = M(u, v_star)
```

Symbolically, `one_form_A` is a 1-linear form which symbolically corresponds to the linear operation M_1

$$M(;u,\bullet) = M_1(;u)(;\bullet) \in V. \tag{9.3.29}$$

If we assemble this with Firedrake we get a `firedrake.Function` (a `ufl.Coefficient`) in the `firedrake.FunctionSpace`, `V`.

3. If we supply only the dual argument to M, i.e.

   - $u \in U$ is an unknown `ufl.Argument` (argument slot 0, a `ufl.TestFunction`) and
   - $v^* \in V^*$ is a known `ufl.Cofunction`

   then we have UFL code

   ```
   u = Argument(U, 0)  # Or TestFunction(U)
   v_star = Cofunction(V.dual())
   one_form_B = M(u, v_star)
   ```

   Symbolically, `one_form_B` is a 1-linear form which symbolically corresponds to the adjoint linear operation M_1^*

   $$M(;\bullet,v^*) = M_1^*(;v^*)(;\bullet) \in U^*. \tag{9.3.30}$$

   If we assemble this with Firedrake we get a `firedrake.Cofunction` (a `ufl.Cofunction`) in the `firedrake.FiredrakeDualSpace`, `U.dual()`.

4. If we supply neither argument to M, i.e.

   - $v^* \in V^*$ is an unknown `ufl.Coargument` (argument slot 0, a `ufl.TestFunction` on a dual function space)

   then we have UFL code

   ```
   u = Argument(U, 1)  # Or TrialFunction(U)
   v_star = Coargument(V.dual(), 0)  # Or TestFunction(V.dual())
   two_form = M(u, v_star)
   ```

   Symbolically, `two_form` is the 2-linear form we started with

   $$M : U \times V^* \to \mathbb{R} \tag{9.3.31}$$

which corresponds to our raw linear operator M_1. If we assemble this with Firedrake we get a $\dim(V) \times \dim(U)$ matrix (we use that $\dim(V) = \dim(V^*)$). Symbolically the output is the new UFL `ufl.Matrix` type that was introduced with `ufl.Cofunction` and `ufl.Coargument`.

In UFL we can take a symbolic form but change the argument order. For example, we might take our form M from above and assemble the 2-linear form with the argument slots reversed:

```
u = Argument(U, 0)  # before this was Argument(U, 1)
v_star = Coargument(V.dual(), 1)  # before this was Coargument(V.dual(), 0)
two_form = M(u, v_star)
```

This represents a new form

$$M^* : V^* \times U \to \mathbb{R} \tag{9.3.32}$$

which uses the adjoint to our linear operator M_1

$$M^*(; v^*, u) = u(; M_1^*(; v^*)). \tag{9.3.33}$$

We call M^* the adjoint to M and, when we use the UFL function `ufl.adjoint` on M we get M^*. The assembly rules with swapped arguments follow the rules for M above, but with M swapped for M^* and M_1 swapped for M_1^*. So in this case, where we assemble a 2-linear form, we get a $\dim(U) \times \dim(V)$ matrix (again, $\dim(U) = \dim(U^*)$ rather than a $\dim(V) \times \dim(U)$ one.

The significance of representing linear operations as forms is that we can always represent them symbolically in a form language. We only need to perform the operation itself when we reach the point of assembly. This has two clear advantages. Raising the level of abstraction like this allows one to derive things like the derivative and to compose operations at a high level without having to perform the operation itself. As was highlighted in Sect. 5.4, that is the principal advantage of dolfin-adjoint/pyadjoint's approach to automatic differentiation. This work has played a significant role in allowing general operations from outside Firedrake to be integrated with it, including giving a simple interface for integrating with dolfin-adjoint/pyadjoint as detailed in Bouziani and Ham [106].

9.3.1 Dual Evaluation Interpolation as a Form

Dual evaluation interpolation, defined in defined in Sect. 3.2, is a linear operator between function spaces

$$\mathcal{I}_V : U \to V. \tag{9.3.34}$$

Treating this as M_1[6], in equation 9.3.23, we come up with an equivalent uncurried 2-linear form (M) which we will call 'interp'.

[6]technically $\tilde{M}_1$, but recall that we are treating V and V^{**} as indistinguishable so the tilde is dropped

Recall from Sect. 3.2 (Eq. 3.2.19) that when we have $u = \sum_{i=0}^{\dim(U)-1} u_i\psi_i \in U$ and $v = \sum_{i=0}^{\dim(V)-1} u_i\phi_i \in V$, then we find a $\dim(V) \times \dim(U)$ interpolation matrix A_{ij}

$$\left[\mathcal{I}_V(;u)\right](x) = \sum_{i=0}^{\dim(V)-1} \bar{\phi}_i^*(;u)\phi_i(x) \tag{9.3.35}$$

$$= \sum_{i=0}^{\dim(V)-1} \sum_{i=0}^{\dim(U)-1} A_{ij}u_j\phi_i(x) \in V \;\forall\, u \in U. \tag{9.3.36}$$

In much the same way as the mass matrix is associated with assembling an L^2 inner product (see Sect. 2.4.2), the interpolation matrix is associated with assembling the interp form.

In UFL, we call this form `ufl.Interpolate` and it is used as follows:

1. Starting with interp as a 0-linear form such that

   - $u \in U$ is a known `ufl.Coefficient` and
   - $v^* \in V^*$ is a known `ufl.Cofunction`

   then we have UFL code

   ```
   u = Coefficient(U)
   v_star = Cofunction(V.dual())
   zero_form = Interpolate(u, v_star)
   ```

   Symbolically, `zero_form` is

$$\text{interp}(;u, v^*) = v^*(;\mathcal{I}_V(;u)) = \mathcal{I}_V^*(;v^*)(;u) \tag{9.3.37}$$

$$= \sum_{k=0}^{\dim(V)-1} \sum_{i=0}^{\dim(V)-1} \sum_{j=0}^{\dim(U)-1} v_k^*\phi_k^*(;A_{ij}u_j\phi_i) \tag{9.3.38}$$

$$= \sum_{k=0}^{\dim(V)-1} \sum_{i=0}^{\dim(V)-1} \sum_{j=0}^{\dim(U)-1} v_k^*A_{ij}u_j \underbrace{\phi_k^*(;\phi_i)}_{\delta_{ki}} \tag{9.3.39}$$

$$= \sum_{i=0}^{\dim(V)-1} \sum_{j=0}^{\dim(U)-1} v_i^*A_{ij}u_j \in K. \tag{9.3.40}$$

   If we assemble this with `firedrake.assemble` we perform the sums and get a scalar $\in K$, typically be a Python `float` or `complex`.

   As before

$$\mathcal{I}_V^* : V^* \to U^* \tag{9.3.41}$$

   is the adjoint interpolation operator.

2. If we have a 1-linear form such that

   - $u \in U$ is a known `ufl.Coefficient` and
   - $v^* \in V^*$ is an unknown `ufl.Coargument` (argument slot 0, a `ufl.TestFunction` on a dual function space)

   then we have UFL code

   ```
   u = Coefficient(U)
   v_star = Coargument(V.dual(), 0)  # or TestFunction(V.dual())
   one_form_A = Interpolate(u, v_star)
   ```

   then, symbolically, `one_form_A` is the dual evaluation interpolation operation

$$\begin{aligned} \text{interp}(;u,\bullet) &= \mathcal{I}_V(;u)(;\bullet) && (9.3.42)\\ &= \left(\sum_{i=0}^{\dim(V)-1}\sum_{j=0}^{\dim(U)-1} A_{ij}u_j\phi_i\right)(;\bullet). && (9.3.43)\end{aligned}$$

   If we assemble this we get a `firedrake.Function` (a `ufl.Coefficient`) in V.

3. If we have a 1-linear form such that

   - $u \in U$ is an unknown `ufl.Argument` (argument slot 0, a `ufl.TestFunction`) and
   - $v^* \in V^*$ is a known `ufl.Cofunction`

   i.e. we have UFL code

   ```
   u = Argument(U, 0)  # or TestFunction(U)
   v_star = Cofunction(V.dual())
   one_form_B = Interpolate(u, v_star)
   ```

   then, symbolically, `one_form_B` is the adjoint dual evaluation interpolation operation

$$\begin{aligned} \text{interp}(;\bullet,v^*) &= \mathcal{I}_V^*(;v^*)(;\bullet) && (9.3.44)\\ &= \sum_{i=0}^{\dim(V)-1}\sum_{j=0}^{\dim(U)-1} v_i^*A_{ij}\psi_j^*(;\bullet) \in U^{*}[7] && (9.3.45)\end{aligned}$$

   i.e. the jth global basis cofunction coefficient in U^* is given by $\sum_{i=0}^{\dim(V)-1} v_i^*A_{ij}$. If we assemble this we get a `firedrake.Cofunction` (`ufl.Cofunction`) in U^*.

   This is the operation performed when doing the 'adjoint action' for the annotation of interpolation in pyadjoint: see section 5.5.1 and specifically the code on page 118.

[7]To prove this is correct, consider supplying this expression with some $u = \sum_{k=0}^{\dim(U)-1} u_k\psi_k$, making it a

4. If we have a 2-linear form such that

   - $u \in U$ is an unknown `ufl.Argument` (argument slot 1, a `ufl.TrialFunction`) and
   - $v^* \in V^*$ is an unknown `ufl.Coargument` (argument slot 0, a `ufl.TestFunction` on a dual function space)

   i.e. we have UFL code

   ```
   u = Argument(U, 1)  # or TrialFunction(U)
   v_star = Coargument(V.dual(), 0)  # or TestFunction(V.dual())
   two_form = Interpolate(u, v_star)
   ```

   then, symbolically, `two_form` is the dual evaluation interpolation operation (the interpolation matrix) itself. If we assemble this we get the $\dim(V) \times \dim(U)$ interpolation matrix A_{ij} as a `firedrake.Matrix`) (itself now a `ufl.Matrix`).

As before, we can swap the UFL `ufl.Argument` and `ufl.Coargument` argument slot numbering

```
u = Argument(U, 0)  # or TestFunction(U)
v_star = Coargument(V.dual(), 1)  # or TrialFunction(V.dual())
adj_two_form = Interpolate(u, v_star)
```

and assemble the 2-form

$$\text{interp}^* : V^* \times U \to \mathbb{R} \tag{9.3.49}$$

to get the adjoint interpolation operator $\mathcal{I}_V^*$ with equivalent $\dim(U) \times \dim(V)$ matrix A^*_{ji}.[8]

9.4 Using the interp Form in UFL and Firedrake

Thus far, we have considered using interp for the linear dual evaluation operation between finite element function spaces. As was discussed in Sect. 5.5, Firedrake's interpolation operation

0-linear form:

$$\text{interp}(;u,v^*) = \sum_{i=0}^{\dim(V)-1} \sum_{j=0}^{\dim(U)-1} \sum_{k=0}^{\dim(U)-1} v_i^* A_{ij} \psi_j^*(;u_k\psi_k) \tag{9.3.46}$$

$$= \sum_{i=0}^{\dim(V)-1} \sum_{j=0}^{\dim(U)-1} \sum_{k=0}^{\dim(U)-1} v_i^* A_{ij} u_k \underbrace{\psi_j^*(;\psi_k)}_{\delta_{jk}} \tag{9.3.47}$$

$$= \sum_{i=0}^{\dim(V)-1} \sum_{j=0}^{\dim(U)-1} v_i^* A_{ij} u_j \in K \tag{9.3.48}$$

which is the expression for the 0-linear form in Eq. 9.3.40.

[8]Equivalently we can use UFL's symbolic `adjoint` operator: `adjoint(Interpolate(u, v_star))` where `u` is a `ufl.TrialFunction` and `v_star` is `ufl.Coargument` 0 to get A^*_{ji}.

covers both the dual evaluation interpolation operator $\mathcal{I}$, which is linear in its arguments, and evaluation of some expression 'expr', which may not be linear in its arguments. With this in mind, interp can be treated in the same way. When calculations involving interp are evaluated, the expression will either (a) have all its arguments specified with a `ufl.Coefficient` and `ufl.Cofunction`, in which case the expression can simply be evaluated prior to dual evaluation, or (b) will have a `ufl.Argument`, `ufl.Coargument`, or both, in which case we have a nonlinear problem which will require linearisation and iteration to find a solution (as previously discussed).

Returning to our motivating example, we need to be able to express 'find $\nu \in V_{\text{viscosity}}$ such that

$$\begin{aligned} F^{\text{viscosity}}(u, \nu; \bar{\gamma}^*) &= \bar{\gamma}^*(; f(u, \nu;)) \\ &= 0 \quad \forall \bar{\gamma}^* \in \bar{V}^*_{\text{viscosity}}, \end{aligned} \tag{9.4.1}$$

as part of our problem statement. This can now be directly represented with the new interp 2-linear form as 'find $\nu \in V_{\text{viscosity}}$ such that

$$\text{interp}(; f(u, \nu;), \gamma^*) = 0 \quad \forall \gamma^* \in V^*_{\text{viscosity}}. \text{'} \tag{9.4.2}$$

In UFL we would express this as

```
u = Coefficient(V_velocity)
nu = Coefficient(V_viscosity)
gamma_star = TestFunction(V_viscosity.dual())
f_expr = f(u, nu)  # = 0
F_viscosity = Interpolate(f_expr, gamma_star)
```

where `f` is a function that returns a nonlinear UFL expression of `u` and `nu` representing the nonlinear constitutive law. Here, `gamma_star` is the new `Coargument` type.[9]

When assembling this, we need to be able to solve the following linear problem during each Newton iteration: 'For some $\nu^n \in V_{\text{viscosity}}$, find $\nu' \in V_{\text{viscosity}}$ such that

$$\bar{\gamma}^*(; \partial f_\nu(u, \nu^n; \nu')) = -\bar{\gamma}^*(; f(u, \nu^n;)) \quad \forall \bar{\gamma}^* \in \bar{V}^*_{\text{viscosity}}, \tag{9.4.3}$$

(Eq. 9.1.16). This can be represented as 'For some $\nu^n \in V_{\text{viscosity}}$, find $\nu' \in V_{\text{viscosity}}$ such that

$$a(; \nu', \gamma^*) = L(; \gamma^*) \quad \forall \gamma^* \in V^*_{\text{viscosity}}. \text{'} \tag{9.4.4}$$

[9]Notice that the interp form uses cofunctions γ^* rather than $\bar{\gamma}^*$ in the problem statement. This is intentional: the unbarred cofunctions can be thought of as selecting the target interpolation space, in this case $V_{\text{viscosity}}$, but are not themselves involved in dual evaluation. Nevertheless, if they are a `ufl.Argument` they will, once linearised, lead to the correct dual evaluation calculation detailed in the previous section. See equations 9.4.7 and 9.4.8 for the linearisation.

where

$$a(;\nu', \gamma^*) = \text{interp}(;\partial f_\nu(u, \nu^n; \nu'), \gamma^*) \tag{9.4.5}$$
$$L(;\gamma^*) = -\text{interp}(; f(u, \nu^n), \gamma^*). \tag{9.4.6}$$

In Firedrake we write this as

```
u = Function(V_velocity)
nu_n = Function(V_viscosity)
...  # assign values to u and nu_n
gamma_star = TestFunction(V_viscosity.dual())
nu_prime = TrialFunction(V_viscosity)
f_expr = f(u, nu_n)  # = 0
a = Interpolate(derivative(f_expr, nu_prime), gamma_star)
L = -Interpolate(f_expr, gamma_star)
nu_prime_sol = Function(V_viscosity)
solve(a == L, nu_prime_sol)
```

Assembling a gives a $\dim(V_{\text{viscosity}}) \times \dim(V_{\text{viscosity}})$ matrix which is the interpolation operator $\mathcal{I}_{V_{\text{viscosity}}}$ acting on the partial Gateaux derivative ∂f_ν. Using the notation from chapter 5, we can write this as

$$a_{ij} = \left[\mathcal{I}_{V_{\text{viscosity}}} \circ \left. \frac{\partial f}{\partial \nu} \right|_{u,\nu^n} \right]_{ij} \tag{9.4.7}$$

which acts on a primal trial function $\nu' \in V_{\text{viscosity}}$. Assembling L gives a $\dim(V_{\text{viscosity}})$ vector corresponding to the interpolation of f into $V_{\text{viscosity}}$ (a primal function):

$$L_i = -\mathcal{I}_{V_{\text{viscosity}}}(; f(u, \nu))_i. \tag{9.4.8}$$

This is in contrast to the assembly of a 1-linear form when solving a PDE which gives the basis coefficients of a cofunction.[10] These can be then be passed to a linear solver at each Newton iteration, producing the coefficients of a new primal function $\nu' \in V_{\text{viscosity}}$ at each step.

9.4.1 Demonstration Code

At time of writing (January 2024), the interp form has just been implemented (albeit not directly by the author) in both UFL and Firedrake. When a symbolic `Interpolate` form is identified in assembly, Firedrake's pre-existing interpolation functionality is used. Firedrake's assembly mechanisms have had to be extended to allow forms to be composed with other forms,

[10]Similarly, the matrix a_{ij} for solving a PDE (such as the mass matrix) acts on a the basis coefficients of a primal function, and gives the basis coefficients of a cofunction.

to allow for interpolation expressed with the interp form within another form, such as an L^2 inner product.

Code which demonstrates how the example problem would be solved is shown in Listing 18. It is not intended to represent a well conditioned problem, but exemplifies the concept.[11]

After a backwards Euler time discretisation F^{velocity} becomes 'find $u^{n+1} \in V_{\text{velocity}}$ such that

$$\begin{aligned} F_{\delta t}^{\text{velocity}}(u^{n+1}, u^n, \nu; v) &= \int_\Omega \frac{u^{n+1} - u^n}{\delta t} \cdot v + ((u^{n+1} \cdot \nabla) u^{n+1}) \cdot v + \nu \nabla u^{n+1} \cdot \nabla v \; dx \\ &= 0 \ \ \forall\, v \in V_{\text{velocity}}.\text{'} \end{aligned} \tag{9.4.9}$$

Firedrake solves coupled equations by formulating them as a single variational problem with solutions and test functions drawn from so-called 'mixed' finite element spaces. Such approaches were first discussed by Fraeijs de Veubeke [107] and Herrmann [108] and involve creating Cartesian products of function spaces from which solutions can be drawn. The formulation becomes 'find $(u^{n+1}, \nu) \in V_{\text{velocity}} \times V_{\text{viscosity}}$ such that

$$F((u^{n+1}, \nu); (v, \bar{\gamma}^*)) = 0 \ \ \forall\, (v, \bar{\gamma}^*) \in V_{\text{velocity}} \times \bar{V}^*_{\text{viscosity}}\text{'} \tag{9.4.10}$$

where

$$\begin{aligned} F((u^{n+1}, \nu); (v, \bar{\gamma}^*)) = \underbrace{\int_\Omega \frac{u^{n+1} - u^n}{\delta t} \cdot v + ((u^{n+1} \cdot \nabla) u^{n+1}) \cdot v + \nu \nabla u^{n+1} \cdot \nabla v \; dx}_{F^{\text{velocity}}} \\ + \underbrace{\bar{\gamma}^*(; f(u^{n+1}, \nu;))}_{F^{\text{viscosity}}}. \end{aligned} \tag{9.4.11}$$

To linearise and solve this using Newton's method, we have to take the Gateaux derivative of this new F with respect to a member of the mixed function space $V_{\text{velocity}} \times V_{\text{viscosity}}$. Each of the two terms of F only act on part of the mixed function, so we can take the Gateaux derivative of each such term separately and sum them to get the full derivative. This means our existing mathematics for taking the derivative of $F^{\text{viscosity}}$ is still what happens when we reformulate as a mixed problem.

9.5 Future Work

Eventually Firedrake's API could be simplified to have only two operations which directly manipulate data: function and cofunction assignment and assembly. Should one wish to eagerly perform an operation like interpolation[12], one would assemble the interp form. This ought to

[11]In time, the existing `firedrake.interpolate` function will be changed to return the symbolic output of the `Interpolate` form.

[12]to, for example, avoid symbolic expression swell when one comes to differentiate it

allow much more flexibility in the way that Firedrake can be used, as demonstrated here for the case of dual evaluation interpolation. Other concrete operations, such as Galerkin projection, will need to be identified and made symbolic.

9.6 Summary of Contributions

I have demonstrated how linear operations can be reformulated as forms. This allows us to represent them as symbolic objects which can be assembled at a later stage. This work was the joint work of me, David Ham, Colin Cotter and Nacime Bouziani, all of Imperial College London.

David Ham and I developed the theory of the interp form. This can be used to represent dual evaluation interpolation and its adjoint as forms, themselves linear operations. As I have shown here, this can be used to solve coupled problems where part of the problem formulation includes interpolating a nonlinear expression of one of the solutions.

This is a whole new class of problems which Firedrake should now be able to solve. I have demonstrated this for the case of the Burgers equation with a viscosity term dictated by a nonlinear constitutive law. At time of writing, the interp form has just been implemented in Firedrake as `firedrake.Interpolate`. Firedrake is not the only finite element library which could benefit from this, since the theory is general and forms are a common abstraction in finite element libraries.

```python
from firedrake import *

n = 30
mesh = UnitSquareMesh(n, n)
timestep = 1.0 / n

# Function Spaces
V_velocity = VectorFunctionSpace(mesh, "CG", 2)
V_velocity_out = VectorFunctionSpace(mesh, "CG", 1)
V_viscosity = FunctionSpace(mesh, "CG", 1)
W_sol = V_velocity * V_viscosity
W_test = V_velocity * V_viscosity.dual()

# Functions and Test Functions
w = Function(W_sol, name="State")
u_ = Function(V_velocity, name="Velocity")
w.subfunctions[0].name = "VelocityNext"
w.subfunctions[1].name = "Viscosity"
u, nu = split(w)  # split is for inserting into UFL expressions
v, gamma_star = TestFunctions(W_test)

# Initial Condition
x = SpatialCoordinate(mesh)
ic = project(as_vector([sin(pi * x[0]), 0]), V_velocity)
u_.assign(ic)
w.subfunctions[0].assign(ic)

# burgers equation residual
F_velocity = (
    inner((u - u_) / timestep, v)
    + inner(dot(u, nabla_grad(u)), v)
    + nu * inner(grad(u), grad(v))
) * dx

# Algebraic equation residual (for demonstration,
# can be made explicit!)
f = 10000 * nu - exp(-inner(u, u))
F_viscosity = Interpolate(f, gamma_star)

# Complete residual
F = F_velocity + F_viscosity

# Save initial condition
outfile = File("burgers.pvd")
outfile.write(project(u, V_velocity_out, name="Velocity"))

# Solve
t = 0.0
end = 0.5
while t <= end:
    solve(F == 0, w)
    u_.assign(w.subfunctions[0])  # update u_ with u
    t += timestep
    outfile.write(project(u, V_velocity_out, name="Velocity"))
```

Listing 18: Solving the Burgers equation with a viscosity term dictated by a nonlinear constitutive law.

Chapter 10

Conclusion

10.1 Conclusion

The aim here was to identify abstractions for representing arbitrary data in the finite element method, and operators for their interaction with scalar, vector and tensor fields represented as finite element functions. This has been successfully achieved this for point data which can now also be represented as finite element functions (Sect. 4.3): in this case, the coordinates are a vertex-only mesh and the values as the finite element function space of zeroth-order discontinuous Lagrange polynomials on that mesh (P0DG(Ω_v)). The dual evaluation interpolation operation (Sect. 3.2) has been identified as a suitable way of describing interactions between these data and finite element fields (Sect. 4.4).

The diverse range of scenarios where this abstraction proves useful demonstrates that it is sufficiently general: Point evaluation of fields represented as finite element functions can be done by interpolating onto the function space on the vertex-only mesh (mathematics in Sect. 4.4, implementation in Sect. 4.5.3). Point sources can be represented as these finite element functions with the interpolation operator and its adjoint giving us an appropriate forcing function (Sect. 4.6.1).

Differentiation of the interpolation operator has been demonstrated (Sect. 5.5). This allows optimisation problems and sensitivity analysis involving arbitrary point data and finite element models to be solved. As a demonstration of PDE constrained optimisation, it was shown that arbitrary point data can be assimilated, with ease, into finite element models in a way that is consistent with Bayes' Theorem (chapter 6).

By considering the dual bases of finite element functions as weighted sums of point evaluations, this work has been extended from point data to arbitrary external data represented on a domain as outlined in chapter 8. These data could already be represented as finite element functions, but the new abstraction of point data allows arbitrary dual evaluation interpolations between them and external input data (see Sect. 8.4.2). These can be used for interacting with data defined on lines planes or volumes, much as we could with point data: this work has shown how one can introduce line, plane or volume source or sink terms for a sufficiently dense mesh (see Sect. 8.4.3).

The ability to interpolate across meshes also allows for model coupling, another form of model-data interaction. The fact that this all starts from abstractions for point data again demonstrates the sufficiency of that abstraction. Since the point-evaluation interpolation operation is differentiable and has an adjoint, the arbitrary mesh-to-mesh dual evaluation interpolations, and model coupling, do too.

This, other than perhaps model coupling which works as a result of the Firedrake implementation, is applicable generally in the finite element method framework. One can implement vertex-only meshes and interpolations onto them as point-evaluations in another library and achieve the same results. So long as one represents dual bases as sums of point evaluations, one can extend this work to arbitrary function spaces on other meshes, as has been implemented here.

On the implementation side, the new abstractions in Firedrake are entirely parallel safe, as discussed in chapters 7 and 8. Elsewhere, I have been involved in adding dual spaces to UFL and Firedrake (chapter 2). This improves the quality of the abstraction that is provided, making it closer to the mathematical objects we aim to represent. The element local dual evaluation routines have also been improved, as described in chapter 3, to make better use of the finite element abstraction provided. By using the correct abstraction there, one can now compose finite elements from pre-defined ones, and have the dual evaluation operations efficiently compose as well. Lastly, the work to make dual-evaluation interpolation into a symbolic operation will, when fully completed, allow new systems of equations to be solved.

10.2 Future work

In each chapter, future work that could add to what has been done has been noted, so the reader is therefore referred to each such section. Here some more general suggestions are made.

10.2.1 More Applications

The demonstrations here are in no way an exhaustive list of what this work could be used for.

A sensitivity analysis demonstration using `firedrake.adjoint` would also be a good addition, for example to investigate the sensitivity of the unknown conductivity measurements (Sect. 6.2) to the measured data.

There are other opportunities to, relatively straightforwardly, implement real modelling applications using this work. Nixon-Hill et al. [3] already contains a textbook groundwater hydrology inverse problem using point data, alongside a glaciological data assimilation example. This assimilates the MEaSUREs InSAR phase-based velocity map [62] into a model of the Larcen C ice shelf in Antarctica. The model is built in Icepack [109] which is built on top of Firedrake and so has access to all its features. Further data assimilation could be done using other packages built on top of Firedrake: atmospheric flow measurements could be assimilated into the dynamical core Gusto [102] and various ocean measurements into the ocean model Thetis [110]. I am aware of work that would use point data in seismic inversion problems built on top of Firedrake, though none of these have yet been published.

To bring all this work together, one could come up with a multiphysics simulation which includes data assimilation and model coupling, perhaps even using data from a model produced by another software package. Firedrake could then operate so-called ‘digital twin’ simulations [111] where real-life objects or systems are simulated with multiple sets of coupled equations. These are updated as data becomes available, allowing one to, for example, know the stresses on an aircraft wing as a result of wind speed measurements and internal deflection measurements.

10.2.2 Moving Points

As was highlighted when looking at existing capability in Sect. 7.3, moving particles are widely used and would be a significant addition to Firedrake. The generality of the abstractions of vertex-only meshes, functions on them, and associated differentiable interpolation operations are very powerful. Movable vertex-only meshes would, partially, allow for assimilation of moving point data (for example, assimilating measurements from ocean floats into an ocean model or balloon trajectories into an atmospheric model) and other moving data (see cross-mesh interpolation, Chapter 8). A general approach, from a Bayesian point of view, is outlined by Apte, Jones, and Stuart [112] (see also Cotter, Dashti, and Stuart [113]). The abstraction is a key difference between other libraries which implement moving points such as Deal.II's particles: the point evaluation operator is differentiable without having to resort to cumbersome external automatic differentiation tools.

Moving points allow tracer methods, where the motion of the vertices (here, Lagrangian particles) is determined by a velocity vector field on the underlying mesh. In so-called active tracer methods, the movement of the particles then influences the equations being solved. These are used for computational fluid simulations in areas ranging from biomedicine [114, 115] to environmental fluid mechanics [116, 117, 118, 119, 120]. Indeed, any method which involves moving particles and meshes, such as particle-in-cell methods, would become available.

In the simplest sense, one could implement moving particles by producing a new vertex-only mesh at each time step. The problem with this is that the two sets of particles have no link to one another so there's no obvious way to move data from one to the other. One could somehow engineer them having the same global index but that would be cumbersome. Nevertheless in a hypothetical scenario where this was the approach taken, one would create a new function on the new vertex-only mesh and create an operation that moved data from one to the other via some PETSc star forest operation.

Since the issue is point identification within a vertex-only mesh, one could create a wrapper class which creates a single vertex-only mesh for each particle. Point identification and the assignment of values would then be devolved to the wrapper which can make use of Firedrake's interpolation as point evaluation functionality.

Alternatively, assuming one is happy to accept the cost of re-embedding all particles each time they are moved, one could have the wrapper class control the input coordinates to a vertex-only mesh Ω_v which the input-ordering vertex-only mesh $\Omega_v^{\text{input-ordering}}$ is guaranteed to respect. This would aid solution setting and transfer: values would be members of a function on the input-ordering vertex-only mesh, which one would interpolate onto the usual vertex-only mesh for interactions with the parent mesh. To move coordinates, the wrapper class coordinates list would be updated and a new vertex-only mesh Δ_v created with input-ordering $\Delta_v^{\text{input-ordering}}$. Solution transfer would require interpolation from Ω_v onto $\Omega_v^{\text{input-ordering}}$, changes in values as necessary to match the change in coordinates, then assignment to a function on $\Delta_v^{\text{input-ordering}}$

which would then be interpolated onto a function on Δ_v.

Ideally we would maintain a single vertex-only mesh and move the vertices: any functions defined on the mesh would move with it (as currently happens with other meshes that have their coordinates function changed). If this movement was performed by modifying the coordinates function, we would need to recalculate the embedding in the parent mesh. That means (1.) finding the parent mesh cells and (2.) finding the coordinates of the point within the cell. We can speed up step (1.) by initially assuming that no particles have moved, checking that is the case, then only relocating those that have: this is the general approach outlined by Martin, Loth, and Lankford [121] and would be a good first step.

Where we have information about particle trajectory, for example an assigned velocity at a given time step, we gain access to techniques which track the trajectory of particles. These avoid the high computational cost of resorting all particles [88] since one can simply work out whether a particle will move outside of a cell or mesh partition boundary. Many algorithms are available [122, 87, 123] and they could be combined with the 'nearest cell' approach to avoid known problems with floating point imprecision [88]. The challenge will be to maintain generality: how do we tell a moving vertex-only mesh that we have velocity information? At this point we might again consider wrapping up our functionality in something like the `particles` classes found in deal.II and LEoPart. Ideally we would avoid the design limitations these have.[1] Before committing to an approach, significant consideration of the correct design should be undertaken.

Once we have very large numbers of particles involved in computations, load balancing may become an issue. Gassmöller et al. [89] make relevant suggestions for dealing with this including repartitioning the parent mesh, adjusting particle density (if the application allowed) and adjusting the parent mesh to accommodate the particle density (again, if the application allowed).

To assimilate data from moving particles, one would need to consider all the places where the misfit functional to minimise is affected by the change in coordinates: the operation that changes those coordinates would need to be differentiable to minimise with variational assimilation methods. For Firedrake and `firedrake.adjoint` this would mean ensuring that such operations are taped for adjoint generation.

Tracing and data assimilation could be tested in the ocean model Thetis [110] and the atmospheric model Gusto [102], both of which have been built using Firedrake. Tracers would be useful for studying the movement of pollutants or other substances in the atmosphere or ocean, whilst moving point data assimilation could be used to produce forecasts.

[1]Deal.II's particle implementation, for example, is not suitable for points which move large distances. Those which move outside cell halos are deleted. To find them again, one has to manually request that they be re-located [89].

10.2.3 Moving Mesh to Mesh Interpolation

Once vertex-only mesh vertices are movable, we ought to be able to interpolate between moving meshes without having to recreate the interpolation object each time. No finite element library, to the author's knowledge, has such capability. With a well thought-out API, we ought to be able to achieve efficient interpolation from lines and planes that move with the flow of a vector field, data assimilation from moving lines planes or volumes, and coupling between problems with moving meshes.

10.2.4 Alternative Implementation

To prove the generality of the abstractions that have been identified, it would be useful to implement them in another library. The most obvious candidate is FEniCSx, the rewrite of FEniCS [16, 17] which uses UFL and maintains a similar level of abstraction to Firedrake. The first step would be to check if point cells, and a P0DG finite element on it, can be instatiated with Basix [39], which FEniCSx uses for finite element tabulation.

NGSolve [22] is another candidate for straightforward implementation since it also maintains a high level of abstraction and has a UFL-like API, though recent work has coupled its mesh generation capabilities to Firedrake making this a less attractive proposition. Libraries which do not maintain the high level of abstraction found in Firedrake and FEniCS ought to be able to implement the same functionality but could require more work.

Of course no other library (exluding legacy FEniCS) currently supports automated adjoint generation with dolfin-adjoint/pyadjoint [46], so their are fewer use cases. Nevertheless, any library that supports adjoint dual-evaluation interpolation would allow for point sources and sinks, as demonstrated here, and cross-mesh interpolation could be implemented in a similar way to that outlined in chapter 8.

10.2.5 Querying Geoscientific Model Outputs

Motivation: Data Output from Geoscientific Models

Geoscientific models, particularly those of the earth's climate, produce very large, unwieldy data sets. When more supercomputing resources become available, models tend to become more precise and run for longer, producing bigger and bigger data sets. The Coupled Model Intercomparison Project (CMIP) data sets provide a good example: these have increased exponentially in size with each report [124] with the sixth phase (CMIP6) producing between 20 and 40 Petabytes of data [125]. So-called 'big-data' does not get much bigger than this. Different models may be simulating the same phenomenon but typically will use one or more different numerical and discretisation methods (finite volume, finite element, spectral, finite difference and more) to reach results, any of which will produce data in different formats with different properties.

Querying geoscientific models is therefore not a straightforward or unified process. Bespoke code may have to be written, a time-consuming and easily error-prone endeavour, in order to investigate some metric of interest: such code may not be referenced or, if it is, may not give reproducible results without reimplementation. Data is often copied from a data center instead of using the powerful High Performance Computing (HPC) systems which run the models and are typically nearest to the storage data centres containing the model outputs. As well as being inefficient, community commentators [126] [127] have pointed out that such data movement has associated energy and environmental costs which ought not be neglected. When comparing some particular statistic across models, especially where such statistics are complicatedly defined, these challenges are almost guaranteed. Statistic comparison is often done to quantify uncertainty in headline results (as is done in CMIP). Statistic comparison is also valuable when developing an understanding of complex phenomena which are known to occur in nature, have measurements limited due to their temporal or spacial extent, but which emerge in models such as climate variability and certain ocean currents (see, for example, Medhaug and Furevik [128]).

Efforts to address the difficulties of model statistic querying are underway. The UK's JASMIN HPC cluster [129] and the US's National Center for Atmospheric Research (NCAR) CMIP Analysis Platform [130] provide dedicated resources for data analysis of climate and earth system model data. Common file formats such as NetCDF [131] exist and tools such as NetCDF Operators [132] and the NCAR Command Language (NCL) [133] provide serialised tools for their analysis. Integration of NetCDF with traditional data science parsing tools have also been attempted [124].

Perhaps the most exciting recent development has been Pangeo [134] which provides a stack for data analysis locally, in the cloud, or on an HPC cluster. The user interface, data model, computing system, data storage system and resource management system are all separate interchangeable modules. For more on Pangeo, Odaka et. al. [135] provide a good introduction.

HPC and cloud based approaches to model output data analysis using tools like Pangeo is likely to grow in future with uptake actively encouraged in the community [136] [137] [135]. Indeed, data analysis in the Intergovernmental Panel on Climate Change (IPCC) Sixth Assessment Report (AR6) was meant to have taken such approaches [138].

Solution: An 'In Situ' Domain Specific Language for Querying

Available statistics in tools such as Pangeo are necessarily limited to those which have been pre-programmed into the software. Instead, a Domain Specific Language (DSL) for querying geoscientific models for statistics derived from their saved field data, with generated code running on the HPC systems closest to the data (hence *'in situ'*), is proposed.

The DSL should contain abstractions of the mathematical concepts needed to describe a particular statistic and compile to produce optimised, scalable, and, crucially, highly parallelised code (i.e. using multiple computing 'cores' or 'nodes' distributed across an HPC system) for finding it. Should another model need investigating then the same query, written in the DSL, could be compiled and run for that model. Like UFL, introduced in chapter 1, and the tensor algebra DSL GEM, introduced in chapter 3, the new DSL should be embedded in another language such as Python to aid development and allow straightforward interfacing with existing tools. The language and the tools for parsing and compiling it should be provably correct, freely and openly available and implement a straightforward referencing procedure so that queries can be reproduced.

This DSL may have other uses: for example, complex geoscientific models typically do not output all their field data at every time step due to the quantity of data produced. As well as providing a tool for querying models, the language could perhaps be used to specify a desired statistic prior to the running of a model such that it would be output at each time step.

The work described here provides an ideal platform for the development of this DSL. UFL can now be used to describe point evaluations (chapter 4) as well as line, surface and volume integrals. One merely needs to read in the external data (in parallel) then use cross-mesh interpolation operations (chapter 8) to compute the query.

Generic external data input in parallel was the topic of Sect. 8.4.2. All that is left to do to is to introduce tools that can read the stored NetCFD data in parallel and to deal with the missing mesh-building infrastructure, as raised in the future work section of chapter 8 (Sect. 8.5). In particular we require a tool for building a mesh at the coordinates of data alongside a tool for building meshes from a query of, for example, a surface integral.

Firedrake was built from the outset to run in parallel on HPC system since the kinds of problems it is designed to solve (big PDE systems including ocean and weather models) often require the enormous computing power an HPC gives you. Firedrake also contains the infrastructure to enable full reproducibility and referencing of the exact code used to reach a solution which can be adapted for use here.

Models written in Firedrake can be used as initial test beds. In particular, there is the glaciological modelling toolkit Icepack [109], the atmospheric model Gusto [102] and the ocean model Thetis [110].

The completed software should allow any query expressible by the new DSL to be found for any grid or unstructured mesh based field data set. The design of the language will need therefore to avoid using the language of specific application areas or particular numerical methods to ensure its wide use. Work currently being done on UFL is removing most finite-element specific terminology (elements, for example, will no longer be defined in UFL), nevertheless to gain traction, it might be necessary to write a layer on top of Firedrake to obfuscate some of the inner workings.

The earth sciences aren't the only area where very large quantities of gridded field data are produced. Pangeo, for example, has found applications in astrophysics [139]. The impact of this work could therefore be wide-ranging.

If the DSL is sufficiently self contained then it may be desirable to integrate with Pangeo.

Below, Pangeo's architecture is explored to get a sense of how this might be done. Pangeo has separable pieces which are supposed to be interchangeable[2]. Working from the bottom up, the 'foundation' is the Python programming language, which suits Firedrake. There are 3 supported 'compute platforms' - HPC, Amazon Web Services (AWS) and Google Cloud Platform and, above this 3 'processing modes' which are interactive Jupyter sessions, batch runs of scripts and the 'serverless computing' cloud execution model. Firedrake merely needs a Python virtual environment in which to run so is agnostic to these. Next there are two 'N-D Arrays' pieces: NumPy and DASK [140]. At this point things start to diverge: under the hood, Firedrake is an increasingly thin layer around PETSc [10, 9]. Firedrake uses PETSc vectors (`Vecs`) and matrices (`Mats`) for data storage alongside numpy arrays. DASK is a parallel computing tool that distributes serial tools such as numpy: whilst it's possible that this could be used alongside Firedrake, it is not directly compatible with it since Firedrake is already parallelised. Firedrake would therefore be a new third column at this point and would not be easilly hot-swappable. At the highest level there are three 'data models' for high level data analysis: xarray [141], Iris [142] and pandas [143]. Firedrake also provides a 'data model' in the form of Firedrake `Functions` which represent fields and and meshes which represent the domains that those fields are defined over; data analysis involves manipulating these with UFL and assembly operations.

[2]see `https://pangeo.io/architecture.html#interoperability-in-pangeo`

Bibliography

[1] Colin J. Cotter and David A. Ham. *Imperial College London Mathematics module MATH96063/MATH97017/MATH97095 Finite Elements: Numerical Analysis and Implementation*. Lecture Series. Imperial College London, 2020. URL: https://finite-element.github.io/ (visited on 05/26/2020).

[2] V. V. Davydov. *Types of Generalization in Instruction: Logical and Psychological Problems in the Structuring of School Curricula. Soviet Studies in Mathematics Education. Volume 2*. en. Tech. rep. ISBN: 9780873532914 ERIC Number: ED318633. National Council of Teachers of Mathematics, 1906 Association Dr, 1990. (Visited on 11/21/2023).

[3] Reuben W. Nixon-Hill et al. *Consistent Point Data Assimilation in Firedrake and Icepack*. arXiv:2304.06058 [cs, math]. Aug. 2023. URL: http://arxiv.org/abs/2304.06058 (visited on 10/30/2023).

[4] Florian Rathgeber et al. "Firedrake: automating the finite element method by composing abstractions". In: *ACM Transactions on Mathematical Software* 43.3 (Dec. 2016). arXiv: 1501.01809, pp. 1–27. ISSN: 00983500. DOI: 10.1145/2998441. URL: http://arxiv.org/abs/1501.01809 (visited on 02/04/2020).

[5] David A. Ham et al. *Firedrake User Manual*. First edition. Imperial College London et al., May 2023. DOI: 10.25561/104839.

[6] Martin Sandve Alnæs. "UFL: a finite element form language". en. In: *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book*. Ed. by Anders Logg, Kent-Andre Mardal, and Garth Wells. Lecture Notes in Computational Science and Engineering. Berlin, Heidelberg: Springer, 2012, pp. 303–338. ISBN: 978-3-642-23099-8. DOI: 10.1007/978-3-642-23099-8_17. URL: https://doi.org/10.1007/978-3-642-23099-8_17 (visited on 02/04/2020).

[7] Marjan Mernik, Jan Heering, and Anthony M. Sloane. "When and how to develop domain-specific languages". In: *ACM Computing Surveys* 37.4 (Dec. 2005), pp. 316–344. ISSN: 0360-0300. DOI: 10.1145/1118890.1118892. URL: https://dl.acm.org/doi/10.1145/1118890.1118892 (visited on 11/20/2023).

[8] Miklós Homolya et al. "TSFC: a structure-preserving form compiler". In: *SIAM Journal on Scientific Computing* 40.3 (Jan. 2018). arXiv: 1705.03667, pp. C401–C428. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/17M1130642. URL: http://arxiv.org/abs/1705.03667 (visited on 03/04/2020).

[9] Satish Balay et al. *PETSc Web page*. 2022. URL: https://petsc.org/.

[10] Satish Balay et al. *PETSc/TAO Users Manual*. Tech. rep. ANL-21/39 - Revision 3.18. Argonne National Laboratory, 2022.

[11] Message Passing Interface Forum. *MPI: A Message-Passing Interface Standard Version 4.0*. June 2021. URL: https://www.mpi-forum.org/docs/mpi-4.0/mpi40-report.pdf.

[12] William Gropp, Ewing Lusk, and Anthony Skjellum. *Using MPI: Portable Parallel Programming with the Message-Passing Interface*. The MIT Press, 2014. ISBN: 978-0-262-52739-2.

[13] Robert C. Kirby. "Algorithm 839: FIAT, a new paradigm for computing finite element basis functions". In: *ACM Transactions on Mathematical Software* 30.4 (Dec. 2004), pp. 502–516. ISSN: 0098-3500. DOI: 10.1145/1039813.1039820. URL: https://doi.org/10.1145/1039813.1039820 (visited on 02/26/2020).

[14] Miklós Homolya, Robert C. Kirby, and David A. Ham. "Exposing and exploiting structure: optimal code generation for high-order finite element methods". In: *arXiv:1711.02473 [cs]* (Nov. 2017). arXiv: 1711.02473. URL: http://arxiv.org/abs/1711.02473 (visited on 04/06/2020).

[15] Florian Rathgeber et al. "PyOP2: A High-Level Framework for Performance-Portable Simulations on Unstructured Meshes". In: *2012 SC Companion: High Performance Computing, Networking Storage and Analysis*. ISSN: null. Nov. 2012, pp. 1116–1123. DOI: 10.1109/SC.Companion.2012.134.

[16] Anders Logg. "Efficient Representation of Computational Meshes". In: *International Journal of Computational Science and Engineering* 4.4 (2009). arXiv: 1205.3081, p. 283. ISSN: 1742-7185, 1742-7193. DOI: 10.1504/IJCSE.2009.029164. URL: http://arxiv.org/abs/1205.3081 (visited on 03/04/2020).

[17] Martin Alnæs et al. "The FEniCS Project Version 1.5". en. In: *Archive of Numerical Software* 3.100 (Dec. 2015). Number: 100. ISSN: 2197-8263. DOI: 10.11588/ans.2015.100.20553. URL: https://journals.ub.uni-heidelberg.de/index.php/ans/article/view/20553 (visited on 07/30/2020).

[18] Andreas Dedner et al. "A generic interface for parallel and adaptive discretization schemes: abstraction principles and the Dune-Fem module". en. In: *Computing* 90.3 (Nov. 2010), pp. 165–196. ISSN: 1436-5057. DOI: 10.1007/s00607-010-0110-3. URL: https://doi.org/10.1007/s00607-010-0110-3 (visited on 03/23/2023).

[19] Mathias Louboutin et al. “Devito (v3.1.0): an embedded domain-specific language for finite differences and geophysical exploration”. English. In: *Geoscientific Model Development* 12.3 (Mar. 2019). Publisher: Copernicus GmbH, pp. 1165–1187. ISSN: 1991-959X. DOI: 10.5194/gmd-12-1165-2019. URL: https://gmd.copernicus.org/articles/12/1165/2019/ (visited on 11/21/2023).

[20] Eric Heisler, Aadesh Deshmukh, and Hari Sundar. “Finch: Domain Specific Language and Code Generation for Finite Element and Finite Volume in Julia”. en. In: *Computational Science – ICCS 2022*. Ed. by Derek Groen et al. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 118–132. ISBN: 978-3-031-08751-6. DOI: 10.1007/978-3-031-08751-6_9.

[21] F. Hecht. “New development in FreeFem++”. In: *Journal of Numerical Mathematics* 20.3-4 (2012), pp. 251–265. ISSN: 1570-2820. URL: https://freefem.org/.

[22] Joachim Schöberl. “C++ 11 implementation of finite elements in NGSolve”. In: *Institute for analysis and scientific computing, Vienna University of Technology* 30 (2014). Publisher: Citeseer.

[23] CSC – IT CENTER FOR SCIENCE LTD. *Elmer*. Nov. 2020. URL: https://www.csc.fi/web/elmer/.

[24] Peter Randall Schunk et al. “GOMA 6.0 - A Full-Newton Finite Element Program for Free and Moving Boundary Problems with Coupled Fluid/ Solid Momentum, Energy, Mass, and Chemical Species Transport: User’s Guide”. In: (July 2013). DOI: 10.2172/1089869. URL: https://www.osti.gov/biblio/1089869.

[25] Cody J. Permann et al. “MOOSE: Enabling massively parallel multiphysics simulation”. In: *SoftwareX* 11 (Jan. 2020), p. 100430. ISSN: 2352-7110. DOI: 10.1016/j.softx.2020.100430. URL: https://www.sciencedirect.com/science/article/pii/S2352711019302973 (visited on 11/21/2023).

[26] *Abaqus - Mechanical and Civil Engineering Simulation*. en. URL: https://www.3ds.com/products-services/simulia/products/abaqus/ (visited on 11/21/2023).

[27] *COMSOL: Multiphysics Software for Optimizing Designs*. en. URL: https://www.comsol.com/ (visited on 11/21/2023).

[28] *Ansys — Engineering Simulation Software*. en-US. URL: https://www.ansys.com/ (visited on 11/21/2023).

[29] T. Schwedes et al. *Mesh dependence in PDE-constrained optimisation an application in tidal turbine array layouts*. Accepted: 2017-11-06T14:18:09Z. Springer, Aug. 2017. ISBN: 978-3-319-59483-5. URL: http://spiral.imperial.ac.uk/handle/10044/1/53084 (visited on 06/22/2020).

[30] David A. Ham. “UFL Dual Spaces, a proposal”. In: *arXiv:2101.05158 [cs]* (Jan. 2021). arXiv: 2101.05158. URL: http://arxiv.org/abs/2101.05158 (visited on 03/18/2021).

[31] R. Anderson et al. “MFEM: A Modular Finite Element Methods Library”. In: *Computers & Mathematics with Applications* 81 (2021), pp. 42–74. DOI: 10.1016/j.camwa.2020.06.009.

[32] *MFEM: Modular Finite Element Methods [Software].* Published: mfem.org. DOI: 10.11578/dc.20171025.1248.

[33] Daniel Arndt et al. “The \texttdeal.II Library, Version 9.5”. In: *Journal of Numerical Mathematics* 31.3 (2023), pp. 231–246. DOI: 10.1515/jnma-2023-0089. URL: https://dealii.org/deal95-preprint.pdf.

[34] Philippe G. Ciarlet. *The Finite Element Method for Elliptic Problems.* Classics in Applied Mathematics. Society for Industrial and Applied Mathematics, Jan. 2002. ISBN: 978-0-89871-514-9. DOI: 10.1137/1.9780898719208. URL: https://epubs.siam.org/doi/book/10.1137/1.9780898719208 (visited on 02/25/2020).

[35] Susanne C. Brenner and L. Ridgway Scott. *The Mathematical Theory of Finite Element Methods.* Ed. by J. E. Marsden, L. Sirovich, and S. S. Antman. Vol. 15. Texts in Applied Mathematics. New York, NY: Springer, 2008. ISBN: 978-0-387-75933-3 978-0-387-75934-0. DOI: 10.1007/978-0-387-75934-0. URL: http://link.springer.com/10.1007/978-0-387-75934-0 (visited on 11/09/2022).

[36] J. R. Maddison and P. E. Farrell. “Directional integration on unstructured meshes via supermesh construction”. en. In: *Journal of Computational Physics* 231.12 (June 2012), pp. 4422–4432. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2012.02.009. URL: https://www.sciencedirect.com/science/article/pii/S0021999112000885 (visited on 03/21/2023).

[37] Robert C. Kirby. “FIAT: numerical construction of finite element basis functions”. en. In: *Automated Solution of Differential Equations by the Finite Element Method: The FEniCS Book.* Ed. by Anders Logg, Kent-Andre Mardal, and Garth Wells. Lecture Notes in Computational Science and Engineering. Berlin, Heidelberg: Springer, 2012, pp. 247–255. ISBN: 978-3-642-23099-8. DOI: 10.1007/978-3-642-23099-8_13. URL: https://doi.org/10.1007/978-3-642-23099-8_13 (visited on 02/04/2020).

[38] Matthew W. Scroggs. “Symfem: a symbolic finite element definition library”. en. In: *Journal of Open Source Software* 6.64 (Aug. 2021), p. 3556. ISSN: 2475-9066. DOI: 10.21105/joss.03556. URL: https://joss.theoj.org/papers/10.21105/joss.03556 (visited on 10/28/2023).

[39] Matthew W. Scroggs et al. "Basix: a runtime finite element basis evaluation library". en. In: *Journal of Open Source Software* 7.73 (May 2022), p. 3982. ISSN: 2475-9066. DOI: 10.21105/joss.03982. URL: https://joss.theoj.org/papers/10.21105/joss.03982 (visited on 10/28/2023).

[40] Oliver Sander. "Local Finite Elements and the dune-localfunctions Module". en. In: *DUNE — The Distributed and Unified Numerics Environment*. Ed. by Oliver Sander. Lecture Notes in Computational Science and Engineering. Cham: Springer International Publishing, 2020, pp. 301–324. ISBN: 978-3-030-59702-3. DOI: 10.1007/978-3-030-59702-3_8. URL: https://doi.org/10.1007/978-3-030-59702-3_8 (visited on 10/28/2023).

[41] Peter Bastian, Felix Heimann, and Sven Marnach. "Generic implementation of finite element methods in the Distributed and Unified Numerics Environment (DUNE)". eng. In: *Kybernetika* 46.2 (2010). Publisher: Institute of Information Theory and Automation AS CR, pp. 294–315. ISSN: 0023-5954. URL: https://dml.cz/handle/10338.dmlcz/140745 (visited on 10/28/2023).

[42] Christian Engwer et al. *Function space bases in the dune-functions module*. arXiv:1806.09545 [cs]. June 2018. DOI: 10.48550/arXiv.1806.09545. URL: http://arxiv.org/abs/1806.09545 (visited on 10/28/2023).

[43] Steven A Orszag. "Spectral methods for problems in complex geometries". en. In: *Journal of Computational Physics* 37.1 (Aug. 1980), pp. 70–92. ISSN: 0021-9991. DOI: 10.1016/0021-9991(80)90005-4. URL: https://www.sciencedirect.com/science/article/pii/0021999180900054 (visited on 09/24/2021).

[44] Charles R. Harris et al. "Array programming with NumPy". In: *Nature* 585.7825 (Sept. 2020). Publisher: Springer Science and Business Media LLC, pp. 357–362. DOI: 10.1038/s41586-020-2649-2. URL: https://doi.org/10.1038/s41586-020-2649-2.

[45] Andrew T. T. McRae et al. "Automated generation and symbolic manipulation of tensor product finite elements". In: *SIAM Journal on Scientific Computing* 38.5 (Jan. 2016). arXiv: 1411.2940, S25–S47. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/15M1021167. URL: http://arxiv.org/abs/1411.2940 (visited on 08/26/2020).

[46] Sebastian K. Mitusch, Simon W. Funke, and Jørgen S. Dokken. "dolfin-adjoint 2018.1: automated adjoints for FEniCS and Firedrake". en. In: *Journal of Open Source Software* 4.38 (June 2019), p. 1292. ISSN: 2475-9066. DOI: 10.21105/joss.01292. URL: https://joss.theoj.org/papers/10.21105/joss.01292 (visited on 06/22/2020).

[47] Michael Lange et al. "Efficient mesh management in Firedrake using PETSc-DMPlex". In: *SIAM Journal on Scientific Computing* 38.5 (Jan. 2016). arXiv: 1506.07749, S143–S155. ISSN: 1064-8275, 1095-7197. DOI: 10.1137/15M1026092. URL: http://arxiv.org/abs/1506.07749 (visited on 02/07/2020).

[48] Gheorghe-Teodor Bercea et al. “A structure-exploiting numbering algorithm for finite elements on extruded meshes, and its performance evaluation in Firedrake”. English. In: *Geoscientific Model Development* 9.10 (Oct. 2016). Publisher: Copernicus GmbH, pp. 3803–3815. ISSN: 1991-959X. DOI: 10.5194/gmd-9-3803-2016. URL: https://gmd.copernicus.org/articles/9/3803/2016/ (visited on 01/25/2023).

[49] R. E. Wengert. “A simple automatic derivative evaluation program”. en. In: *Communications of the ACM* 7.8 (Aug. 1964), pp. 463–464. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/355586.364791. URL: https://dl.acm.org/doi/10.1145/355586.364791 (visited on 07/29/2022).

[50] Atilim Gunes Baydin et al. *Automatic differentiation in machine learning: a survey.* arXiv:1502.05767 [cs, stat]. Feb. 2018. DOI: 10.48550/arXiv.1502.05767. URL: http://arxiv.org/abs/1502.05767 (visited on 08/04/2022).

[51] Uwe Naumann. *The Art of Differentiating Computer Programs.* Software, Environments and Tools. Society for Industrial and Applied Mathematics, Jan. 2011. ISBN: 978-1-61197-206-1. DOI: 10.1137/1.9781611972078. URL: https://epubs.siam.org/doi/book/10.1137/1.9781611972078 (visited on 02/19/2021).

[52] James Bradbury et al. *JAX: composable transformations of Python+NumPy programs.* 2018. URL: http://github.com/google/jax.

[53] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems.* 2015. URL: https://www.tensorflow.org/.

[54] Adam Paszke et al. “Automatic differentiation in PyTorch”. en. In: (Oct. 2017). URL: https://openreview.net/forum?id=BJJsrmfCZ (visited on 09/27/2022).

[55] Bruce Christianson. “A Leibniz Notation for Automatic Differentiation”. In: *Recent Advances in Algorithmic Differentiation.* Ed. by Shaun Forth et al. Vol. 87. Lecture Notes in Computational Science and Engineering. ISSN: 1439-7358. Berlin: Springer, 2012, pp. 1–9. ISBN: 978-3-540-68935-5. DOI: 10.1007/978-3-642-30023-3_1.

[56] Andreas Griewank and Andrea Walther. *Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation.* Second. _eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9780898717761. Society for Industrial and Applied Mathematics, 2008. DOI: 10.1137/1.9780898717761. URL: https://epubs.siam.org/doi/abs/10.1137/1.9780898717761.

[57] Bruce Christianson. “Automatic Hessians by reverse accumulation”. In: *IMA Journal of Numerical Analysis* 12.2 (Apr. 1992), pp. 135–150. ISSN: 0272-4979. DOI: 10.1093/imanum/12.2.135. URL: https://doi.org/10.1093/imanum/12.2.135 (visited on 09/15/2022).

[58] Barak A. Pearlmutter. “Fast Exact Multiplication by the Hessian”. In: *Neural Computation* 6.1 (Jan. 1994), pp. 147–160. ISSN: 0899-7667. DOI: 10.1162/neco.1994.6.1.147. URL: https://doi.org/10.1162/neco.1994.6.1.147 (visited on 09/01/2022).

[59] P. E. Farrell et al. “Automated Derivation of the Adjoint of High-Level Transient Finite Element Programs”. In: *SIAM Journal on Scientific Computing* 35.4 (Jan. 2013). Publisher: Society for Industrial and Applied Mathematics, pp. C369–C393. ISSN: 1064-8275. DOI: 10.1137/120873558. URL: https://epubs.siam.org/doi/10.1137/120873558 (visited on 12/15/2020).

[60] Jan Hückelheim et al. *Understanding Automatic Differentiation Pitfalls.* arXiv:2305.07546 [cs, math]. May 2023. DOI: 10.48550/arXiv.2305.07546. URL: http://arxiv.org/abs/2305.07546 (visited on 05/15/2023).

[61] Mathieu Morlighem et al. “Deep glacial troughs and stabilizing ridges unveiled beneath the margins of the Antarctic ice sheet”. In: *Nature Geoscience* 13.2 (2020). Publisher: Nature Publishing Group UK London, pp. 132–137.

[62] J Mouginot, E Rignot, and B Scheuchl. “Continent-wide, interferometric SAR phase, mapping of Antarctic ice velocity”. In: *Geophysical Research Letters* 46.16 (2019). Publisher: Wiley Online Library, pp. 9710–9718.

[63] Douglas R. MacAyeal. “The basal stress distribution of Ice Stream E, Antarctica, inferred by control methods”. en. In: *Journal of Geophysical Research: Solid Earth* 97.B1 (1992). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/91JB02454, pp. 595–603. ISSN: 2156-2202. DOI: 10.1029/91JB02454. URL: https://onlinelibrary.wiley.com/doi/abs/10.1029/91JB02454 (visited on 01/24/2023).

[64] Ian Joughin, Douglas R. MacAyeal, and Slawek Tulaczyk. “Basal shear stress of the Ross ice streams from control method inversions”. en. In: *Journal of Geophysical Research: Solid Earth* 109.B9 (2004). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/2003JB002960. ISSN: 2156-2202. DOI: 10.1029/2003JB002960. URL: https://onlinelibrary.wiley.com/doi/abs/10.1029/2003JB002960 (visited on 01/24/2023).

[65] Andreas Vieli et al. “Numerical modelling and data assimilation of the Larsen B ice shelf, Antarctic Peninsula”. In: *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 364.1844 (May 2006). Publisher: Royal Society, pp. 1815–1839. DOI: 10.1098/rsta.2006.1800. URL: https://royalsocietypublishing.org/doi/10.1098/rsta.2006.1800 (visited on 01/24/2023).

[66] Daniel R Shapero et al. “Basal resistance for three of the largest Greenland outlet glaciers”. In: *Journal of Geophysical Research: Earth Surface* 121.1 (2016). Publisher: Wiley Online Library, pp. 168–180.

[67] Pauli Virtanen et al. “SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python”. In: *Nature Methods* 17 (2020), pp. 261–272. DOI: 10.1038/s41592-019-0686-2.

[68] Rudolph Emil Kalman. “A New Approach to Linear Filtering and Prediction Problems”. In: *Transactions of the ASME–Journal of Basic Engineering* 82.Series D (1960), pp. 35–45.

[69] S.J. Julier, J.K. Uhlmann, and H.F. Durrant-Whyte. “A new approach for filtering nonlinear systems”. In: *Proceedings of 1995 American Control Conference - ACC’95*. Vol. 3. June 1995, 1628–1632 vol.3. DOI: 10.1109/ACC.1995.529783. URL: https://ieeexplore.ieee.org/abstract/document/529783?casa_token=LQHB45niHncAAAAA:j120uzZFEs620vBlgTrG-zEaR2W4PQ-ctPUwqNlPe-4u408SCzgVFBkeEh-MiJdPfFCpmy6ZFg (visited on 11/15/2023).

[70] Michael Roth, Gustaf Hendeby, and Fredrik Gustafsson. “Nonlinear Kalman Filters Explained: A Tutorial on Moment Computations and Sigma Point Methods”. In: *Journal of Advances in Information Fusion* 11 (June 2016).

[71] Michael Roth et al. “The Ensemble Kalman filter: a signal processing perspective”. In: *EURASIP Journal on Advances in Signal Processing* 2017.1 (Aug. 2017), p. 56. ISSN: 1687-6180. DOI: 10.1186/s13634-017-0492-x. URL: https://doi.org/10.1186/s13634-017-0492-x (visited on 11/15/2023).

[72] Geir Evensen. “Sequential data assimilation with a nonlinear quasi-geostrophic model using Monte Carlo methods to forecast error statistics”. en. In: *Journal of Geophysical Research: Oceans* 99.C5 (1994). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/94JC00572, pp. 10143–10162. ISSN: 2156-2202. DOI: 10.1029/94JC00572. URL: https://onlinelibrary.wiley.com/doi/abs/10.1029/94JC00572 (visited on 11/15/2023).

[73] N. J. Gordon, D. J. Salmond, and A. F. M. Smith. “Novel approach to nonlinear/non-Gaussian Bayesian state estimation”. en. In: *IEE Proceedings F (Radar and Signal Processing)* 140.2 (Apr. 1993). Publisher: IET Digital Library, pp. 107–113. ISSN: 2053-9045. DOI: 10.1049/ip-f-2.1993.0015. URL: https://digital-library.theiet.org/content/journals/10.1049/ip-f-2.1993.0015 (visited on 11/15/2023).

[74] Fredrik Gustafsson. “Particle filter theory and practice with positioning applications”. In: *IEEE Aerospace and Electronic Systems Magazine* 25.7 (July 2010). Conference Name: IEEE Aerospace and Electronic Systems Magazine, pp. 53–82. ISSN: 1557-959X. DOI: 10.1109/MAES.2010.5546308. URL: https://ieeexplore.ieee.org/document/5546308 (visited on 11/15/2023).

[75] Mark Buehner, Ron McTaggart-Cowan, and Sylvain Heilliette. “An Ensemble Kalman Filter for Numerical Weather Prediction Based on Variational Data Assimilation: VarEnKF”. EN. In: *Monthly Weather Review* 145.2 (Feb. 2017). Publisher: American Meteorological Society Section: Monthly Weather Review, pp. 617–635. ISSN: 1520-0493, 0027-0644. DOI: 10.1175/MWR-D-16-0106.1. URL: https://journals.ametsoc.org/view/journals/mwre/145/2/mwr-d-16-0106.1.xml (visited on 11/15/2023).

[76] F. Rawlins et al. “The Met Office global four-dimensional variational data assimilation scheme”. en. In: *Quarterly Journal of the Royal Meteorological Society* 133.623 (2007). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/qj.32, pp. 347–362. ISSN: 1477-870X. DOI: 10.1002/qj.32. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/qj.32 (visited on 11/14/2023).

[77] A. M. Clayton, A. C. Lorenc, and D. M. Barker. “Operational implementation of a hybrid ensemble/4D-Var global data assimilation system at the Met Office”. en. In: *Quarterly Journal of the Royal Meteorological Society* 139.675 (2013). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/qj.2054, pp. 1445–1461. ISSN: 1477-870X. DOI: 10.1002/qj.2054. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/qj.2054 (visited on 11/15/2023).

[78] Fathalla A. Rihan, Chris G. Collier, and Ian Roulstone. “Four-dimensional variational data assimilation for Doppler radar wind data”. In: *Journal of Computational and Applied Mathematics* 176.1 (Apr. 2005), pp. 15–34. ISSN: 0377-0427. DOI: 10.1016/j.cam.2004.07.003. URL: https://www.sciencedirect.com/science/article/pii/S0377042704003176 (visited on 11/15/2023).

[79] Per Christian Hansen and Dianne O’leary. “The Use of the L-Curve in the Regularization of Discrete Ill-Posed Problems”. In: *SIAM J. Sci. Comput.* 14 (Nov. 1993), pp. 1487–1503. DOI: 10.1137/0914086.

[80] Reuben W. Nixon-Hill and Daniel Shapero. *ReubenHill/point-data-paper-code: Consistent Point Data Assimilation in Firedrake and Icepack: Code.* May 2023. DOI: 10.5281/zenodo.7950441. URL: https://doi.org/10.5281/zenodo.7950441.

[81] Lorraine Schwartz. “On bayes procedures”. In: *Zeitschrift für Wahrscheinlichkeitstheorie und verwandte Gebiete* 4.1 (1965). Publisher: Springer, pp. 10–26.

[82] Douglas R. MacAyeal. “A tutorial on the use of control methods in ice-sheet modeling”. en. In: *Journal of Glaciology* 39.131 (1993). Publisher: Cambridge University Press, pp. 91–98. ISSN: 0022-1430, 1727-5652. DOI: 10.3189/S0022143000015744. URL: https://www.cambridge.org/core/journals/journal-of-glaciology/article/tutorial-on-the-use-of-control-methods-in-icesheet-modeling/CD1F93F5EBF26E7CDAADC376CA2BC6DE (visited on 01/25/2023).

[83] Douglas R. MacAyeal, Robert A. Bindschadler, and Theodore A. Scambos. “Basal friction of Ice Stream E, West Antarctica”. en. In: *Journal of Glaciology* 41.138 (1995). Publisher: Cambridge University Press, pp. 247–262. ISSN: 0022-1430, 1727-5652. DOI: 10.3189/S0022143000016154. URL: https://www.cambridge.org/core/journals/journal-of-glaciology/article/basal-friction-of-ice-stream-e-west-antarctica/B6ADB22419F499A72AE52FF6D7B71AD4 (visited on 01/24/2023).

[84] Tobin Isaac et al. “Scalable and efficient algorithms for the propagation of uncertainty from data through inference to prediction for large-scale problems, with application to flow of the Antarctic ice sheet”. In: *Journal of Computational Physics* 296 (Sept. 2015), pp. 348–368. ISSN: 0021-9991. DOI: 10.1016/j.jcp.2015.04.047. URL: https://www.sciencedirect.com/science/article/pii/S0021999115003046 (visited on 04/19/2024).

[85] *Software used in ‘Consistent Point Data Assimilation in Firedrake and Icepack: Unknown Conductivity Demonstration’*. Mar. 2023. DOI: 10.5281/zenodo.7741741. URL: https://doi.org/10.5281/zenodo.7741741.

[86] Thomas George and Vivek Sarin. “Domain Decomposition”. In: *Encyclopedia of Parallel Computing*. Ed. by David Padua. Boston, MA: Springer US, 2011, pp. 578–587. ISBN: 978-0-387-09766-4. DOI: 10.1007/978-0-387-09766-4_291. URL: https://doi.org/10.1007/978-0-387-09766-4_291.

[87] S. B. Kuang, A. B. Yu, and Z. S. Zou. “A new point-locating algorithm under three-dimensional hybrid meshes”. In: *International Journal of Multiphase Flow* 34.11 (Nov. 2008), pp. 1023–1030. ISSN: 0301-9322. DOI: 10.1016/j.ijmultiphaseflow.2008.06.007. URL: https://www.sciencedirect.com/science/article/pii/S030193220800102X (visited on 10/19/2023).

[88] Severin Strobl, Marcus N. Bannerman, and Thorsten Pöschel. “Robust event-driven particle tracking in complex geometries”. In: *Computer Physics Communications* 254 (Sept. 2020), p. 107229. ISSN: 0010-4655. DOI: 10.1016/j.cpc.2020.107229. URL: https://www.sciencedirect.com/science/article/pii/S0010465520300667 (visited on 10/19/2023).

[89] Rene Gassmöller et al. “Flexible and Scalable Particle-in-Cell Methods With Adaptive Mesh Refinement for Geodynamic Computations”. en. In: *Geochemistry, Geophysics, Geosystems* 19.9 (2018). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/2018GC007508, pp. 3596–3604. ISSN: 1525-2027. DOI: 10 . 1029 / 2018GC007508. URL: https://onlinelibrary.wiley.com/doi/abs/10.1029/2018GC007508 (visited on 10/19/2023).

[90] Jakob M. Maljaars, Chris N. Richardson, and Nathan Sime. “LEoPart: A particle library for FEniCS”. en. In: *Computers & Mathematics with Applications*. Development and Application of Open-source Software for Problems with Numerical PDEs 81 (Jan. 2021), pp. 289–315. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2020.04.023. URL: https://www.sciencedirect.com/science/article/pii/S089812212030170X (visited on 03/31/2023).

[91] Margaret Lawson, William Gropp, and Jay Lofstead. *Exploring Spatial Indexing for Accelerated Feature Retrieval in HPC*. arXiv:2106.13972 [cs]. Aug. 2021. URL: http://arxiv.org/abs/2106.13972 (visited on 10/18/2023).

[92] Norbert Beckmann et al. “The R*-tree: an efficient and robust access method for points and rectangles”. In: *Proceedings of the 1990 ACM SIGMOD international conference on Management of data*. SIGMOD ’90. New York, NY, USA: Association for Computing Machinery, May 1990, pp. 322–331. ISBN: 978-0-89791-365-2. DOI: 10.1145/93597.98741. URL: https://dl.acm.org/doi/10.1145/93597.98741 (visited on 10/18/2023).

[93] Marios Hadjieleftheriou. *libspatialindex*. Oct. 2019. URL: https://github.com/libspatialindex/libspatialindex.

[94] E.G. Gilbert, D.W. Johnson, and S.S. Keerthi. “A fast procedure for computing the distance between complex objects in three-dimensional space”. In: *IEEE Journal on Robotics and Automation* 4.2 (Apr. 1988). Conference Name: IEEE Journal on Robotics and Automation, pp. 193–203. ISSN: 2374-8710. DOI: 10.1109/56.2083. URL: https://ieeexplore.ieee.org/abstract/document/2083?casa_token=jD9mGKEsjy0AAAAA:T9TF1h7czvnLUacFLusY0-i27Rh8G6nb1lLaa2CeUAY-h8rKQK5p0EiaZVAurCAA17CqA-qNdA (visited on 10/24/2023).

[95] Aaron Meurer et al. “SymPy: symbolic computing in Python”. In: *PeerJ Computer Science* 3 (Jan. 2017), e103. ISSN: 2376-5992. DOI: 10.7717/peerj-cs.103. URL: https://doi.org/10.7717/peerj-cs.103.

[96] James R. Maddison. “Adaptive mesh modelling of the thermally driven annulus”. eng. Book Title: Adaptive mesh modelling of the thermally driven annulus. PhD thesis. Pro-

Quest Dissertations Publishing, 2011. URL: https://search.proquest.com/docview/1221974795?pq-origsite=primo (visited on 04/19/2024).

[97] Junchao Zhang et al. *The PetscSF Scalable Communication Layer*. arXiv:2102.13018 [cs]. May 2021. DOI: 10.48550/arXiv.2102.13018. URL: http://arxiv.org/abs/2102.13018 (visited on 07/12/2023).

[98] William Robert Saunders. *NESO-Particles*. June 2023. DOI: 10.5281/zenodo.8386778. URL: https://doi.org/10.5281/zenodo.8386778.

[99] L. Ridgway Scott and Shangyou Zhang. "Higher-Dimensional Nonnested Multigrid Methods". In: *Mathematics of Computation* 58.198 (1992). Publisher: American Mathematical Society, pp. 457–466. ISSN: 0025-5718. DOI: 10.2307/2153196. URL: https://www.jstor.org/stable/2153196 (visited on 09/28/2023).

[100] P. E. Farrell et al. "Conservative interpolation between unstructured meshes via supermesh construction". en. In: *Computer Methods in Applied Mechanics and Engineering* 198.33 (July 2009), pp. 2632–2642. ISSN: 0045-7825. DOI: 10.1016/j.cma.2009.03.004. URL: http://www.sciencedirect.com/science/article/pii/S0045782509001315 (visited on 07/27/2020).

[101] Peter Bastian et al. *The DUNE Framework: Basic Concepts and Recent Developments*. arXiv:1909.13672 [cs]. June 2020. URL: http://arxiv.org/abs/1909.13672 (visited on 10/28/2023).

[102] David Ham et al. "Automating the generation of finite element dynamical cores with Firedrake". In: *EGU General Assembly Conference Abstracts*. 2017, p. 17987.

[103] Joachim Schöberl. "NETGEN An advancing front 2D/3D-mesh generator based on abstract rules". en. In: *Computing and Visualization in Science* 1.1 (July 1997), pp. 41–52. ISSN: 1432-9360. DOI: 10.1007/s007910050004. URL: https://doi.org/10.1007/s007910050004 (visited on 10/10/2023).

[104] Remi Lam et al. *GraphCast: Learning skillful medium-range global weather forecasting*. arXiv:2212.12794 [physics]. Aug. 2023. URL: http://arxiv.org/abs/2212.12794 (visited on 11/22/2023).

[105] K. R. RAJAGOPAL. "On implicit constitutive theories for fluids". In: *Journal of Fluid Mechanics* 550 (2006). Publisher: Cambridge University Press, pp. 243–249. DOI: 10.1017/S0022112005008025.

[106] Nacime Bouziani and David A. Ham. *Escaping the abstraction: a foreign function interface for the Unified Form Language [UFL]*. arXiv:2111.00945 [cs, math]. Nov. 2021. URL: http://arxiv.org/abs/2111.00945 (visited on 11/07/2023).

[107] Baudouin Fraeijs de Veubeke. "Displacement and equilibrium models in the finite element method". In: *Stress Analysis*. John Wiley & Sons, 1965.

[108] Leonard R Herrmann. "Elasticity equations for incompressible and nearly incompressible materials by a variational theorem." In: *AIAA journal* 3.10 (1965), pp. 1896–1900.

[109] Daniel R. Shapero et al. "icepack: a new glacier flow modeling package in Python, version 1.0". English. In: *Geoscientific Model Development* 14.7 (July 2021). Publisher: Copernicus GmbH, pp. 4593–4616. ISSN: 1991-959X. DOI: 10.5194/gmd-14-4593-2021. URL: https://gmd.copernicus.org/articles/14/4593/2021/ (visited on 11/21/2023).

[110] Tuomas Kärnä et al. "Thetis coastal ocean model: discontinuous Galerkin discretization for the three-dimensional hydrostatic equations". English. In: *Geoscientific Model Development* 11.11 (Oct. 2018). Publisher: Copernicus GmbH, pp. 4359–4382. ISSN: 1991-959X. DOI: https://doi.org/10.5194/gmd-11-4359-2018. URL: https://gmd.copernicus.org/articles/11/4359/2018/ (visited on 07/30/2020).

[111] Fei Tao et al. "Digital twin modeling". In: *Journal of Manufacturing Systems* 64 (July 2022), pp. 372–389. ISSN: 0278-6125. DOI: 10.1016/j.jmsy.2022.06.015. URL: https://www.sciencedirect.com/science/article/pii/S0278612522001108 (visited on 11/22/2023).

[112] A. Apte, C. K. R. T. Jones, and A. M. Stuart. "A Bayesian approach to Lagrangian data assimilation". In: *Tellus A: Dynamic Meteorology and Oceanography* 60.2 (Jan. 2008). Publisher: Taylor & Francis _eprint: https://doi.org/10.1111/j.1600-0870.2007.00295.x, pp. 336–347. ISSN: null. DOI: 10.1111/j.1600-0870.2007.00295.x. URL: https://doi.org/10.1111/j.1600-0870.2007.00295.x (visited on 10/25/2023).

[113] S. L. Cotter, M. Dashti, and A. M. Stuart. "Variational data assimilation using targetted random walks". en. In: *International Journal for Numerical Methods in Fluids* 68.4 (2012). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/fld.2510, pp. 403–421. ISSN: 1097-0363. DOI: 10.1002/fld.2510. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/fld.2510 (visited on 10/25/2023).

[114] E. A. Hathway et al. "CFD simulation of airborne pathogen transport due to human activities". In: *Building and Environment* 46.12 (Dec. 2011), pp. 2500–2511. ISSN: 0360-1323. DOI: 10.1016/j.buildenv.2011.06.001. URL: https://www.sciencedirect.com/science/article/pii/S0360132311001727 (visited on 10/24/2023).

[115] D. C. Cohen Stuart, C. R. Kleijn, and S. Kenjereš. "An efficient and robust method for Lagrangian magnetic particle tracking in fluid flow simulations on unstructured grids". In: *Computers & Fluids* 40.1 (Jan. 2011), pp. 188–194. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2010.09.001. URL: https://www.sciencedirect.com/science/article/pii/S004579301000229X (visited on 10/24/2023).

[116] Eric J. M. Delhez et al. “Toward a general theory of the age in ocean modelling”. In: *Ocean Modelling* 1.1 (Jan. 1999), pp. 17–27. ISSN: 1463-5003. DOI: 10.1016/S1463-5003(99)00003-7. URL: https://www.sciencedirect.com/science/article/pii/S1463500399000037 (visited on 10/24/2023).

[117] Paul J. Tackley and Scott D. King. “Testing the tracer ratio method for modeling active compositional fields in mantle convection simulations”. en. In: *Geochemistry, Geophysics, Geosystems* 4.4 (2003). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1029/2001GC000214. ISSN: 1525-2027. DOI: 10.1029/2001GC000214. URL: https://onlinelibrary.wiley.com/doi/abs/10.1029/2001GC000214 (visited on 10/24/2023).

[118] Eric Deleersnijder, Jean-Michel Campin, and Eric J. M. Delhez. “The concept of age in marine modelling: I. Theory and preliminary model results”. In: *Journal of Marine Systems* 28.3 (Apr. 2001), pp. 229–267. ISSN: 0924-7963. DOI: 10.1016/S0924-7963(01)00026-4. URL: https://www.sciencedirect.com/science/article/pii/S0924796301000264 (visited on 10/24/2023).

[119] Amvrossios C. Bagtzoglou, David E. Dougherty, and Andrew F. B. Tompson. “Application of particle methods to reliable identification of groundwater pollution sources”. en. In: *Water Resources Management* 6.1 (Mar. 1992), pp. 15–23. ISSN: 1573-1650. DOI: 10.1007/BF00872184. URL: https://doi.org/10.1007/BF00872184 (visited on 10/24/2023).

[120] Oliver J. Tooth, Helen L. Johnson, and Chris Wilson. “Lagrangian Overturning Pathways in the Eastern Subpolar North Atlantic”. EN. In: *Journal of Climate* -1.aop (Oct. 2022). Publisher: American Meteorological Society Section: Journal of Climate, pp. 1–53. ISSN: 0894-8755, 1520-0442. DOI: 10.1175/JCLI-D-21-0985.1. URL: https://journals.ametsoc.org/view/journals/clim/aop/JCLI-D-21-0985.1/JCLI-D-21-0985.1.xml (visited on 11/04/2022).

[121] G. D. Martin, E. Loth, and D. Lankford. “Particle host cell determination in unstructured grids”. In: *Computers & Fluids* 38.1 (Jan. 2009), pp. 101–110. ISSN: 0045-7930. DOI: 10.1016/j.compfluid.2008.01.005. URL: https://www.sciencedirect.com/science/article/pii/S0045793008000261 (visited on 10/19/2023).

[122] A. Haselbacher, F.M. Najjar, and J.P. Ferry. “An efficient and robust particle-localization algorithm for unstructured grids”. en. In: *Journal of Computational Physics* 225.2 (Aug. 2007), pp. 2198–2213. ISSN: 00219991. DOI: 10.1016/j.jcp.2007.03.018. URL: https://linkinghub.elsevier.com/retrieve/pii/S002199910700126X (visited on 10/19/2023).

[123] Graham B. Macpherson, Niklas Nordin, and Henry G. Weller. "Particle tracking in unstructured, arbitrary polyhedral meshes for use in CFD and molecular dynamics". en. In: *Communications in Numerical Methods in Engineering* 25.3 (2009). _eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cnm.1128, pp. 263–273. ISSN: 1099-0887. DOI: 10 . 1002 / cnm . 1128. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cnm.1128 (visited on 10/19/2023).

[124] Juan Manuel Carmona Loaiza, Graziano Giuliani, and Giuseppe Fiameni. "Big-Data in Climate Change Models — A Novel Approach with Hadoop MapReduce". In: *2017 International Conference on High Performance Computing Simulation (HPCS)*. July 2017, pp. 45–50. DOI: 10.1109/HPCS.2017.17.

[125] Veronika Eyring et al. "Overview of the Coupled Model Intercomparison Project Phase 6 (CMIP6) experimental design and organisation". In: *Geoscientific Model Development Discussions* 8 (Dec. 2015), pp. 10539–10583. DOI: 10.5194/gmdd-8-10539-2015.

[126] *Energy and the Information Infrastructure Part 4: Data is 'The New Oil,' Blowing Past the Zettabyte Era — RealClearEnergy*. Library Catalog: www.realclearenergy.org. URL: https://www.realclearenergy.org/articles/2019/02/01/energy_and_the_information_infrastructure_part_4_data_is_the_new_oil_blowing_past_the_zettabyte_era_110389.html (visited on 07/31/2020).

[127] *Earth System Modeling Must Become More Energy Efficient*. en-US. Library Catalog: eos.org. URL: https://eos.org/opinions/earth-system-modeling-must-become-more-energy-efficient (visited on 07/31/2020).

[128] Iselin Medhaug and Tore Furevik. "North Atlantic 20th century multidecadal variability in coupled climate models: Sea surface temperature and ocean overturning circulation". In: *Ocean Science* 7 (June 2011), pp. 389–404. DOI: 10.5194/os-7-389-2011.

[129] B. N. Lawrence et al. "Storing and manipulating environmental big data with JASMIN". In: *2013 IEEE International Conference on Big Data*. Oct. 2013, pp. 68–75. DOI: 10.1109/BigData.2013.6691556.

[130] Computational And Information Systems Laboratory. *CMIP Analysis Platform*. en. Publisher: UCAR/NCAR. 2016. URL: https://www2.cisl.ucar.edu/resources/cmip-analysis-platform (visited on 07/31/2020).

[131] *Unidata — NetCDF*. URL: https://www.unidata.ucar.edu/software/netcdf/ (visited on 07/29/2020).

[132] Charles S. Zender. "Analysis of self-describing gridded geoscience data with netCDF Operators (NCO)". en. In: *Environmental Modelling & Software* 23.10 (Oct. 2008), pp. 1338–1342. ISSN: 1364-8152. DOI: 10.1016/j.envsoft.2008.03.004. URL: http:

//www.sciencedirect.com/science/article/pii/S1364815208000431 (visited on 08/03/2020).

[133] *The NCAR Command Language (Version 6.6.2) [Software].* Boulder, Colorado, 2019. URL: http://dx.doi.org/10.5065/D6WD3XH5 (visited on 08/03/2020).

[134] *Pangeo — Pangeo documentation.* URL: https://pangeo.io/ (visited on 07/29/2020).

[135] Tina Erica Odaka et al. "The Pangeo Ecosystem: Interactive Computing Tools for the Geosciences: Benchmarking on HPC". en. In: *Tools and Techniques for High Performance Computing.* Ed. by Guido Juckeland and Sunita Chandrasekaran. Communications in Computer and Information Science. Cham: Springer International Publishing, 2020, pp. 190–204. ISBN: 978-3-030-44728-1. DOI: 10.1007/978-3-030-44728-1_12.

[136] Venkatramani Balaji et al. "Requirements for a global data infrastructure in support of CMIP6". en. In: *Geoscientific Model Development* 11.9 (Sept. 2018), pp. 3659–3680. ISSN: 1991-9603. DOI: 10.5194/gmd-11-3659-2018. URL: https://gmd.copernicus.org/articles/11/3659/2018/ (visited on 07/31/2020).

[137] Niall H. Robinson, Joe Hamman, and Ryan Abernathey. "Seven Principles for Effective Scientific Big-DataSystems". In: *arXiv:1908.03356 [cs]* (June 2020). arXiv: 1908.03356. URL: http://arxiv.org/abs/1908.03356 (visited on 07/31/2020).

[138] Martina Stockhause et al. "Data Distribution Centre Support for the IPCC Sixth Assessment". en. In: *Data Science Journal* 18.1 (June 2019). Number: 1 Publisher: Ubiquity Press, p. 20. ISSN: 1683-1470. DOI: 10.5334/dsj-2019-020. URL: http://datascience.codata.org/articles/10.5334/dsj-2019-020/print/ (visited on 07/31/2020).

[139] W. T. Barnes, S. J. Bradshaw, and N. M. Viall. "Understanding Heating in Active Region Cores through Machine Learning. I. Numerical Modeling and Predicted Observables". en. In: *The Astrophysical Journal* 880.1 (July 2019). Publisher: American Astronomical Society, p. 56. ISSN: 0004-637X. DOI: 10.3847/1538-4357/ab290c. URL: https://doi.org/10.3847%2F1538-4357%2Fab290c (visited on 07/31/2020).

[140] Dask Development Team. *Dask: Library for dynamic task scheduling.* 2016. URL: https://dask.org.

[141] Stephan Hoyer and Joseph J. Hamman. "xarray: N-D labeled Arrays and Datasets in Python". en. In: *Journal of Open Research Software* 5 (Apr. 2017), p. 10. ISSN: 2049-9647. DOI: 10.5334/jors.148. URL: http://openresearchsoftware.metajnl.com/articles/10.5334/jors.148/ (visited on 07/29/2020).

[142] Met Office. *Iris: A powerful, format-agnostic, and community-driven Python package for analysing and visualising Earth science data.* v3.6. Exeter, Devon, 2010. DOI: 10.5281/zenodo.7948293. URL: http://scitools.org.uk/.

[143] Wes McKinney et al. “Data structures for statistical computing in python”. In: *Proceedings of the 9th Python in Science Conference*. Vol. 445. Austin, TX, 2010, pp. 51–56.

Appendices

Appendix A

Chapter Appendices

A.1 Appendix to Chapter 4

A.1.1 Analysis of Point Clouds

Here we analyse how point clouds can considered in the language of meshes and finite elements.

Definition A.1.1 (Point Cloud)**.** A point cloud is a set of N points $\{X_i\}_{i=0}^{N-1} \in \mathbb{R}^n$ where each point has values (be they scalar, vector or tensor) associated with it.

Definition A.1.2 (Vertex-Only Mesh)**.** Consider an arbitrary meshed domain $\Omega \subset \mathbb{R}^n$ with geometric dimension n and some topological dimension dimension $\leq n$ as typically employed in the finite element method (see for example [16]). A vertex-only mesh is a disconnected mesh $\Omega_v \subseteq \Omega$ with topological dimension zero (its cells are vertices) and geometric dimension n.

Why is this useful? A vertex-only mesh provides a method of describing the locations of the points in a point cloud $\{X_i\}$ as mesh cells. We can define then a function space on the mesh: functions in this function space then associate values with each point.

Functions on Vertex-Only Meshes

Let

$$\Omega_v \subseteq \Omega \subset \mathbb{R}^n \tag{A.1.1}$$

be a vertex-only mesh immersed in a parent mesh Ω with N vertices $\{X_i\}_{i=0}^{N-1}$.

Only one function space need be defined on a vertex-only mesh since the topological dimension limits the available degrees of freedom. An appropriate function space has no continuity constraints and should take a single value at each cell. A Polynomial degree 0 Discontinuous Galerkin ("P0DG") function space provides these properties. Therefore let

$$V = \text{P0DG}(\Omega_v) : \Omega_v \to \mathbb{R}^m \tag{A.1.2}$$

For completeness, both the scalar and vector valued function space cases are analysed in this section. As the analysis will demonstrate, each component of a vector function space like this can be treated as the equivalent scalar function space. Future analysis need not, therefore, deal with the vector valued function case and its more cumbersome bookkeeping.

Integration Sums Point Evaluations

Since these functions are only defined at vertices, integration necessarily yields the sum of point evaluations within the integration domain

$$\int_{\Omega_v} \boldsymbol{v}(x)dx = \sum_{i=0}^{N-1} \boldsymbol{v}(X_i) \quad \forall\, \boldsymbol{v} \in V : \Omega_v \to \mathbb{R}^m. \tag{A.1.3}$$

[0]Department of Mathematics, Imperial College London

Notes on the Choice of Function Space and Integration Behaviour

As with other finite element function spaces, functions on a vertex-only mesh are defined by a union of restrictions to each cell: here, vertices. Each restriction, $\hat{f}$, can, in this case, be described as a polynomial on a local 'reference' coordinate $\mathcal{K}$. The local reference coordinate $\mathcal{K}$ is an arbitrary zero dimensional coordinate, and the only possible polynomial is $\hat{f}(\mathcal{K}) = c$ for $c \in \mathbb{R}$. This coincides with the definition of P0DG on meshes of higher dimension, so it makes sense to use that name. In higher dimensions, the value of c depends on the choice of dual basis functional: there are an infinite number of possibilities with popular choices being $\phi^*(\hat{f}) = \int_K \hat{f} dx_K$ and $\phi^*(\hat{f}) = \hat{f}(X)$ with X being at the centre of the reference cell K.

In zero dimensions both of these reduce to the value of the function $\hat{f}$ on the reference vertex K: for the integration case, Lebesgue integration of a single point over its domain (itself a point) is the value of $\hat{f}$ at the point (see the below point about choice of measure). For the cell centre case, the cell has no extent so its centre is the value at the point itself. The function space on the vertex-only mesh could equivalently be called P0CG or CG0 ('Polynomial degree 0 Continuous Galerkin' or just 'Continuous Galerkin 0'), but this is less natural: it has no analogy in higher dimensions and any concept of 'continuity' is moot due to the cells being disconnected.

In zero dimensions there is no Lebesgue measure for performing integration. Nevertheless integration over a vertex-only mesh is desirable in order to characterise elements of the dual space as integrals (see for example the use of vertex-only meshes for point forcing). The natural choice for integrating over a vertex only mesh are Dirac measures $\mathrm{d}\delta_{X_i}$ at each vertex X_i:

$$\int_{\Omega_v} f(x)\mathrm{d}\delta_{X_i} = f(X_i). \tag{A.1.4}$$

More generally one can define a discrete measure for integrating over a vertex-only mesh as a weighted sum of Dirac measures. Weightings of 1 are natural because (a) there is no reason to prefer one vertex over another and (b) different weights can be achieved by taking a Euclidean inner product of the integrand with a weights function defined on the vertex-only mesh.

Non-Differentiability

Given that the domain Ω_v is disconnected and has topological dimension zero a derivative is difficult to define. For most imaginable purposes therefore,

$$\nabla v \text{ is undefined} \quad \forall\, v \in V : \Omega_v \to \mathbb{R}. \tag{A.1.5}$$

Global Basis Functions

Functions in V can be expressed as a sum of $D = N \times m$ global vector valued basis functions also in V

$$\boldsymbol{v}(x) = \sum_{i=0}^{D-1} v_i \boldsymbol{\phi}_i(x) = \sum_{i=0}^{N-1} \sum_{k=0}^{m-1} v_{ik} \boldsymbol{\phi}_{ik}(x) \tag{A.1.6}$$

where

$$\mathrm{span}(\{\boldsymbol{\phi_i}\}_{i=0}^{D-1}) = V \tag{A.1.7}$$

and

$$\boldsymbol{\phi}_{ik}(x) = \begin{cases} \boldsymbol{e}_k & \text{if } x = X_i, \\ \boldsymbol{0} & \text{otherwise.} \end{cases} \tag{A.1.8}$$

where $\boldsymbol{e}_k$ is the k^{th} cartesian unit vector ($k = 0, \dots, m-1$).

For scalar valued functions where $D = N$ this simplifies to

$$\begin{aligned} v(x) &= \sum_{i=0}^{N-1} v_i \phi_i(x) \\ \mathrm{span}(\{\phi_i\}_{i=0}^{N-1}) &= V \\ \phi_i(x) &= \begin{cases} 1 & \text{if } x = X_i, \\ 0 & \text{otherwise.} \end{cases} \end{aligned} \tag{A.1.9}$$

Note that if the function is scalar valued then for a given vertex $x = X_i$ we yield an appropriate equivalent coefficient v_i

$$v(x) = v(X_i) = v_i. \tag{A.1.10}$$

If the function is vector valued then the same result is obtained for each dimension $k = 0, \dots, m-1$

$$\boldsymbol{v}(x) \cdot \boldsymbol{e}_k = \boldsymbol{v}(X_i) \cdot \boldsymbol{e}_k = v_{ik}. \tag{A.1.11}$$

Integrating Global Basis Functions

When integrating a a global basis function from equation A.1.8 it is useful to note that the Kronecker delta makes an appearance

$$\int_{\Omega_v} \boldsymbol{\phi}_i(x) = \sum_{j=0}^{D-1} \boldsymbol{\phi}_i(X_j) = \sum_{j=0}^{N-1} \sum_{k=0}^{m-1} \boldsymbol{\phi}_{ik}(X_j) = \sum_{j=0}^{N-1} \sum_{k=0}^{m-1} \delta_{ij} \boldsymbol{e}_k = \sum_{k=0}^{m-1} \boldsymbol{e}_k \quad \forall\, i = 0, \dots, D-1. \tag{A.1.12}$$

For scalar valued functions this simplifies to

$$\int_{\Omega_v} \phi_i(x) = \sum_{j=0}^{D-1} \phi_i(X_j) = \sum_{j=0}^{N-1} \phi_i(X_j) = \sum_{j=0}^{N-1} \delta_{ij} = 1 \quad \forall\, i = 0, \ldots, D-1. \tag{A.1.13}$$

L^2 Inner Product Equal to l_2

Given $u, v \in V : \Omega_v \to \mathbb{R}$, it follows from equations A.1.8, A.1.3 and A.1.13 that the L^2 inner product is equal to the l_2 inner product of each function.

$$\begin{aligned}
\langle u, v\rangle_{L^2} &= \int_{\Omega_v} u(x)v(x)dx \\
&= \int_{\Omega_v} \sum_{i=0}^{N-1} u_i\phi_i(x) \sum_{j=0}^{N-1} v_j\phi_j(x)dx \\
&= \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} u_i v_j \int_{\Omega_v} \phi_i(x)\phi_j(x)dx \quad \text{(by linearity of integration)} \\
&= \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} u_i v_j \sum_{k=0}^{N-1} \delta_{ik}\delta_{jk}dx \qquad \text{(from equation A.1.13)} \\
&= \sum_{i=0}^{N-1} u_i v_i = \langle u, v\rangle_{l_2}
\end{aligned} \tag{A.1.14}$$

For real vector valued functions $\boldsymbol{u}, \boldsymbol{v} \in V : \Omega_v \to \mathbb{R}^m$, it similarly follows from equations A.1.8, A.1.3 and A.1.12, that the L^2 inner product is equal to the l_2 inner products for the k^{th} vector component ($k = 0, \dots, m-1$) of each function.

$$
\begin{aligned}
\langle \boldsymbol{u}, \boldsymbol{v} \rangle_{L^2} &= \int_{\Omega_v} \big\langle \boldsymbol{u}(x), \boldsymbol{v}(x) \big\rangle_{l_2} dx \\
&= \int_{\Omega_v} \left\langle \sum_{i=0}^{D-1} u_i \boldsymbol{\phi}_i(x), \sum_{j=0}^{D-1} v_j \boldsymbol{\phi}_j(x) \right\rangle_{l_2} dx \\
&= \int_{\Omega_v} \left\langle \sum_{i=0}^{N-1} \sum_{k=0}^{m-1} u_{ik} \boldsymbol{\phi}_{ik}(x), \sum_{j=0}^{N-1} \sum_{l=0}^{m-1} v_{jl} \boldsymbol{\phi}_{jl}(x) \right\rangle_{l_2} dx \\
&= \sum_{i=0}^{N-1} \sum_{k=0}^{m-1} \sum_{j=0}^{N-1} \sum_{l=0}^{m-1} u_{ik} v_{jl} \int_{\Omega_v} \big\langle \boldsymbol{\phi}_{ik}(x), \boldsymbol{\phi}_{jl}(x) \big\rangle_{l_2} dx \qquad \text{(by linearity of integration} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{and } \langle \cdot, \cdot \rangle_{l_2}) \\
&= \sum_{i=0}^{N-1} \sum_{k=0}^{m-1} \sum_{j=0}^{N-1} \sum_{l=0}^{m-1} \sum_{p=0}^{N-1} u_{ik} v_{jl} \big\langle \boldsymbol{\phi}_{ik}(X_p), \boldsymbol{\phi}_{jl}(X_p) \big\rangle_{l_2} \qquad \text{(from equation A.1.12)} \\
&= \sum_{i=0}^{N-1} \sum_{k=0}^{m-1} \sum_{j=0}^{N-1} \sum_{l=0}^{m-1} \sum_{p=0}^{N-1} u_{ik} v_{jl} \big\langle \delta_{ip} \boldsymbol{e}_k, \delta_{jp} \boldsymbol{e}_l \big\rangle_{l_2} \qquad \text{(also from equation A.1.12)} \\
&= \sum_{i=0}^{N-1} \sum_{k=0}^{m-1} \sum_{j=0}^{N-1} \sum_{l=0}^{m-1} \sum_{p=0}^{N-1} u_{ik} v_{jl} \delta_{ip} \delta_{kl} \delta_{jp} \qquad \text{(by evaluating } \langle \cdot, \cdot \rangle_{l_2}) \\
&= \sum_{i=0}^{N-1} \sum_{k=0}^{m-1} u_{ik} v_{ik} = \langle u_k, v_k \rangle_{l_2} \quad k = 0, \dots, m-1.
\end{aligned}
\tag{A.1.15}
$$

A word on complex basis functions, i.e. $u, v : \Omega_v \to \mathbb{C}^m$, our L^2 inner product is still the l_2 inner product, with the complex conjugate of one set of basis function coefficients

$$\int_{\Omega_v} \big\langle \boldsymbol{u}(x), \bar{\boldsymbol{v}}(x) \big\rangle_{l_2} dx = \langle u_k, \bar{v}_k \rangle_{l_2} \quad k = 0, \dots, m-1. \tag{A.1.16}$$

This analysis focuses on real valued functions.

Mass Matrix is Identity

It is useful to consider equation A.1.14 more closely in the context of the finite element method. If v is treated as some trial function for which we are seeking a solution with the finite element method and u an arbitrary test function then the mass matrix $\boldsymbol{M}$, yielded by the L^2 inner product, is seen to be the identity matrix.

$$
\begin{aligned}
u, v \in V : \Omega_v \to \mathbb{R} \quad &\text{or} \quad u, v \in W : \Omega \to \mathbb{R} \\
\langle u, v\rangle &= \int_{\Omega_v} u(x)v(x)dx \\
&= \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} u_i v_j \underbrace{\int_{\Omega_v} \phi_i(x)\phi_j(x)dx}_{\boldsymbol{M}} && (\equiv \boldsymbol{u}^{\boldsymbol{T}}\boldsymbol{M}\boldsymbol{v} \text{ once the integral is discretised}) \\
&= \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} u_i v_j \sum_{k=0}^{N-1} \delta_{ik}\delta_{jk}dx && (\equiv \boldsymbol{u}^T\boldsymbol{I}\boldsymbol{I}\boldsymbol{v}) \\
&= \sum_{i=0}^{N-1}\sum_{j=0}^{N-1} u_i v_j \sum_{k=0}^{N-1} \delta_{ij}dx && (\equiv \boldsymbol{u}^T\boldsymbol{I}\boldsymbol{v}) \\
&= \sum_{i=0}^{N-1} u_i v_i = \langle u, v\rangle_{l_2} && (\equiv \boldsymbol{u}^T\boldsymbol{v}).
\end{aligned}
\tag{A.1.17}
$$

The same result is found when considering vector valued functions since the same integral similarly reduces to sums of Kronecker deltas. Note this is only valid for real valued functions.

It is usual for a P0DG function space to yield a diagonal mass matrix where the values at each diagonal entry M_{ii} are the length/area/volume (in 1D/2D/3D) of the cell with globally numbered basis function ϕ_i. Since integration performs point evaluation we implicitly state that the "size" of each cell has unit length/area/volume (in 1D/2D/3D).

Ciarlet Triple Formulation of Point Evaluation Functions on a Vertex-Only Mesh

A local finite element on a vertex-only mesh can be formulated as a Ciarlet triple [34]

$$(\mathcal{K}, \mathcal{P}, \mathcal{N}). \tag{A.1.18}$$

For any given vertex $X_i \in \Omega_v$ we define a local coordinate $\tilde{X}_0 \in \mathbb{R}^n$ as an arbitrary vertex cell.[1] Therefore

$$\mathcal{K} = \tilde{X}_0. \tag{A.1.19}$$

$\mathcal{P}$ is a local vector valued function space (with value dimension m) of degree zero polynomials (constants)

$$\boldsymbol{f} \in \mathcal{P} \text{ where } \mathcal{P} : \tilde{X}_0 \to \mathbb{R}^m. \tag{A.1.20}$$

[1]Here we only consider real valued functions.

The set of $k+1$ local linear functional nodes which form a basis for $\mathcal{P}^*$

$$\begin{aligned} f^* \in \mathcal{P}^* \text{ where } \mathcal{P}^* : \mathcal{P} \to \mathbb{R} \\ \operatorname{span}(\mathcal{N}) = \mathcal{P}^* \\ \mathcal{N} = \{\phi_i^*\}_{i=0}^{k} \end{aligned} \tag{A.1.21}$$

are split by dimension $l = 0, \dots, m-1$ with corresponding unit vectors $\boldsymbol{e}_l$ and and are given by

$$\begin{aligned} \text{let } \phi_i^* = \phi_{jl}^* \\ \text{where} \quad 0 \le i \le k, \quad 0 \le j \le \frac{k+1}{m} - 1, \quad \text{and} \quad 0 \le l \le m-1, \\ \text{then} \quad \phi_{jl}^*(\boldsymbol{f}) = \boldsymbol{f}(\tilde{X}_j) \cdot \boldsymbol{e}_l = f_l(\tilde{X}_j) \end{aligned} \tag{A.1.22}$$

Since there is only one local point $\tilde{X}_j = \tilde{X}_0$ there are only as many nodes as dimensions m

$$k = m - 1 \implies (k+1)/m - 1 = 0 \implies \mathcal{N} = \{\phi_{0i}^*\}_{i=0}^{m-1}. \tag{A.1.23}$$

From the definition of the nodal basis in [34] we can now recover our corresponding local basis functions for each dimension $l = 0, \dots, m-1$

$$\begin{aligned} \operatorname{span}(\{\boldsymbol{\phi}_i\}_{i=0}^{k}) = \mathcal{P} \\ \text{let } \boldsymbol{\phi}_i = \boldsymbol{\phi}_{jl} \\ \text{where} \quad 0 \le i \le k, \quad 0 \le j \le \frac{k+1}{m} - 1 \quad \text{and} \quad 0 \le l \le m-1, \\ \text{then the nodal basis definition is} \quad \phi_{pl}^*(\boldsymbol{\phi}_{jl}) = \delta_{pj} \quad \forall\, l \end{aligned} \tag{A.1.24}$$

which gives a single basis function for each dimension $l = 0, \dots, m-1$

$$\begin{aligned} \phi_{pl}^*(\boldsymbol{\phi}_{jl}) = \boldsymbol{\phi}_{jl}(\tilde{X}_p) \cdot \boldsymbol{e}_l = \delta_{pj} \\ \tilde{X}_p = \tilde{X}_0 \quad \text{locally} \quad \implies p = 0 \\ \phi_{0l}^*(\boldsymbol{\phi}_{jl}) = \boldsymbol{\phi}_{jl}(\tilde{X}_0) \cdot \boldsymbol{e}_l = \delta_{0j} \quad \implies j = 0 \\ \boldsymbol{\phi}_{jl}(\tilde{x}) = \boldsymbol{e}_l \end{aligned} \tag{A.1.25}$$

which is the usual global basis function given in equation A.1.8 restricted locally to have domain $\tilde{x} = \tilde{X}_0$ $(j = 0)$. If $\mathcal{P}$ is a space of scalar valued functions $(m = 1)$ then no dimension splitting is required and the analysis simplifies to give one node and one basis function within the finite element

$$\begin{aligned}
f \in \mathcal{P} \text{ where } \mathcal{P} : \tilde{X}_0 \to \mathbb{R} \\
f^* \in \mathcal{P}^* \text{ where } \mathcal{P}^* : \mathcal{P} \to \mathbb{R} \\
\phi_i^*(f) = f(\tilde{X}_i) \\
\tilde{X}_i = \tilde{X}_0 \implies i = 0 \\
\mathcal{N} = \{\phi_0^*\} \\
\phi_0^*(\phi_j) = \phi_j(\tilde{X}_0) = \delta_{0j} \implies j = 0 \\
\phi_j = \phi_0 = 1.
\end{aligned} \tag{A.1.26}$$

This element has been added to FIAT [37] and UFL [6] by introducing vertex cells $\tilde{X}$ and, in FIAT, by defining an appropriate tabulation. The equivalent vector element is generated automatically by creating a scalar basis for each vector dimension which, as shown in this analysis, yields the correct result.

When extended to the entire domain we get

$$(\mathcal{K}_i, \mathcal{P}_i, \mathcal{N}_i) \; i = 0, \ldots, N-1 \tag{A.1.27}$$

for each vertex cell $\mathcal{K}_i = X_i \in \Omega_v$. The discontinuous global finite element function space is then P0DG (Polynomial degree 0 Discontinuous Galerkin) with vertex cells defined as

$$\text{P0DG}(\Omega_v) = V = \{f : f|_{X_i} \in \mathcal{P}_i, \; \forall X_i \in \Omega_v\}.^{2} \tag{A.1.28}$$

As usual

$$V^* = \text{span}(\{\mathcal{N}_i\}_{i=0}^{N-1}). \tag{A.1.29}$$

Cofunctions in L^2 are Self-Dual

The Reisz representation theorem is outlined in 1.4. As stated there, the mass matrix is the inner product matrix for the L^2 inner product. Since it is identity (equivalently, the L^2 inner product is equivalent to the l_2 inner product) then the cofunction to a given function is its transpose. For complex valued functions the cofunction is the conjugate-transpose.

Interpolation is L^2 Galerkin Projection

If we allow the whole domain to take on the value of some particular function then we find that dual evaluation interpolation, defined in Sect. 3.2 and L^2 Galerkin Projection, defined in Sect. 3.3, are equivalent.

Let $f_v \in \text{P0DG}(\Omega_v)$ be a scalar point evaluation function

$$f_v(x) = \mathcal{I}_{\text{P0DG}(\Omega_v)}(; f(x)) \tag{A.1.30}$$

[2] Adapted from lecture 2 of [1].

where f is function on the parent mesh $f \in \mathrm{FS}(\Omega)$. The necessary dual basis functions for dual evaluation interpolation are outlined in Sect. 4.4.

$$\begin{aligned} f_v(x) &= \begin{cases} f(x) & \text{if } x = \tilde{X}_i, \\ 0 & \text{otherwise} \end{cases} \\ &= \sum_{i=0}^{N-1} (f_v)_i \phi_i(x) \\ &= \sum_{i=0}^{N-1} f(X_i)\phi_i(x) \end{aligned} \tag{A.1.31}$$

using equation A.1.14 we see that this is an L^2 Galerkin projection of f into $V = \mathrm{P0DG}(\Omega_v)$

$$\langle v, f \rangle_{L^2} = \langle v, f \rangle_{l_2} = \sum_{i=0}^{N-1} v(X_i) f(X_i) = \langle v, f_v \rangle_{L^2}. \tag{A.1.32}$$

f_v is also the global interpolation of f into $V = \mathrm{P0DG}(\Omega_v)$

$$\begin{aligned} \left[\mathcal{I}_{\mathrm{P0DG}(\Omega_v)}(f)\right](x) &= \sum_{i=0}^{D-1} \phi_i^*(f)\phi_i(x) \\ &= \sum_{i=0}^{N-1} f(X_i)\phi_i(x) \quad \text{by extending equation A.1.22 to the whole} \\ &\qquad\qquad \text{domain with A.1.29 and noting that there is one} \\ &\qquad\qquad \text{basis function per cell on a vertex-only mesh} \\ &= f_v. \end{aligned} \tag{A.1.33}$$

If $\boldsymbol{f_v}$ is a vector valued point evaluation function for some function $\boldsymbol{f}$, the same analysis holds when considering each vector component $(f_v)_k = \boldsymbol{f_v} \cdot \boldsymbol{e}_k$ individually: The L^2 inner product is equal to the l_2 inner product for (see equation A.1.15) and there is still only one basis function per cell.

Equivalence with Dirac Deltas

Let Ω_v have N vertices $\{x_i\}_{i=0}^{N-1}$.

$$\int_{\Omega_v} \mathcal{I}_{\mathrm{P0DG}(\Omega_v)}(; f(x))dx = \sum_{i=0}^{N-1} f(x_i) = \sum_{i=0}^{N-1} \delta_{x_i}(; f(x)) = \sum_{i=0}^{N-1} \int_{\Omega} f(x)\delta(x - x_i)dx \tag{A.1.34}$$

where $\delta(x)$ is a dirac delta distribution

$$\int_\Omega \delta(x)dx = \begin{cases} 1 & \text{if } x = 0, \\ 0 & \text{otherwise} \end{cases} \tag{A.1.35}$$

and each δ_{x_i} is a dirac delta linear functional which performs point evaluation at each x_i

$$\delta_{x_i} : V \to \mathbb{R} \text{ where } \delta_{x_i}(; f) = x_i, \ \forall f \in V. \tag{A.1.36}$$

Note that δ_{x_i} cannot be represented in L^2. To get entirely expected behaviour when taking derivatives with respect to f, for example when formulating a functional to minimise then performing mesh refinement, we require $f \in H^1$ in 1 dimension or $f \in H^2$ for higher dimensions.

A.1.2 Details of the DMSwarm vertex-only mesh implementation

With reference to Lange et al. [47] we create a new mesh topology associated with the DMSwarm (a new `VertexOnlyMeshTopology` class). We instantiate a vertex-only mesh in Firedrake with a new `VertexOnlyMesh` constructor which associates this topology with geometric information, returning the usual MeshGeometry type used to represent Firedrake meshes.

At present, the VertexOnlyMesh constructor requires both a 'parent' mesh and list of coordinates at which to create vertices. We firstly store the list of coordinates of each vertex on the DMSwarm as a DMSwarm 'field'[3]. DMSwarm fields associate some piece of data with each entity (each vertex) in the DMSwarm (see the DMSwarm documentation in the PETSc manual for more information).

A full list of the DMSwarm fields used for Firedrake vertex-only meshes at the time of writing can be found in Sect. A.3.2.

A.1.3 PyOP2 Details for Interpolating onto a Vertex-Only Mesh

In step (c) of Sect. 4.5.3, we need to ensure that we only compute the kernel produced by TSFC where we have some $\hat{X}_i$ (i.e. once per vertex-only mesh cell X_i) and provide the appropriate global basis functions and basis function coefficients of $u(x)$ on the parent mesh cell. PyOP2 has three key data structures which are relevant here: `Maps`, `Sets` and `Dats`. `Sets` are equivalent to their mathematical counterparts: they are sets of entities, typically globally defined for a whole mesh such as mesh vertices or global basis function coefficients. `Maps` are mappings between `Sets` which can be used to control kernel execution. Imagine one has two `Sets` of entities A and

[3]One could consider this a pollution of the mesh topology with geometric information. However, all Firedrake meshes (of type MeshGeometry) require a mesh topology at creation whilst creation of a VertexOnlyMeshTopology class from a DMSwarm does not require this field or any of the subsequently mentioned DMSwarm fields. We can therefore consider this as an 'adding on' of extra information through a convenient feature of the implementation.

B between which we have constructed a `Map` m. For each entity $a \in A$, the `Map` m will provide us the associated subset of entities $b \in B$. If we make the `Set` A our *iteration set*, PyOP2 will run the TSFC kernel once for each entity $a \in A$: we can then use m to get associated values $b \in B$ which we provide to the kernel. These are laid out in detail in Rathgeber et al. [15].

We create a `Map` from the `Set` of vertex-only mesh vertices[4] to the `Set` of parent mesh nodes which we store on the vertex-only mesh. The latter correspond to the global basis functions and basis function coefficients of $u(x)$ on each parent mesh cell. Upon vertex-only mesh construction in Firedrake, we expose a `Map` from vertex-only mesh cell number to parent cell number using the previously identified Firedrake cell number for each $X_i \in \Omega_v$. Firedrake's implementation of finite element function spaces, `FunctionSpace`, and their `Function` members always expose a `Map` from mesh cell to associated nodes. When performing the interpolation in Firedrake, we therefore compose the two `Maps` using functionality that was specially written for this application but which has since been absorbed into PyOP2 itself.[5]

PyOP2 `Dats` are used to store data defined over an entire mesh such as the values of a `Function` in a `FunctionSpace`. Returning to step (b), we rely on the existing dual-evaluation interpolation functionality which packs the kernel output into the `Dat` of the previously created `Function`. We get a kernel output for each member of the iteration set of our `Map`, here the cells $X_i \in \Omega_v$, each of which is written to the appropriate location in the `Dat`. Here the previously created `Function` is in $\mathrm{P0DG}(\Omega_v)$: whilst the TSFC kernel produced values that can be considered as still being on the parent mesh, they are point evaluations of the correct shape and value so can be assigned directly to the corresponding vertex-only mesh cell X_i. PyOP2 provides structures for storing data that kernels produce (Dats) and for providing mappings between lists to dictate kernel execution (Maps). Details of these can be found in Rathgeber et al. [15]. To execute over each cell of the vertex-only mesh, but run on the correct cell of the parent mesh, a PyOP2 Map from vertex-only mesh cell number to parent mesh cell number is created[6] and is stored on the vertex-only mesh.

A.1.4 Runtime Tabulation with a FInAT `QuadratureElement`

Quadrature rules in FInAT use a FInAT `AbstractPointSet` to represent the points in the quadrature rule. This, along with the $\boldsymbol{Q}$ identity tensor, is produced by FInAT when requesting the `dual_basis` of the `QuadratureElement` (see Sect. 3.7). For runtime tabulation, we can use the `UnknownPointSet` created for this purpose (previously described in Sect. 3.8): we instantiate it with a GEM `Variable` representing a single point of shape $(1, \dim(\mathcal{K}))$ where $\dim(\mathcal{K})$ is the topological dimension of the reference cell. When TSFC compiles the kernel this

[4]the cell `Set`, available upon construction for all meshes

[5]In the case of extruded meshes, described in [48], a the vertex-only mesh has `Maps` from its cell `Set` to the parent mesh base cell number and extrusion height. We use these together to build the `Map` from vertex-only mesh cells to the nodes on the extruded parent mesh.

[6]In the case of extruded meshes, described in [48], a mapping from vertex-only mesh cell to base mesh cell and extrusion height is also required.

is left as a kernel argument which we can supply.

A.2 Appendix to Chapter 5

A.2.1 Finite Dimensional Gateaux Deriviative Example

In finite element function spaces we have a known set of global basis functions which allows us to write these derivatives as a finite-dimensional tensor operator multiplied by a linear perturbation. If, for example, $U := \mathbb{R}^U$ and $V := \mathbb{R}^V$ we have

$$u = \begin{pmatrix} u_0 \\ \vdots \\ u_{U-1} \end{pmatrix} \in \mathbb{R}^U \tag{A.2.1}$$

and

$$f(u) = \begin{pmatrix} f_0(u) \\ \vdots \\ f_{V-1}(u) \end{pmatrix} \in \mathbb{R}^V. \tag{A.2.2}$$

The Gateaux derivative is then (with reference back to the notation introduced in equation 5.2.10)

$$\begin{aligned} df_u(u; u') &= \underbrace{\begin{pmatrix} \frac{\partial f_0}{\partial u_0}\Big|_u & \cdots & \frac{\partial f_0}{\partial u_{U-1}}\Big|_u \\ \vdots & \ddots & \vdots \\ \frac{\partial f_{V-1}}{\partial u_0}\Big|_u & \cdots & \frac{\partial f_{V-1}}{\partial u_{U-1}}\Big|_u \end{pmatrix}}_{\in \mathbb{R}^V \times \mathbb{R}^U} \cdot \underbrace{\begin{pmatrix} u'_0 \\ \vdots \\ u'_{U-1} \end{pmatrix}}_{\in \mathbb{R}^U} \in \mathbb{R}^V & \text{(A.2.3)} \\ &:= \frac{\mathrm{d}f}{\mathrm{d}u}\bigg|_u \cdot u' & \text{(A.2.4)} \end{aligned}$$

where $\mathbb{R}^V \times \mathbb{R}^U$ is the Cartesian product of the two spaces. We now see that, in the above finite dimensional vector spaces, $\frac{\mathrm{d}f_i}{\mathrm{d}u_j}\big|_u$ is the commonly used definition of the Jacobian matrix $[J_f]_{ij}$ applied to u'_j.

Given that we typically deal with discretised equations where our Gateaux derivative is a finite dimensional tensor, it is useful to introduce some compact notation. Each element of our rank-1 tensor (i.e. a vector) is

$$\begin{aligned} df_u(u; u')_i &= \sum_{j=0}^{U-1} \frac{\partial f_i}{\partial u_j}\bigg|_u u'_j & \text{(A.2.5)} \\ &:= \frac{\partial f_i}{\partial u_j}\bigg|_u u'_j & \text{(A.2.6)} \\ &:= \left[\frac{\mathrm{d}f}{\mathrm{d}u}\bigg|_u\right]_{ij} u'_j & \text{(A.2.7)} \end{aligned}$$

where we have used the convention in equations A.2.6 and A.2.7 of summing over repeated indices. In equation A.2.7 $\frac{\mathrm{d}f}{\mathrm{d}u}\big|_u$ is our ordinary Gateaux derivative which we have discretised and supplied a direction u'_j. Since we have a function of a single argument, it is equal to to the partial Gateaux derivative $\partial f_u(u; u')_i :- [\frac{\partial f}{\partial u}\big|_u \cdot u']_i$.

A.2.2 Gateaux Derivative Chain Rule Example

This example demonstrates how each term in the chain rule can be broken down into simple uncoupled calculations which can then be substituted back together.

We start with

$$J(v) = \int v^2 \, dx \in \mathbb{R} \tag{A.2.8}$$

and let

$$[J \circ f](u) = \int f(u)^2 \, dx \in \mathbb{R} \tag{A.2.9}$$

where

$$f(u) = 2u^3 \in V. \tag{A.2.10}$$

We want

$$\left.\frac{\mathrm{d}[J \circ f]}{\mathrm{d}u}\right|_u \cdot u' \tag{A.2.11}$$

which we will find using equation 5.2.27. We substitute

$$v = f|_u \tag{A.2.12}$$

such that

$$J(v) = \int v^2 \, dx \tag{A.2.13}$$

and find the left-most term in our chain rule

$$\begin{aligned}\left.\frac{\mathrm{d}[J \circ f]}{\mathrm{d}u}\right|_u \cdot u' &= \left.\left(\frac{\mathrm{d}J}{\mathrm{d}v} \cdot v'\right)\right|_{v=f|_u} \\ &= \left.\left(\int 2v \, dx \cdot v'\right)\right|_{v=f|_u} \\ &= \int 2f|_u \, dx \cdot v'|_{v=f|_u}\end{aligned} \tag{A.2.14}$$

where we have used the linearity of the derivatives applied to integrals (which itself could be a step in the chain rule but here is skipped over!). We now find v' at $v = f|_u$, the right-most term in our chain rule:

$$\begin{aligned} v'|_{v=f|_u} &= \left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_u \cdot u' \\ &= 6u^2 \cdot u'. \end{aligned} \tag{A.2.15}$$

Substituting everything back we get

$$\left.\frac{\mathrm{d}[J \circ f]}{\mathrm{d}u}\right|_u \cdot u' = \int 2f|_u \, dx \cdot 6u^2 \cdot u'. \tag{A.2.16}$$

The expression for $f|_u$ is specifically not substituted in here since we are not trying to get a big unifying expression for $\left.\frac{\mathrm{d}[J\circ f]}{\mathrm{d}u}\right|_u \cdot u'$: we assume that we have already calculated $f|_u$ for some particular value of u and are propagating it through the chain rule from right to left. If we wanted to add another term to the chain rule such that we calculate

$$\left.\frac{\mathrm{d}[J \circ f \circ h]}{\mathrm{d}x}\right|_x \cdot x' = \left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_{v=f|_u} \cdot \underbrace{\left.\frac{\mathrm{d}f}{\mathrm{d}u}\right|_{u=h|_x} \cdot \underbrace{\left.\frac{\mathrm{d}h}{\mathrm{d}x}\right|_x \cdot x'}_{u'|_{u=h|_x}}}_{v'|_{v=f|_u}} \tag{A.2.17}$$

where

$$h : X \to U \tag{A.2.18}$$

for some particular $x \in X$ we would need to start by finding

$$u = h|_x \tag{A.2.19}$$

then propagate that information from the right of equation A.2.17 to the left by first calculating

$$u'|_{u=h|_x} = \left.\frac{\mathrm{d}h}{\mathrm{d}x}\right|_x \cdot x' \tag{A.2.20}$$

then calculating

$$\begin{aligned} v'|_{v=f|_u} &= \left.\left(\frac{\mathrm{d}f}{\mathrm{d}u} \cdot u'\right)\right|_{u=h|_x} \\ &= (6u^2 \cdot u')|_{u=h|_x}. \end{aligned} \tag{A.2.21}$$

In the notation of equations 5.2.26 and 5.2.34 this is the same as working from inner expressions to outer. It is this propagation of information from the right to the left (or inner to outer) that forms the basis of the *forward* or *tangent-linear* mode of Algorithmic/Automatic Differentiation (AD).

We *could* work from left to right but we would need to have access to pre-calculated values of $v = f|_u$ and $u = h|_x$ at each step. This is the basis of the *reverse* or *adjoint* mode[7] of AD where such values are saved from a pre-calculated running of whatever it is one wishes to differentiate (here $[J \circ f \circ h]$).

The term 'tangent-linear' is used since we take a derivative *at a chosen point x*: To take a

[7]sometimes called *cotangent linear* mode.

simple example, the derivative of the nonlinear function $f(x) = x^3$ is another nonlinear function $\frac{\mathrm{d}f}{\mathrm{d}x}(x) = 3x^2$. If I evaluate this at some point (such as $x = 1$) I get a value for the derivative at that point ($\frac{\mathrm{d}f}{\mathrm{d}x}\big|_1 = 3$) which is the gradient of a tangent to the original function at that point. In the multidimensional or gateaux derivative (without giving a direction x') cases this is the same as saying my gradient operator, here $\frac{\mathrm{d}[J \circ f \circ h]}{\mathrm{d}x}$, is a linear operator when I evaluate it at a chosen point (i.e. do $\frac{\mathrm{d}[J \circ f \circ h]}{\mathrm{d}x}\big|_x$).

A.2.3 The Adjoint Approach with the Alternative Adjoint Definition

In each case we need to look at our intermediate Jacobians evaluated at given points:

$$dJ_v(v; \bullet) : V \to \mathbb{R} \text{ for given } v \in V, \tag{A.2.22}$$

$$\partial f_u(u, g; \bullet) : U \to V \text{ for given } u \in U, g \in G, \tag{A.2.23}$$

$$\partial f_g(u, g; \bullet) : G \to V \text{ for given } u \in U, g \in G, \tag{A.2.24}$$

$$dh_m(m; \bullet) : M \to U \text{ for given } m \in M, \tag{A.2.25}$$

$$dk_m(m; \bullet) : M \to G \text{ for given } m \in M. \tag{A.2.26}$$

The $\bullet^\dagger$ adjoints of these (taken with respect to a particular inner product) are

$$dJ_v^\dagger(v; \bullet) : \mathbb{R} \to V \text{ for given } v \in V, \tag{A.2.27}$$

$$\partial f_u^\dagger(u, g; \bullet) : V \to U \text{ for given } u \in U, g \in G, \tag{A.2.28}$$

$$\partial f_g^\dagger(u, g; \bullet) : V \to G \text{ for given } u \in U, g \in G, \tag{A.2.29}$$

$$dh_m^\dagger(m; \bullet) : U \to M \text{ for given } m \in M, \tag{A.2.30}$$

$$dk_m^\dagger(m; \bullet) : G \to M \text{ for given } m \in M. \tag{A.2.31}$$

and similarly use these in the $\bullet^\dagger$ adjoints of our 'hat' operators

$$\hat{j}^\dagger(;\bullet) = I(;\bullet) : \mathbb{R} \to \mathbb{R} \;\; \forall j \in \mathbb{R} \tag{A.2.32}$$

$$\hat{v}^\dagger(;\bullet) = \partial J_v^\dagger\Big(v; \hat{j}^\dagger(;\bullet)\Big) : \mathbb{R} \to V \text{ for given } v \in V, \tag{A.2.33}$$

$$\hat{u}^\dagger(;\bullet) = \partial f_u^\dagger\Big(u, g; \hat{v}|^\dagger_{v=f|_{u,g}}(;\bullet)\Big) : \mathbb{R} \to U \text{ for given } u \in U, g \in G, \tag{A.2.34}$$

$$\hat{g}^\dagger(;\bullet) = \partial f_g^\dagger\Big(u, g; \hat{v}|^\dagger_{v=f|_{u,g}}(;\bullet)\Big) : \mathbb{R} \to G \text{ for given } u \in U, g \in G, \tag{A.2.35}$$

$$\hat{m}^\dagger(;\bullet) = dh_m^\dagger\Big(m; \hat{u}|^\dagger_{u=h|_m, g=k|_m}(;\bullet)\Big) + dk_m^\dagger\Big(m; \hat{g}|^\dagger_{u=h|_m, g=k|_m}(;\bullet)\Big) : \mathbb{R} \to M \text{ for given } m \in M. \tag{A.2.36}$$

The full $\bullet^\dagger$ adjoint Jacobian for our example is $\hat{m}^\dagger(;\bullet)$. Supplying any of these with a direction vector $j' \in \mathbb{R}$ gives a $\bullet^\dagger$ adjoint Jacobian vector product. As in the other derivation, to recover the Jacobian operator for our example we supply a seed of $j' = 1$ to $\hat{m}^\dagger$ and take the $\bullet^\dagger$ adjoint of $\hat{m}^\dagger(;1)$: i.e.

$$\left[\hat{m}^\dagger(;j'=1)\right]^\dagger \cdot \bullet = d[J \circ f \circ (h,k)]_m(m;\bullet) \tag{A.2.37}$$

and, again, were $J : V \to X$ and we had a complete basis for X then, by operating on each basis vector in turn, we would be able to recover the complete $\bullet^\dagger$ adjoint-Jacobian.

Our operations still break down conveniently giving an adjoint variable each time

$$\bar{j} = I(;\bar{j}) \in \mathbb{R} \tag{A.2.38}$$
$$\bar{v} = \partial J_v^\dagger(v;\bar{j}) \in V \text{ for given } v \in V, \tag{A.2.39}$$
$$\bar{u} = \partial f_u^\dagger(u,g;\bar{v}) \in U \text{ for given } u \in U, g \in G, \tag{A.2.40}$$
$$\bar{g} = \partial f_g^\dagger(u,g;\bar{v}) \in G \text{ for given } u \in U, g \in G, \tag{A.2.41}$$
$$\bar{m} = dh_m^\dagger(m;\bar{u}) + dk_m^\dagger(m;\bar{g}) \; (= \hat{m}^\dagger(;\bar{j})) \in M \text{ for given } m \in M. \tag{A.2.42}$$

These are the same as those in the main derivation but, in each case, have been transformed by Riesz representer for the particular inner product that gives the $\bullet^\dagger$ adjoint here. Alongside equations A.2.27 to A.2.31 these form a DAG which is shown pictorially in figure A.1.

In our alternative notation the $\bullet^\dagger$ adjoints of the intermediate Jacobians are

$$\left.\frac{\mathrm{d}J}{\mathrm{d}v}\right|_v^\dagger \cdot \bullet : \mathbb{R} \to V \text{ for given } v \in V, \tag{A.2.43}$$
$$\left.\frac{\partial f}{\partial u}\right|_{u,g}^\dagger \cdot \bullet : V \to U \text{ for given } u \in U, g \in G, \tag{A.2.44}$$
$$\left.\frac{\partial f}{\partial g}\right|_{u,g}^\dagger \cdot \bullet : V \to G \text{ for given } u \in U, g \in G, \tag{A.2.45}$$
$$\left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_m^\dagger \cdot \bullet : U \to M \text{ for given } m \in M, \tag{A.2.46}$$
$$\left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_m^\dagger \cdot \bullet : G \to M \text{ for given } m \in M. \tag{A.2.47}$$

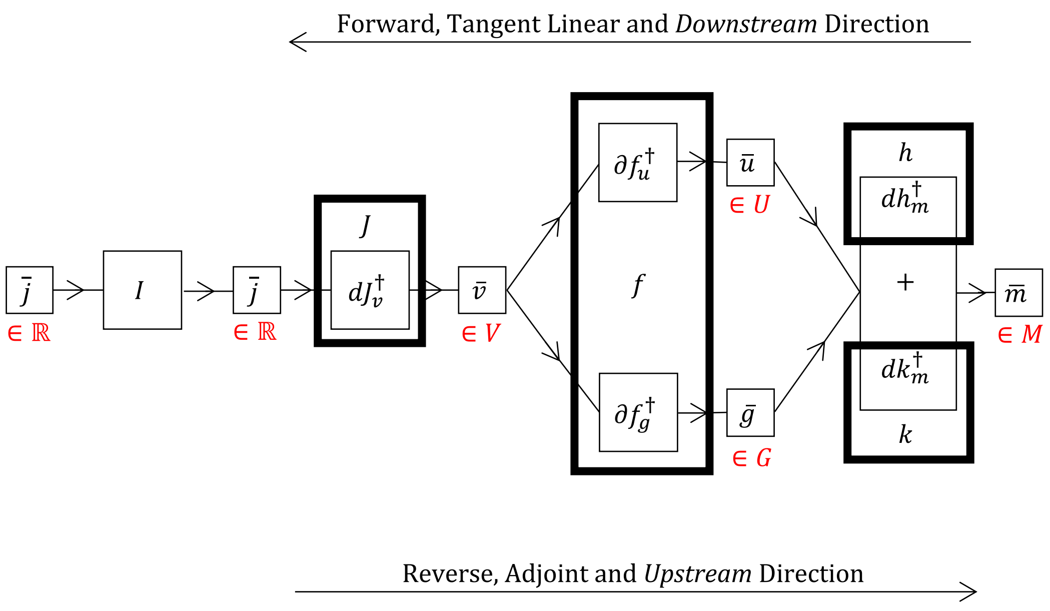


Figure A.1: Directed Acyclic Graph (DAG) for the operations given in equations A.2.27 to A.2.31; linked by the intermediate output variables in equations A.2.38 to A.2.42. Functions are in large boxes with input and output variables directly connected. In practice the initial identity operation is skipped. The *blocks* associated with each operation in pyadjoint which annotate each operation and allow derivatives to be calculated (see section 5.4) are shown as thick boxes. The vector spaces that variables are members of are shown below each set in red. This DAG is very similar to the original example and TLM DAGs (figures 5.1 and 5.2 respectively) but with arrow directions reversed. Once again we see that for each variable we have a corresponding adjoint variable. Operations are now Jacobian evaluations although not all the blocks have arrows pointing towards them which makes implementing this difficult. How this is fixed is discussed and fixed in section A.2.5.

The $\bullet^\dagger$ adjoints of our 'hat' operators are

$$\hat{j}^\dagger(;\bullet) = I^*(;\bullet) = I(;\bullet) : \mathbb{R} \to \mathbb{R} \;\; \forall j \in \mathbb{R} \tag{A.2.48}$$

$$\hat{v}^\dagger(;\bullet) = \left.\frac{\partial J}{\partial v}\right|_v^\dagger \cdot \hat{j}^\dagger(;\bullet) : \mathbb{R} \to V \text{ for given } v \in V, \tag{A.2.49}$$

$$\hat{u}^\dagger(;\bullet) = \left.\frac{\partial f}{\partial u}\right|_{u,g}^\dagger \cdot \hat{v}|^\dagger_{v=f|_{u,g}}(;\bullet) : \mathbb{R} \to U \text{ for given } u \in U, g \in G, \tag{A.2.50}$$

$$\hat{g}^\dagger(;\bullet) = \left.\frac{\partial f}{\partial g}\right|_{u,g}^\dagger \cdot \hat{v}|^\dagger_{v=f|_{u,g}}(;\bullet) : \mathbb{R} \to G \text{ for given } u \in U, g \in G, \tag{A.2.51}$$

$$\hat{m}^\dagger(;\bullet) = \left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_m^\dagger \cdot \hat{u}|^\dagger_{u=h|_m, g=k|_m}(;\bullet) + \left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_m^\dagger \cdot \hat{g}|^\dagger_{u=h|_m, g=k|_m}(;\bullet) : \mathbb{R} \to M \text{ for given } m \in M, \tag{A.2.52}$$

we get the full Jacobian for example by

$$\left[\hat{m}^\dagger(; j' = 1)\right]^\dagger \cdot \bullet = \left.\frac{\mathrm{d}[J \circ f \circ (h,k)]}{\mathrm{d}m}\right|_m \cdot \bullet, \tag{A.2.53}$$

and the intermediate vectors (adjoint variables) for downstream-most seed $\bar{j}$ (which we use to calculate the full Jacobian) are

$$\bar{j} = I(;\bar{j}) \in \mathbb{R} \tag{A.2.54}$$

$$\bar{v} = \left.\frac{\partial J}{\partial v}\right|_v^\dagger \cdot \bar{j} \in V \text{ for given } v \in V, \tag{A.2.55}$$

$$\bar{u} = \left.\frac{\partial f}{\partial u}\right|_{u,g}^\dagger \cdot \bar{v} \in U \text{ for given } u \in U, g \in G, \tag{A.2.56}$$

$$\bar{g} = \left.\frac{\partial f}{\partial g}\right|_{u,g}^\dagger \cdot \bar{v} \in G \text{ for given } u \in U, g \in G, \tag{A.2.57}$$

$$\bar{m} = \left.\frac{\mathrm{d}h}{\mathrm{d}m}\right|_m^\dagger \cdot \bar{u} + \left.\frac{\mathrm{d}k}{\mathrm{d}m}\right|_m^\dagger \cdot \bar{g} \;(= \hat{m}^\dagger(;\bar{j})) \in M \text{ for given } m \in M. \tag{A.2.58}$$

A.2.4 Discretisations of TLM and Adjoint Action

This can be read as continuing from the end of section 5.4.2. If our block f had a second output variable $x \in X$ (i.e. $f(u,g) \in X \times V$ is a matrix valued function) then the `evaluate_tlm_component` method involving $\dot{x} \in X$ for finite dimensional vector spaces would be

$$(\dot{x}_l, \dot{v}_z)(u_k, g_j; \dot{u}_k, \dot{g}_j) = \sum_{k=1}^{\dim(U)} \sum_{j=1}^{\dim(G)} \partial f_u(u_k, g_j; u'_k = \dot{u}_k)_{lz} + \partial f_g(u_k, g_j; g'_k = \dot{g}_j)_{lz} \tag{A.2.59}$$

$$:= \sum_{k=1}^{\dim(U)} \sum_{j=1}^{\dim(G)} \left[\frac{\partial f}{\partial u}\bigg|_{u,g} \right]_{lzkj} \cdot \dot{u}_k + \left[\frac{\partial f}{\partial u}\bigg|_{u,g} \right]_{lzkj} \cdot \dot{g}_j \tag{A.2.60}$$

where $(\dot{x}_l, \dot{v}_z)$ are entries in the matrix formed by $X \times V$. For adjoint-inputs $\bar{v} \in V$ and $\bar{x} \in X$, the `evaluate_adj_component` method would give be

$$\bar{u}^*_{1l}(; \bullet) = \sum_{k=1}^{\dim(V)} \sum_{j=1}^{\dim(X)} \bar{v}^* \big(; \partial f_u(u, g; \bullet)_{kl}\big)_{1k} + \bar{x}^* \big(; \partial f_u(u, g; \bullet)_{jl}\big)_{1j} \text{ for given } u \in U, g \in G \tag{A.2.61}$$

$$:= \sum_{k=1}^{\dim(V)} \sum_{j=1}^{\dim(X)} \bar{v}^*_{1k} \cdot \left[\frac{\partial f}{\partial u}\bigg|_{u,g} \right]_{kl} \cdot \bullet + \bar{x}^*_{1j} \cdot \left[\frac{\partial f}{\partial u}\bigg|_{u,g} \right]_{jl} \cdot \bullet \tag{A.2.62}$$

and

$$\bar{g}^*_{1l}(; \bullet) = \sum_{k=1}^{\dim(V)} \sum_{j=1}^{\dim(X)} \bar{v}^* \big(\partial f_g(u, g; \bullet)_{kl}\big)_{1k} + \bar{x}^* \big(\partial f_g(u, g; \bullet)_{jl}\big)_{1j} \text{ for given } u \in U, g \in G \tag{A.2.63}$$

$$:= \sum_{k=1}^{\dim(V)} \sum_{j=1}^{\dim(X)} \bar{v}^*_{1k} \cdot \left[\frac{\partial f}{\partial g}\bigg|_{u,g} \right]_{kl} \cdot \bullet + \bar{x}^*_{1j} \cdot \left[\frac{\partial f}{\partial g}\bigg|_{u,g} \right]_{jl} \cdot \bullet. \tag{A.2.64}$$

A.2.5 Adjoint Mode Subtlety

When executing a program we sometimes have loops where we write to a variable multiple times - these are so-called *program variables.* This is incompatible with a Directed *Acyclic* Graph (DAG) since this would introduce cycles. In order to fully tape a program we need to be able to recreate all variables, including those which have been overwritten. We hence introduce the concept of *forward block variables* which, when recording or re-running a tape, can only be written to once but can be read multiple times. Each time a program variable is written to it creates a new forward block variable. It is the *tape* therefore, not the original program, which is a DAG.

As mentioned previously TLM variables are block variables, more specifically they are *forward* block variables: they are also written to exactly once, in this case by a block which annotates our program, but can be read multiple times by downstream blocks. Each block has a well defined way of creating a TLM variable which is clearly given by the multivariate chain rule:

1. Initialise the TLM variable to zero,
2. pick a forward block variable argument (pictorially, an arrow pointing into the block),
3. take the block's Jacobian with respect to that argument,
4. apply the Jacobian to the argument's corresponding TLM variable (which is guaranteed to have already been calculated upstream),
5. add the result to the TLM variable and
6. return to step 2 until all arguments have been used.

In the adjoint mode we traverse our tape from downstream to up: i.e. we reverse all the arrows in the tape's DAG. We therefore introduce the concept of *reverse block variables* which have the opposite properties of forward block variables: they can be written to multiple times but can only be read once. Adjoint variables are reverse block variables. Since we are using the same DAG, just in reverse, we associate reverse block variables with forward block variables: in pyadjoint this is a `BlockVariable` type.

In adjoint mode, at the level of a block, pyadjoint

1. finds the partial derivative Jacobian of its operation with respect to a particular forward block variable argument
2. takes an adjoint variable received from downstream[8] and,
3. takes the adjoint of the partial derivative Jacobian and applies it to the adjoint variable.

A new adjoint variable is created by summing together the outputs of this process for the relevant blocks (see, for example, the creation of $\bar{m}^*$ in 5.3.87).

This works from the point of view of creating a working AD tool, but the summation notably does not happen at block level, instead it relies on some global knowledge of the need to perform a sum. If we wish to consider the tape as being composed purely of blocks, an interesting question to ask is whether we can describe the creation of adjoint variables at the level of a block.

We can do this by limiting each forward or reverse block variable in the DAG of the tape to having a single arrow leaving it. Each block's adjoint variable is then found by performing the three steps above. To ensure each block variable only has a single arrow leaving it we need to introduce a hidden operation which distributes the forward block variables. In our example DAG this is an assignment operation which occurs after our variable m but before operations h and k

$$\text{stack}(m) = \begin{pmatrix} m \\ m \end{pmatrix} = \begin{pmatrix} m_0 \\ m_1 \end{pmatrix}. \tag{A.2.65}$$

[8]At this point the downstream adjoint variable could be deleted: this is not done in pyadjoint at present but is an open issue as a potential enhancement. See https://github.com/dolfin-adjoint/pyadjoint/issues/84.

Now our DAG is defined to be

$$J : V \to \mathbb{R} \text{ i.e. } J(v) \in \mathbb{R}, \tag{A.2.66}$$

$$f : U \times G \to V \text{ i.e. } f(u,g) \in V, \tag{A.2.67}$$

$$h : M \to U \text{ i.e. } h(m_0) \in U, \tag{A.2.68}$$

$$k : M \to G \text{ i.e. } k(m_1) \in G, \tag{A.2.69}$$

$$\text{stack} : M \to M \times M \text{ i.e. } \text{stack}(m) \in M \times M, \tag{A.2.70}$$

where

$$j = J|_v \tag{A.2.71}$$

$$v = f|_{(u,g)} \tag{A.2.72}$$

$$u = h|_{m_0} \tag{A.2.73}$$

$$g = k|_{m_1} \tag{A.2.74}$$

$$\begin{pmatrix} m_0 \\ m_1 \end{pmatrix} = \text{stack}|_m. \tag{A.2.75}$$

This is shown pictorially in figure A.2.

Each of m_0 and m_1 has an associated adjoint covector variable given by

$$\bar{m}_0^* = \partial h_{m_0}^*(m_0; \bar{u}^*) \tag{A.2.76}$$

$$\bar{m}_1^* = \partial k_{m_1}^*(m_1; \bar{g}^*). \tag{A.2.77}$$

The Gateaux derivative of stack with respect to m is

$$d\text{stack}_m : M \times \underbrace{M}_{\text{linear}} \to M \times M \tag{A.2.78}$$

where

$$d\text{stack}_m(m; m') := \frac{\partial}{\partial m} \left[\begin{pmatrix} m \\ m \end{pmatrix} \right] \Bigg|_m \cdot m' \tag{A.2.79}$$

$$= \begin{pmatrix} 1 \\ 1 \end{pmatrix} \cdot m' \in M \times M. \tag{A.2.80}$$

The ‘Jacobian’ linear operator is

$$d\text{stack}_m(m; \bullet) = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \cdot \bullet : M \to M \times M \text{ for given } m \in M. \tag{A.2.81}$$

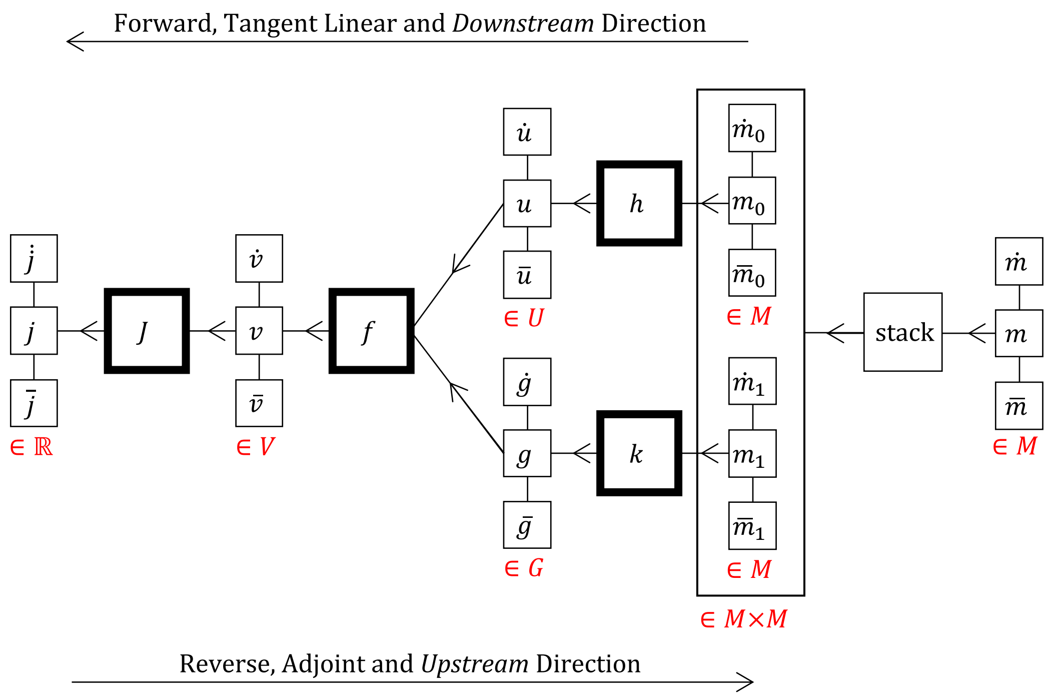


Figure A.2: Directed Acyclic Graph (DAG) for the operations given in equations A.2.66 to A.2.70 linked by the intermediate output variables in equations A.2.71 to A.2.75. Functions are in large boxes with input and output variables directly connected. Associated Tangent Linear Mode (TLM) variables are shown with dots above, whilst adjoint mode variables (in primal form) have bars above. The vector spaces that variables are members of are shown below each set in red.

For the TLM mode we supply this with $\dot{m}$ and get two copies of $\dot{m}$ out

$$\begin{pmatrix}\dot{m}_0\\ \dot{m}_1\end{pmatrix} = d\text{stack}_m(m;\dot{m}) = \begin{pmatrix}1\\1\end{pmatrix}\cdot m = \begin{pmatrix}\dot{m}\\ \dot{m}\end{pmatrix} \in M\times M \text{ for given } m\in M \tag{A.2.82}$$

which shows us that the TLM mode calculations are the same and continue to share the form of the DAG for the original calculation. For completeness the TLM DAG is shown in figure A.3.

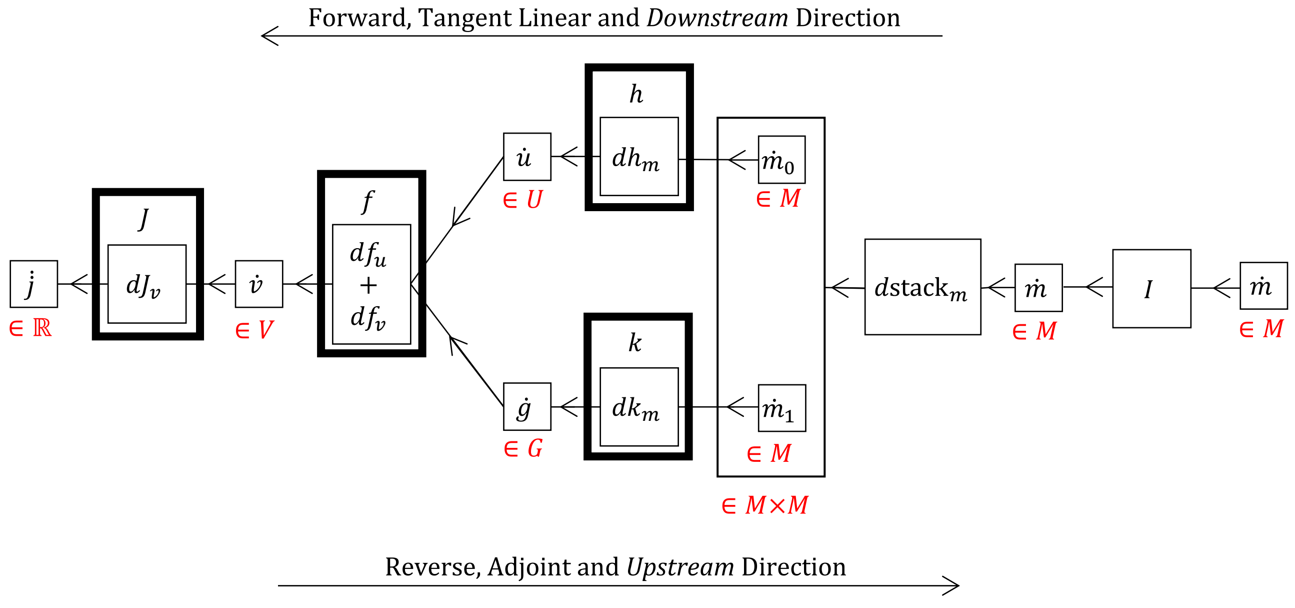


Figure A.3: Directed Acyclic Graph (DAG) for the tangent linear mode of AD applied to the operations given in equations A.2.66 to A.2.70 updated from figure 5.3 to include the new stack operation. Functions are in large boxes with input and output variables directly connected. In practice the initial identity operation is skipped. The *blocks* associated with each operation in pyadjoint which annotate each operation and allow derivatives to be calculated (see section 5.4) are shown as thick boxes. The vector spaces that variables are members of are shown below each set in red.

For the adjoint mode we have

$$d\text{stack}^*_m(m;\bullet) = \bullet\cdot\begin{pmatrix}1\\1\end{pmatrix} : (M\times M)^* \to M^* \text{ for given } m\in M \tag{A.2.83}$$

where

$$(M\times M)^* = M^*\times M^*. \tag{A.2.84}$$

The covector this operates on is

$$\begin{pmatrix}\bar{m}_0\\ \bar{m}_1\end{pmatrix}^* = \begin{pmatrix}\bar{m}_0^* & \bar{m}_1^*\end{pmatrix} \in (M\times M)^* \text{ for given } m\in M. \tag{A.2.85}$$

which gives our adjoint variable covector $\bar{m}^*$

$$\bar{m}^* = d\text{stack}^*_m(m; \begin{pmatrix} \bar{m}^*_0 & \bar{m}^*_1 \end{pmatrix}) = \begin{pmatrix} \bar{m}^*_0 & \bar{m}^*_1 \end{pmatrix} \cdot \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \bar{m}^*_0 + \bar{m}^*_1 \in M^* \tag{A.2.86}$$

which is our expression for $\bar{m}^*$ in equation 5.3.87.

The stack operation hasn't got a block in pyadjoint. Nevertheless imagining its existence shows us what to do with adjoint variables at the block level: for each forward block variable argument a block has we *add* our adjoint variable to the associated reverse block variable.

The updated adjoint mode DAG for approach (2) is shown in figure A.4: notice h and k blocks now have single inputs and no longer overlap with an addition operation. Instead it can be seen that the addition takes place in $d\text{stack}^*_m$.

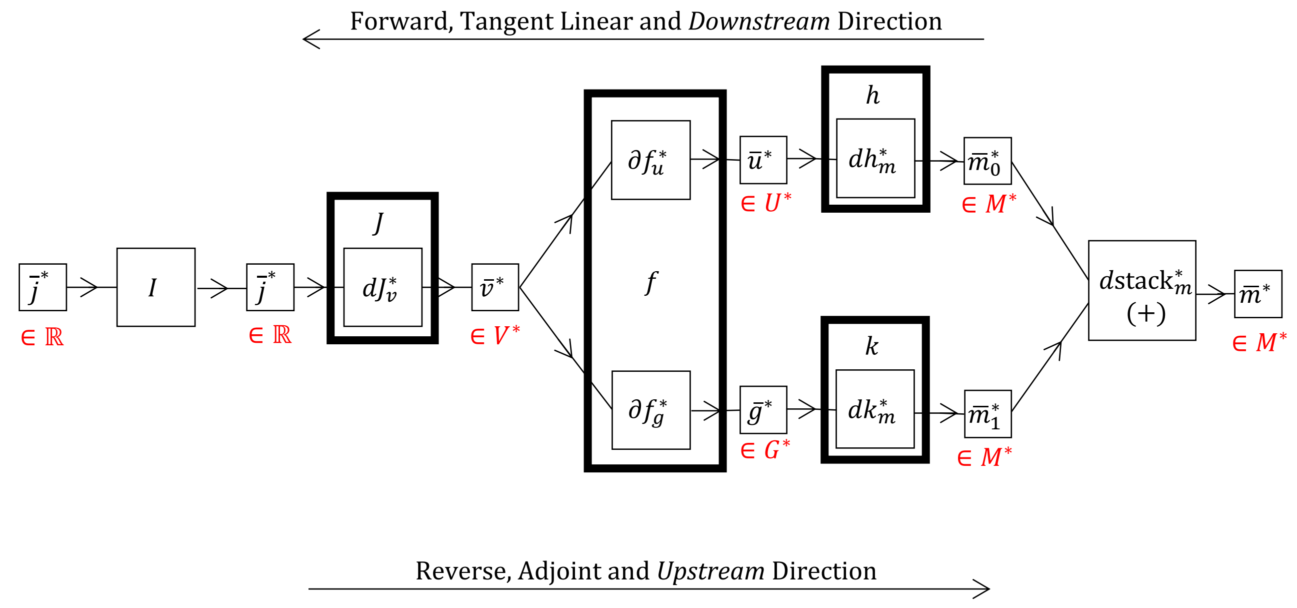


Figure A.4: Directed Acyclic Graph (DAG) for the adjoint mode of AD (via approach (2) outlined in section 5.3.2) applied to the operations given in equations A.2.66 to A.2.70 updated from figure 5.3 to include the new stack operation. Functions are in large boxes with input and output variables directly connected. In practice the initial identity operation is skipped. The *blocks* associated with each operation in pyadjoint which annotate each operation and allow derivatives to be calculated (see section 5.4) are shown as thick boxes. The vector spaces that variables are members of are shown below each set in red.

A.3 Appendix to Chapter 7

A.3.1 Calculating L^1 distances from Barycentric Coordinates

This section draws on text and diagrams in the API documentation I wrote for the FIAT library where this was implemented.

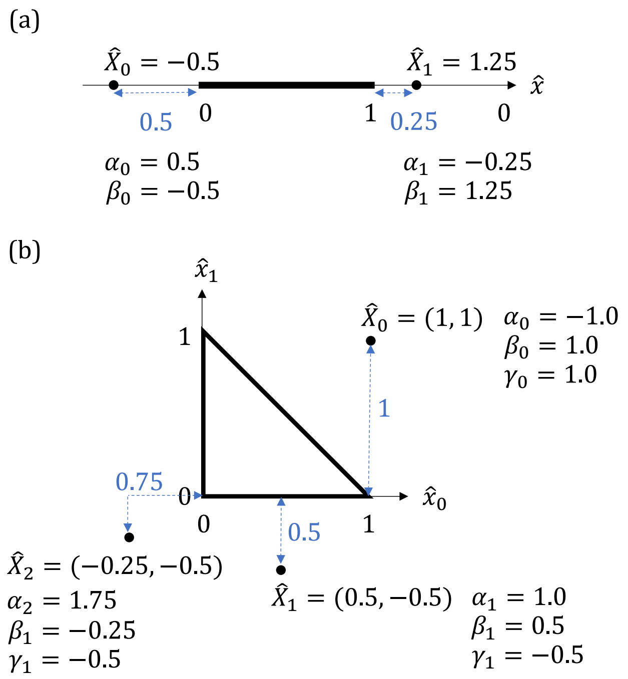


Figure A.5: Barycentric coordinates and L^1 distances on a reference interval (a) and triangle (b).

The L^1 distances of points from the reference cell, in the reference cell (local) coordinates system, are calculated with the help of Barycentric coordinates. These exist specifically to identify the relationship of point locations relative to simplices so that one can tell whether a point is inside or outside the simplex.

Simplices in 1D, 2D and 3D are the interval, the triangle, and the tetrahedron. In Firedrake, the reference interval has vertices $(0, 1)$ and a scalar reference coordinate $\hat{x}$. The two corresponding Barycentric coordinates α and β are defined in terms of the vertices $P_0 = 0$ and $P_1 = 1$ as

$$\hat{x} = \alpha P_0 + \beta P_1 \text{ where} \tag{A.3.1}$$

$$\alpha + \beta = 1. \tag{A.3.2}$$

The solution is

$$\alpha = 1 - \hat{x} \tag{A.3.3}$$

$$\beta = \hat{x}. \tag{A.3.4}$$

If both α and β are positive, the point is inside the reference interval. If either are negative for some $\hat{x}$, we are outside the interval and the L^1 distance[9] to the interval is the absolute value of the negative Barycentric coordinate.

The reference triangle has vector coordinates $\hat{x}$ and vertices

$$P_0 = (0,0), \tag{A.3.5}$$
$$P_1 = (1,0) \text{ and} \tag{A.3.6}$$
$$P_2 = (0,1) \tag{A.3.7}$$

(see Fig. A.5 (b)). There are now 3 Barycentric coordinates defined as

$$(\hat{x}_0, \hat{x}_1) = \alpha \times P_0 + \beta \times P_1 + \gamma * P_2 \text{ where} \tag{A.3.8}$$
$$\alpha + \beta + \gamma = 1.0 \tag{A.3.9}$$

which has solution

$$\alpha = 1 - \hat{x}_0 - \hat{x}_1 \tag{A.3.10}$$
$$\beta = \hat{x}_0 \text{ and} \tag{A.3.11}$$
$$\gamma = \hat{x}_1. \tag{A.3.12}$$

Once again, if any of these are negative for an $\hat{x}$, the point is outside the cell. This time the absolute sum of any negative Barycentric coordinates is the L^1 distance.

This is also true for the reference tetrahedron with vertex coordinates

$$P_0 = (0,0,0), \tag{A.3.13}$$
$$P_1 = (1,0,0), \tag{A.3.14}$$
$$P_2 = (0,1,0) \text{ and} \tag{A.3.15}$$
$$P_3 = (0,0,1). \tag{A.3.16}$$

where we have four Barycentric coordinates defined as

$$\alpha = 1 - \hat{x}_0 - \hat{x}_1 - \hat{x}_2 \tag{A.3.17}$$
$$\beta = \hat{x}_0, \tag{A.3.18}$$
$$\gamma = \hat{x}_1, \text{ and} \tag{A.3.19}$$
$$\delta = \hat{x}_2. \tag{A.3.20}$$

Once again, the absolute sum of any negative Barycentric coordinates is the L^1 distance.

For these three simplices we can now calculate the L^1 distance with a common method on

[9] which is identical to the l_2 or Euclidean distance

the base UFCSimplex FIAT class, from which all Firedrake simplices are subclassed. Given some point coordinate in the form $(\hat{x}_0, ...\hat{x}_{\text{dim}-1})$ I have implemented[10] the following:

```
1  def distance_to_point_l1(point)
2      # bary = [alpha, beta, gamma, delta, ...]
3      bary = [1.0 - sum(point)] + list(point)
4      # remove positive Barycentric coordinates. This doubles
5      # any negative Barycentric coordinates
6      no_pos_bary_double = bary-abs(bary)
7      no_pos_bary = 0.5 * no_pos_bary_double
8      # calculate L1 distance
9      l1_dist = abs(sum(no_pos_bary))
10     return l1_dist
```

Line 2 gives a Python list of the Barycentric coordinates, the first of which, α, always contains the sum term. To calculate the L^1 distance we could branch (for example with an `if` statement) to identify and sum the negative Barycentric coordinates. However, we need to be able to calculate this with a symbolic point in order to generate code. This cannot be done if the code has branches. Instead we calculate it algebraically, as shown in the code.

For example, if we have a point $(-1, -1)$ on the reference triangle. The L^1 distance from the triangle is 2.0. The Barycentric coordinates, from equations A.3.11 to A.3.12 are

```
# bary = [alpha, beta, gamma]
bary = [3, -1, -1]
```

Now we find the L^1 distance:

```
# abs(bary) = [3, 1, 1]
no_pos_bary_double = bary-abs(bary)  # [0, -2, -2]
no_pos_bary = 0.5 * no_pos_bary_double  # [0 -1, -1]
# sum(no_pos_bary) = -2.0
l1_dist = abs(sum(no_pos_bary))  # 2.0
```

Firedrake also supports quadrilaterals and hexahedra which are not simplices but are built as Cartesian products of the reference interval. In these cases, one calculates the L^1 distance from the reference interval for each dimension of $\hat{x}$, then sums the results. For example, the reference quadrilateral has vertices

$$P_0 = (0, 0), \tag{A.3.21}$$
$$P_1 = (1, 0), \tag{A.3.22}$$
$$P_2 = (0, 1) \text{ and} \tag{A.3.23}$$
$$P_3 = (1, 1) \tag{A.3.24}$$

[10]though less verbosely than presented here

and is made up of the cartesian product of two reference intervals $(0, 1)$. This is the cell upon which a quadrilateral tensor product element can be made: see Fig. 3.2. The point $\hat{x} = (1.5, -1.0)$ is 1.5 away from the cell as measured in L^1. $\hat{x}_0 = 1.5$ is 0.5 away from the reference interval whilst $\hat{x}_1 = -1.0$ is 1.0 away from the reference interval. Their sum, 1.5, is the L^1 distance.

A.3.2 Summary of DMSwarm 'Fields' at Time of Writing

In all, each point has the following pieces of information attached, each of which is embedded in a DMSwarm field:

- `DMSwarmPIC_coor`: the Cartesian coordinates of the point,
- `parentcellnum`: the parent-mesh cell number within the local mesh partition (each rank's partition has its own cell numbering) as identified by Firedrake,
- `refcoord`: within the identified parent-mesh cell, the point's location after transformation to the reference cell local coordinate system,
- `DMSwarm_rank`: the MPI rank where the identified parent-mesh cell can be found (for cells at boundaries between partitions, this is the unique non-halo rank),
- `DMSwarm_cellid`: similar to `parentcellnum`, but using PETSc's DMPlex numbering as identified by `DMSwarmSetPointCoordinates` when creating the DMSwarm,
- `globalindex`: the global index (a unique coordinate identifier),
- `inputrank`: the input rank and
- `inputindex`: the input index.

To support parent-meshes which are semi-structured Firedrake 'extruded meshes' [48], we also store

- `parentcellbasenum`: the base-cell number corresponding to the given `parentcellnum` and
- `parentcellextrusionheight`: the extrusion height relative to the base cell number.

A.4 Appendix to Chapter 8

```python
u_bot_old = Function(V_bot)
u_top_old = Function(V_top)
penalty_weight = 10.0  # Chosen experimentally to move the solutions together

for i in range(5000):
    # Helmholtz problem on the top mesh
    u_top = Function(V_top)
    v = TestFunction(V_top)
    f_top = Function(V_top)
    x, y = SpatialCoordinate(m_top)
    f_top.interpolate((1 + 8 * pi * pi) * cos(x * pi * 2) * cos(y * pi * 2))
    # Couple poisson to helmholtz with interpolate (will do nothing on
    # first iteration)
    u_bot_in_V_top = Function(V_top).interpolate(u_bot, allow_missing_dofs=True)
    # penalty moves our solutions together on the interface
    penalty = inner(u_bot_in_V_top - u_top, v) * ds(3)
    # surface term must be included on the interface, where it is nonzero
    surface_term = inner(grad(u_top), FacetNormal(m_top)) * v * ds(3)
    F = (
        (inner(grad(u_top), grad(v)) + inner(u_top, v)) * dx
        - inner(f_top, v) * dx - surface_term - penalty_weight * penalty
    )
    solve(F == 0, u_top)

    # Poisson problem on the bottom mesh
    u_bot = Function(V_bot)
    v = TestFunction(V_bot)
    # Couple helmholtz to poisson with interpolate
    u_top_in_V_bot = Function(V_bot).interpolate(u_top, allow_missing_dofs=True)
    # penalty moves our solutions together on the interfaces
    penalty = inner(u_top_in_V_bot - u_bot, v) * ds(4)
    # surface term must be included on the interface, where it is nonzero
    surface_term = inner(grad(u_bot), FacetNormal(m_bot)) * v * ds(4)
    # We hold our solution to zero on the other boundaries
    bc = DirichletBC(V_bot, 0, [1, 2, 3])
    F = (
        (inner(grad(u_bot), grad(v)) - f_bot * v) * dx
        - surface_term - penalty_weight * penalty
    )
    solve(F == 0, u_bot, bc)

    # Check for convergence
    if np.allclose(u_bot.dat.data_ro, u_bot_old.dat.data_ro) and np.allclose(
        u_top.dat.data_ro, u_top_old.dat.data_ro
    ):
        break
    u_bot_old = u_bot.copy(deepcopy=True)
    u_top_old = u_top.copy(deepcopy=True)
```

Listing 19: Two-way coupling of a Helmholtz problem to a Poisson problem. This code should be read as following directly on from the code in Listing 17. The `weak_penalty_bc` expression introduced to the variational forms of each problem is 8.4.16.

```python
 1 from firedrake import *
 2 from mpi4py import MPI
 3 import numpy as np
 4 
 5 comm = MPI.COMM_WORLD
 6 if comm.size < 2:
 7     raise RuntimeError("This example requires at least two MPI ranks")
 8 
 9 # (1) Here are some data points distributed across two MPI ranks
10 if comm.rank == 0:
11     data_coords = np.array([[0.0, 0.0], [0.0, 0.5], [0.0, 1.0], [0.5, 0.0],
12                             [0.5, 0.5], [0.5, 1.0], [1.0, 0.0], [1.0, 0.5]])
13     data_vals = np.array([2.0, 2.0, 3.0, 1.0, 5.0, 6.0, -1.0, -2.0])
14 elif comm.rank == 1:
15     data_coords = np.array([[1.0, 1.0]])
16     data_vals = np.array([0.0])
17 else:
18     data_coords = np.array([]).reshape(0, 2)
19     data_vals = np.array([])
20 
21 # (2) A mesh which has vertices at the data coordinates.
22 omega_data = UnitSquareMesh(2, 2, comm=comm)
23 
24 # (3) Create a vertex-only mesh with the data coordinates
25 omega_v = VertexOnlyMesh(omega_data, data_coords, redundant=False)
26 
27 # (4) Input the data, using the input_ordering property for parallel safety
28 P0DG_input_ordering = FunctionSpace(omega_v.input_ordering, "DG", 0)
29 f_v_input_ordering = Function(P0DG_input_ordering)
30 f_v_input_ordering.dat.data_wo[:] = data_vals  # This is now safe to do
31 P0DG = FunctionSpace(omega_v, "DG", 0)
32 f_v = interpolate(f_v_input_ordering, P0DG)
33 
34 # (5) Create a function space on m which has point evaluation nodes at
35 # the mesh vertices and interpolate from f_v with a permutation matrix
36 V = FunctionSpace(omega_data, "CG", 1)
37 I = Interpolator(TestFunction(V), P0DG)
38 # NOTE: we need our inputs and outputs to be cofunctions, since we use
39 # transpose interpolation as the inverse. If we choose l2 as our innter product
40 # for the riesz representation we retain the values of the cofunction
41 # coefficients as the function coefficients.
42 f_v_star = f_v.riesz_representation(riesz_map="l2")
43 f_data_star = Cofunction(V.dual())
44 I.interpolate(f_v_star, transpose=True, output=f_data_star)
45 f_data = f_data_star.riesz_representation(riesz_map="l2")
46 
47 # (6) Optionally interpolate to a different mesh and function space
48 m2 = UnitSquareMesh(3, 3, quadrilateral=True)
49 V2 = FunctionSpace(m2, "CG", 2)
50 f2 = interpolate(f_data, V2)
```

Listing 20: Generic external point data input, as described in Sect. 8.4.2. Note that on line 42 the Riesz representation could also be `'L2'` since P0DG(Ω_v) is 'self-dual' in the L^2 inner product, i.e. cofunction basis coefficients are the transpose-conjugate of function basis coefficients and I am using real-valued basis functions.

```python
from firedrake import *
import numpy as np

omega = UnitSquareMesh(40, 40)
U = FunctionSpace(omega, "CG", 1)

# Create line (0, 0), (1, 1) mesh with 10 cells
cells = np.asarray(
    [[0, 1], [1, 2], [2, 3], [3, 4], [4, 5], [5, 6], [6, 7], [7, 8], [8, 9], [9, 10]]
)
vertex_coords = np.asarray(
    [
        [0.0, 0.0],
        [0.1, 0.1],
        [0.2, 0.2],
        [0.3, 0.3],
        [0.4, 0.4],
        [0.5, 0.5],
        [0.6, 0.6],
        [0.7, 0.7],
        [0.8, 0.8],
        [0.9, 0.9],
        [1.0, 1.0],
    ]
)
tdim = 1  # topological dimension
gdim = 2  # geometric dimension
dmplex = mesh.plex_from_cell_list(tdim, cells, vertex_coords, omega.comm)
line = mesh.Mesh(dmplex, dim=gdim)

# RHS line forcing cofunction with 41 point evaluation nodes
P4CG = FunctionSpace(line, "CG", 4)
x, y = SpatialCoordinate(line)
f_l = Function(P4CG).interpolate(x * y)
I = Interpolator(TestFunction(U), P4CG)
v_l = TestFunction(P4CG)
L_l = assemble(f_l * v_l * dx)
L = Cofunction(U.dual())
I.interpolate(L_l, output=L, transpose=True)

# LHS Bilinear form
u = TrialFunction(U)
v = TestFunction(U)
a = inner(grad(u), grad(v)) * dx
# We hold our solution to zero on x=0 and y=0
bc = DirichletBC(U, 0, [1, 3])
u_sol = Function(U)  # solution will be stored here
solve(a == L, u_sol, bc)
```

Listing 21: Forcing a Poisson problem with point sources at point-evaluation node locations on a line mesh. The mathematical description is in Sect. 8.4.3, with the particular problem in Eq. 8.4.30. For results see Fig. 8.10.