Firedrake

User Manual

David Ham, Paul Kelly, Lawrence Mitchell, Colin Cotter, Rob Kirby, Koki Sagiyama, Nacime Bouziani, Sophia Vorderwuelbecke, Tom Gregory, Jack Betteridge, Daniel Shapero, Reuben Nixon-Hill, Connor Ward, Thomas Gibson, Miklós Homolya, Tianjiao (TJ) Sun, Andrew McRae, Fabio Luporini, Alastair Gregory, Michael Lange, Simon Funke, Florian Rathgeber, Doru Bercea, Graham Markall

First edition: 2023

CONTENTS

INTRODUCTION

Firedrake is an automated system for the solution of partial differential equations using the finite element method (FEM). Firedrake uses sophisticated code generation to provide mathematicians, scientists, and engineers with a very high productivity way to create sophisticated high performance simulations.

0.1 Features:

- Expressive specification of any PDE using the Unified Form Language from the FEniCS Project.
- Sophisticated, programmable solvers through seamless coupling with PETSc.
- Triangular, quadrilateral, and tetrahedral unstructured meshes.
- Layered meshes of triangular wedges or hexahedra.
- Vast range of finite element spaces.
- Sophisticated automatic optimisation, including sum factorisation for high order elements, and vectorisation.
- Geometric multigrid.
- Customisable operator preconditioners.
- Support for static condensation, hybridisation, and HDG methods.

0.2 The Firedrake team

Firedrake is brought to you by

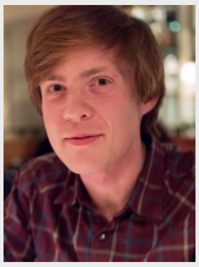
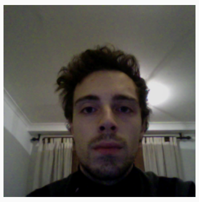
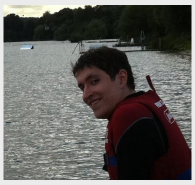
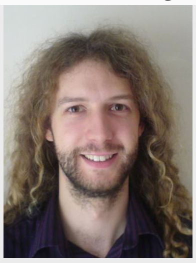
- Imperial College London
  - Department of Mathematics
  - Department of Computing
  - Department of Earth Science and Engineering
- Durham University
  - Department of Computer Science
- Baylor University
  - Department of Mathematics
- University of Washington
  - Applied Physics Laboratory

and the broad community of Firedrake users who contribute to its development.

0.2.1 Active team members

David Ham	Paul Kelly	Lawrence Mitchell	Colin Cotter
Rob Kirby	Koki Sagiyama	Nacime Bouziani	Sophia Vorderwuel-becke
Tom Gregory	Jack Betteridge	Daniel Shapero	Reuben Nixon-Hill
Connor Ward			

0.2.2 Former team members

 Thomas Gibson	Miklós Homolya	Tianjiao (TJ) Sun	Andrew McRae
 Fabio Luporini	Alastair Gregory	Michael Lange	Simon Funke
Florian Rathgeber	 Doru Bercea	 Graham Markall	

0.2.3 Other contributors

Julian Andrej	Melina Giagiozis	Eike Mueller
Nicolas Barral	Tim Greaves	Alberto Paganini
Nicholas Barton	Christopher Hawkes	Francis Poulin
Tom Bendall	Christian Jacobs	Asbjørn Nilsen Riseth
George Boutsioukis	Darko Janeković	Hannah Rittich
Romain Brault	Nick Johnson	Francis P. Russell
Pablo Brubeck	Anna Kalogirou	Thomas Roy
Ed Bueler	Tuomas Kärnä	Tomasz Salwa
Henrik Buesing	Stephan Kramer	Kaho Sato
Justin Chang	Nicolas Loriant	Ben Sepanski
Cyrus Cheng	Scott MacLachlan	Jemma Shipton
Teodoro Fields Collin	India Marsden	Joe Wallwork
Joshua Coutinho	Geordie McBain	Florian Wechsung
Patrick E. Farrell	Oliver Meister	Fangyi Zhou

And more! Please contact us if you believe your name has been left off this list in error.

0.3 Obtaining Firedrake

Firedrake is installed using its install script:

```
curl -O https://raw.githubusercontent.com/firedrakeproject/firedrake/master/
↪scripts/firedrake-install
```

In the simplest cases, such as on a Mac with Homebrew installed or on an Ubuntu workstation on which the user has sudo acccess, the user can simply run:

```
python3 firedrake-install
```

Running `firedrake-install` with no arguments will install Firedrake in a python venv created in a `firedrake` subdirectory of the current directory. Run:

```
python3 firedrake-install --help
```

for a full list of install options. In particular, you may wish to customise the set of options used to build PETSc. To do so, set the environment variable `PETSC_CONFIGURE_OPTIONS` before running `firedrake-install`. You can see the set of options passed to PETSc by providing the flag `--show-petsc-configure-options`.

You will need to activate the venv in each shell from which you use Firedrake:

```
source firedrake/bin/activate
```

Note: Should you use `csh`, you will need:

```
source firedrake/bin/activate.csh
```

0.3.1 Installation and MPI

By default, `firedrake-install` will prompt the PETSc installer to download and install its own MPICH library and executables in the virtual environment. This has implications for the performance of the resulting library when run in parallel. Instructions on how best to configure MPI for the installation process are found here.

0.3.2 Testing the installation

We recommend that you run the test suite after installation to check that Firedrake is fully functional. Activate the venv as above and then run:

```
cd $VIRTUAL_ENV/src/firedrake
pytest tests/regression/ -k "poisson_strong or stokes_mini or dg_advection"
```

This command will run a few of the unit tests, which exercise a good chunk of the functionality of the library. These tests should take a minute or less. If they fail to run for any reason, please see the section below on how to diagnose and debug a failed installation. If you want to run the entire test suite you can do `make alltest` instead, but this takes several hours.

Note: There is a known issue which causes parallel tests to hang without failing. This is particularly a problem on MacOS and is due to the version of MPICH installed with Firedrake failing to resolve the local host at ip address `127.0.0.1`. To resolve this issue modify the hosts database at `/etc/hosts` to include the entries:

```
127.0.0.1       LOCALHOSTNAME.local
127.0.0.1       LOCALHOSTNAME
```

where `LOCALHOSTNAME` is the name returned by running the *hostname* command. Should the local host name change, this may require updating.

0.3.3 Upgrade

The install script will install an upgrade script in *firedrake/bin/firedrake-update*. Running this script will update Firedrake and all its dependencies.

Note: You should activate the venv before running *firedrake-update*.

Just like the `firedrake-install` script, running:

```
firedrake-update --help
```

gives a full list of update options. For instance additional Firedrake packages can be installed into an existing Firedrake installation using `firedrake-update`.

0.3.4 System requirements

Firedrake requires Python 3.7.x to 3.11.x. On MacOS Arm (M1 or M2) Python 3.9.x to 3.11.x is required. Many externally managed dependencies such as VTK have yet to create binary wheels for 3.11.x, but we have generated these for the major supported platforms. The installation script is tested on Ubuntu and MacOS X. On Ubuntu 22.04 or later, the system installed Python 3 is supported and tested. On MacOS, the homebrew installed Python 3 is supported and tested:

```
brew install python3
```

Installation is likely to work well on other Linux platforms, although the script may stop to ask you to install some dependency packages. Installation on other Unix platforms may work but is untested. On Linux systems that do not use the Debian package management system, it will be necessary to pass the *–no-package-manager* option to the install script. In this case, it is the user's responsibilty to ensure that they have the system dependencies:

- A C and C++ compiler (for example gcc/g++ or clang), GNU make
- A Fortran compiler (for PETSc)
- Blas and Lapack
- Git, Mercurial

- Python version 3.7.x-3.11.x (3.9.x-3.11.x on MacOS Arm)
- The Python headers
- autoconf, automake, libtool
- CMake
- zlib
- flex, bison

Firedrake has been successfully installed on Windows 10 using the Windows Subsystem for Linux. There are more detailed instructions for WSL on the Firedrake wiki. Installation on previous versions of Windows is unlikely to work.

System anti-requirements

We strive to make Firedrake work on as many platforms as we can. Some tools, however, make this challenging or impossible for end users.

Anaconda. The Anaconda Python distribution and package manager are often recommended in introductory data science courses because it does effectively handle many aggravating problems of dependency management. Unfortunately, Anaconda does a poor job of isolating itself from the rest of your system and assumes that it will be both the only Python installation and the only supplier of any dependent packages. Anaconda will install compilers and MPI compiler wrappers and put its compilers right at the top of your `PATH`. This is a problem because Firedrake needs to build and use its own MPI. (We keep our MPI isolated from the rest of your system through virtual environments.) When installed on a platform with Anaconda, Firedrake can accidentally try to link to the incompatible Anaconda installation of MPI.

There are three ways to work around this problem.

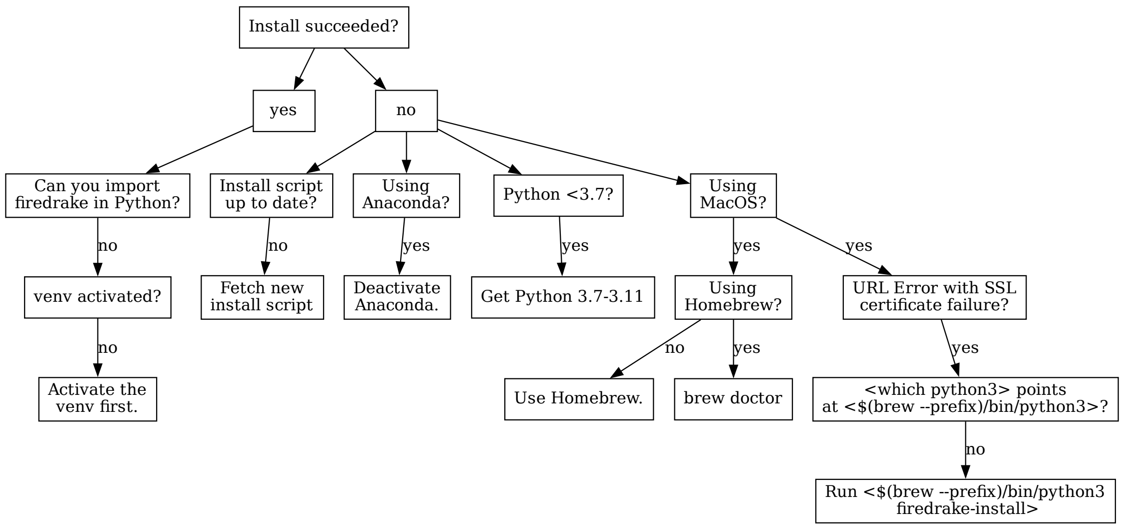
1. Remove Anaconda entirely.
2. Modify your `PATH` environment variable to remove any traces of Anaconda, then install Firedrake. If you need Anaconda later, you can re-enable it with a shell script that will add those directories back onto your path.
3. Use a Docker image that we've built with Firedrake and its dependencies already installed.

MacOS system Python. The official MacOS installer on the Python website does not have a working SSL by default. A working SSL is necessary to securely fetch dependent packages from the internet. You can enable SSL with the system Python, but we strongly recommend using a Python version installed via Homebrew instead.

MacPorts. Mac OS has multiple competing package managers which sometimes cause issues for users attempting to install Firedrake. In particular, the assembler provided by MacPorts is incompatible with the Mac system compilers in a manner which causes Firedrake to fail to install. For this reason, if you are installing Firedrake on a Mac which also has MacPorts installed, you should ensure that `/opt/local/bin` and `/opt/local/sbin` are removed from your `PATH` when installing or using Firedrake. This should ensure that no MacPorts installed tools are found.

0.3.5 Debugging install problems

If `firedrake-install` fails, the following flowchart describes some common build problems and how to solve them. If you understand the prognosis and feel comfortable making these fixes yourself then great! If not, feel free to ask for more help in our *Slack channel*.



If you don't see the issue you're experiencing in this chart, please ask us on Slack or create a post on github discussions. To help us diagnose what's going wrong, **please include the following log files**:

- `firedrake-install.log` from Firedrake, which you can find in the directory where you invoked `firedrake-install` from
- `configure.log` and `make.log` from PETSc, which you can find in `src/petsc/` inside the directory where Firedrake virtual environment was created

Likewise, if it's `firedrake-update` that fails, please include the file `firedrake-update.log`. You can find this in the Firedrake virtual environment.

Recovering from a broken installation script

If you find yourself in the unfortunate position that `firedrake-update` won't run because of a bug, and the bug has been fixed in Firedrake master, then the following procedure will rebuild `firedrake-update` using the latest version.

From the top directory of your Firedrake install, type:

```
cd src/firedrake
git pull
./scripts/firedrake-install --rebuild-script
```

You should now be able to run `firedrake-update`.

0.3.6 Visualisation software

Firedrake can output data in VTK format, suitable for viewing in Paraview. On Ubuntu and similar systems, you can obtain Paraview by installing the `paraview` package. On Mac OS, the easiest approach is to download a binary from the paraview website.

0.3.7 Building the documentation

If you want to be able to view and edit the documentation locally, run:

```
python3 firedrake-install --documentation-dependencies
```

when installing Firedrake, or in an existing instalation (after running `source firedrake/bin/activate` to activate the virtual env) run:

```
firedrake-update --documentation-dependencies
```

The documentation can be found in `firedrake/firedrake/src/firedrake/docs` and can be built by executing:

```
make html
```

This will generate the HTML documentation (this website) on your local machine.

0.3.8 Removing Firedrake

Firedrake and its dependencies can be removed by deleting the Firedrake install directory. This is usually the `firedrake` subdirectory created after having run `firedrake-install`. Note that this will not undo the installation of any system packages which are Firedrake dependencies: removing these might affect subsequently installed packages for which these are also dependencies.

0.4 Citing Firedrake

If you publish results using Firedrake, we would be grateful if you would cite the relevant papers.

The simplest way to determine what these are is by asking Firedrake itself. You can ask that a list of citations relevant to your computation be printed when exiting by calling `Citations.print_at_exit()` after importing Firedrake:

```
from firedrake import *

Citations.print_at_exit()
```

Alternatively, you can select that this should occur by passing the command-line option `-citations`. In both cases, you will also obtain the correct citations for PETSc.

If you cannot use this approach, there are a number of papers. Those which are relevant depend a little on which functionality you used.

For Firedrake itself, please cite [RHM+16]. If you use the *extruded mesh* functionality please cite [MBM+16] and [BMH+16]. When using quadrilateral meshes, please cite [HH16] and [MBM+16].

The form compiler, TSFC, is documented in [HMLH18] and [HKH17]. If, in addition, your work relies on the kernel-level performance optimisations that Firedrake performs using COFFEE, please cite the COFFEE papers [LVR+15] and [LHK17].

If you make use of matrix-free functionality and custom block preconditioning, please cite [KM18].

If you would like to help us to keep track of research directly benefitting from Firedrake, please feel free to add your paper in bibtex format in the bibliography for firedrake applications.

0.4.1 Citing other packages

Firedrake relies heavily on PETSc, which you should cite appropriately. Additionally, if you talk about UFL in your work, please cite the UFL paper.

Making your simulations reproducible with Zenodo integration

In addition to citing the work you use, you will want to provide references to the exact versions of Firedrake and its dependencies which you used. Firedrake supports this through *Zenodo integration*.

0.5 Getting in touch

Should you wish to get in touch with the Firedrake development team, for help, feature requests, bugs, or just to say hello, you can do so in a number of different ways.

0.5.1 GitHub discussions

The go to place for asking for help is GitHub discussions. For support, please ask a question under the Firedrake support category. We're also always very happy to accept third-party contributions to Firedrake.

0.5.2 Slack

Much of the day to day development discussion for Firedrake takes place on our slack channel. This is open to all, but you must request an invite to join the channel. Should this not work for whatever reason, please get in touch via GitHub discussions.

0.5.3 Mailing list

Please join the Firedrake mailing list: firedrake@imperial.ac.uk. This is a very low traffic list but it does carry important announcements, for example when we change a user-facing interface. Join the list on this page.

CHAPTER

ONE

MANUAL

1.1 Defining variational problems

Firedrake uses a high-level language, UFL, to describe variational problems. To do this, we need a number of pieces. We need a representation of the domain we're solving the PDE (partial differential equation) on: Firedrake uses a *Mesh()* for this. On top of this mesh, we build *FunctionSpace*s which define the space in which the solutions to our equation live. Finally we define *Function*s in those function spaces to actually hold the solutions.

1.1.1 Constructing meshes

Firedrake can read meshes in Gmsh, triangle, CGNS, and Exodus formats. To build a mesh one uses the *Mesh()* constructor, passing the name of the file as an argument, which see for more details. The mesh type is determined by the file extension, for example if the provided filename is `coastline.msh` the mesh is assumed to be in Gmsh format, in which case you can construct a mesh object like so:

```
coastline = Mesh("coastline.msh")
```

This works in both serial and parallel, Firedrake takes care of decomposing the mesh among processors transparently.

Reordering meshes for better performance

Most mesh generators produce badly numbered meshes (with bad data locality) which can reduce the performance of assembling and solving finite element problems. By default then, Firedrake reorders input meshes to improve data locality by performing reverse Cuthill-McKee reordering on the adjacency matrix of the input mesh. If you know your mesh has a good numbering (perhaps your mesh generator uses space filling curves to number entities) then you can switch off this reordering by passing `reorder=False` to the appropriate *Mesh()* constructor. You can control Firedrake's default behaviour in reordering meshes with the `"reorder_meshes"` parameter. For example, to turn off mesh reordering globally:

```
from firedrake import *
parameters["reorder_meshes"] = False
```

The parameter passed in to the mesh constructor overrides this default value.

Note: Firedrake numbers degrees of freedom in a function space by visiting each cell in order and performing a depth first numbering of all degrees of freedom on that cell. Hence, if your mesh has a good numbering, the degrees of freedom will too.

Utility mesh functions

As well as offering the ability to read mesh information from a file, Firedrake also provides a number of built in mesh types for a number of standard shapes. 1-dimensional intervals may be constructed with `IntervalMesh()`; 2-dimensional rectangles with `RectangleMesh()`; and 3-dimensional boxes with `BoxMesh()`. There are also more specific constructors (for example to build unit square meshes). See `utility_meshes` for full details.

Immersed manifolds

In addition to the simple meshes described above, Firedrake also has support for solving problems on orientable immersed manifolds. That is, meshes in which the entities are *immersed* in a higher dimensional space. For example, the surface of a sphere in 3D.

If your mesh is such an immersed manifold, you need to tell Firedrake that the geometric dimension of the coordinate field (defining where the points in mesh are) is not the same as the topological dimension of the mesh entities. This is done by passing an optional second argument to the mesh constructor which specifies the geometric dimension. For example, for the surface of a sphere embedded in 3D we use:

```
sphere_mesh = Mesh('sphere_mesh.node', dim=3)
```

Firedrake provides utility meshes for the surfaces of spheres immersed in 3D that are approximated using an icosahedral mesh. You can either build a mesh of the unit sphere with `UnitIcosahedralSphereMesh()`, or a mesh of a sphere with specified radius using `IcosahedralSphereMesh()`. The meshes are constructed by recursively refining a regular icosahedron, you can specify the refinement level by passing a non-zero `refinement_level` to the constructor. For example, to build a sphere mesh that approximates the surface of the Earth (with a radius of 6371 km) that has subdivided the original icosahedron 7 times we would write:

```
earth = IcosahedralSphereMesh(radius=6371, refinement_level=7)
```

Ensuring consistent cell orientations

Variational forms that include particular function spaces (those requiring a *contravariant Piola transform*), require information about the orientation of the cells. For normal meshes, this can be deduced automatically. However, when using immersed meshes, Firedrake needs extra information to calculate the orientation of each cell relative to some global orientation. This is used by Firedrake to ensure that the cell normal on, say, the surface of a sphere, uniformly points outwards. To do this, after constructing an immersed mesh, we must initialise the cell orientation information. This is carried out with the function `init_cell_orientations()`, which takes a UFL expression used to produce the reference normal direction. For example, on the sphere mesh of the earth defined above we can initialise the cell orientations relative to vector pointing out from the origin:

```
earth.init_cell_orientations(SpatialCoordinate(earth))
```

However, a more complicated expression would be needed to initialise the cell orientations on a toroidal mesh.

Semi-structured extruded meshes

Firedrake has special support for solving PDEs on high-aspect ratio domains, such as in the ocean or atmosphere, where the numerics dictate that the “short” dimension should be structured. These are termed *extruded meshes* and have a *separate section* in the manual.

1.1.2 Building function spaces

Now that we have a mesh of our domain, we need to build the function spaces the solution to our PDE will live in, along with the spaces for the trial and test functions. To do so, we use the `FunctionSpace()` constructor. This is the only way to obtain a function space for a scalar variable, such as pressure, which has a single value at each point in the domain.

To construct a function space, you must specify its family and polynomial degree. To build a scalar-valued function space of continuous piecewise-cubic polynomials, we write:

```
V = FunctionSpace(mesh, "Lagrange", 3)
```

There are three main routes to obtaining a function space for a vector-valued variable such as velocity. Firstly, you can pass the `FunctionSpace()` constructor a natively *vector-valued* family such as `"Raviart-Thomas"`. Secondly, you may use the `VectorFunctionSpace()` constructor with a *scalar-valued* family, which gives a vector-valued space where each component is identical to the appropriate scalar-valued `FunctionSpace`. Thirdly, you can create a `VectorElement` directly (which is itself *vector-valued* and pass that to the `FunctionSpace()` constructor).

To build a vector-valued function space using the lowest-order `Raviart-Thomas` elements, we write

```
V = FunctionSpace(mesh, "Raviart-Thomas", 1)
```

To build a vector-valued function space for which each component is a discontinuous piecewise-quadratic polynomial, we can write either

```
V = VectorFunctionSpace(mesh, "Discontinuous Lagrange", 2)
```

or

```
Vele = VectorElement("Discontinuous Lagrange", cell=mesh.ufl_cell(), degree=2)
V = FunctionSpace(mesh, Vele)
```

Advanced usage of `VectorFunctionSpace`

By default, the number of components of a *`VectorFunctionSpace()`* is the geometric dimension of the mesh (e.g. 3, if the mesh is 3D). However, sometimes we might want the number of components in the vector to differ from the geometric dimension of the mesh. We can do this by passing a value for the `dim` argument to the *`VectorFunctionSpace()`* constructor. For example, if we wanted a vector-valued function space on the surface of a unit sphere mesh with only 2 components, we might write:

```
mesh = UnitIcosahedralSphereMesh(refinement_level=3)
V = VectorFunctionSpace(mesh, "Lagrange", 1, dim=2)
```

Mixed function spaces

Many PDEs are posed in terms of multiple, coupled, variables. The variational problem for such a PDE uses a so-called *mixed* function space. In Firedrake, this is represented by a *`MixedFunctionSpace`*. We can either build such a space by invoking the *`constructor directly`*, or, more readably, by taking existing function spaces and multiplying them together using the `*` operator. For example:

```
V = FunctionSpace(mesh, 'RT', 1)
Q = FunctionSpace(mesh, 'DG', 0)
W = V*Q
```

is equivalent to:

```
V = FunctionSpace(mesh, 'RT', 1)
Q = FunctionSpace(mesh, 'DG', 0)
W = MixedFunctionSpace([V, Q])
```

Function spaces on extruded meshes

On *extruded meshes*, we build function spaces by taking a tensor product of the base ("horizontal") space and the extruded ("vertical") space. Firedrake allows us to separately choose the horizontal and vertical spaces when building a function space on an extruded mesh. We refer the reader to the *manual section on extrusion* for details.

1.1.3 Supported finite elements

Firedrake supports the use of the following finite elements.

Name	Short name	Value shape	Valid cells
Bernstein		scalar	interval, triangle, tetrahedron
Brezzi-Douglas-Marini	BDM	vector	triangle, tetrahedron

continues on next page

Table 1 – continued from previous page

Name	Short name	Value shape	Valid cells
Brezzi-Douglas-Fortin-Marini	BDFM	vector	triangle, tetrahedron
Bubble	B	scalar	interval, triangle, tetrahedron
FacetBubble	FB	scalar	interval, triangle, tetrahedron
Crouzeix-Raviart	CR	scalar	triangle, tetrahedron
Discontinuous Lagrange	DG	scalar	interval, triangle, tetrahedron, quadrilateral, hexahedron
Discontinuous Raviart-Thomas	DRT	vector	triangle, tetrahedron
Discontinuous Taylor	TDG	scalar	interval, triangle, tetrahedron
Gauss-Legendre	GL	scalar	interval
Gauss-Lobatto-Legendre	GLL	scalar	interval
HDiv Trace	HDivT	scalar	interval, triangle, tetrahedron, quadrilateral, hexahedron
Hellan-Herrmann-Johnson	HHJ	tensor	triangle
Nonconforming Arnold-Winther	AWnc	tensor	triangle, tetrahedron
Conforming Arnold-Winther	AWc	tensor	triangle, tetrahedron
Hermite	HER	scalar	interval, triangle, tetrahedron
Kong-Mulder-Veldhuizen	KMV	scalar	triangle, tetrahedron
Argyris	ARG	scalar	triangle
Mardal-Tai-Winther	MTW	vector	triangle
Morley	MOR	scalar	triangle
Bell	BELL	scalar	triangle
Lagrange	CG	scalar	interval, triangle, tetrahedron, quadrilateral, hexahedron
Nedelec 1st kind H(curl)	N1curl	vector	triangle, tetrahedron
Nedelec 2nd kind H(curl)	N2curl	vector	triangle, tetrahedron
Raviart-Thomas	RT	vector	triangle, tetrahedron
Regge		tensor	triangle, tetrahedron
DQ		scalar	interval, quadrilateral, hexahedron
Q		scalar	interval, quadrilateral, hexahedron
RTCE		vector	quadrilateral
RTCF		vector	quadrilateral
NCE		vector	hexahedron
NCF		vector	hexahedron
Real	R	scalar	interval, triangle, tetrahedron, quadrilateral, hexahedron
DPC		scalar	interval, quadrilateral, hexahedron
S		scalar	interval, quadrilateral, hexahedron
SminusF		vector	quadrilateral

continues on next page

Table 1 – continued from previous page

Name	Short name	Value shape	Valid cells
SminusDiv		vector	quadrilateral, hexahedron
SminusE		vector	quadrilateral, hexahedron
SminusCurl		vector	quadrilateral, hexahedron
DPC L2		scalar	interval, quadrilateral, hexahedron
Discontinuous Lagrange L2	DG L2	scalar	interval, triangle, tetrahedron, quadrilateral, hexahedron
Gauss-Legendre L2	GL L2	scalar	interval
DQ L2		scalar	interval, quadrilateral, hexahedron
Direct Serendipity	Sdirect	scalar	quadrilateral

In addition, the `TensorProductElement` operator can be used to create product elements on extruded meshes.

Element variants

Some finite element spaces offer more than one choice of nodes. For Q, DQ, DQ L2, RTCE and RTCF spaces on intervals, quadrilaterals and hexahedra, Firedrake offers both equispaced points and better conditioned Legendre points. For discontinuous elements these are the Gauss-Legendre points, and for continuous elements these are the Gauss-Lobatto-Legendre points. These are selected by passing *variant="equispaced"* or *variant="spectral"* to the `FiniteElement` constructor. For example:

```
fe = FiniteElement("RTCE", quadrilateral, 2, variant="equispaced")
```

The default is the spectral variant.

1.1.4 Expressing a variational problem

Firedrake uses the UFL language to express variational problems. For complete documentation, we refer the reader to the UFL package documentation and the description of the language in TOMS. We present a brief overview of the syntax here, for a more didactic introduction, we refer the reader to the Firedrake tutorial examples.

Building test and trial spaces

Now that we have function spaces that our solution will live in, the next step is to actually write down the variational form of the problem we wish to solve. To do this, we will need a test function in an appropriate space along with a function to hold the solution and perhaps a trial function. Test functions are obtained via a call to *`TestFunction`*, trial functions via *`TrialFunction`* and functions with *`Function`*. The former two are purely symbolic objects, the latter contains storage for the coefficients of the basis functions in the function space. We use them as follows:

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

Note: A newly allocated *Function* has coefficients which are all zero.

If V above were a *MixedFunctionSpace*, the test and trial functions we obtain are for the combined mixed space. Often, we would like to have test and trial functions for the subspaces of the mixed space. We can do this by asking for *TrialFunctions* and *TestFunctions*, which return an ordered tuple of test and trial functions for the underlying spaces. For example, if we write:

```
V = FunctionSpace(mesh, 'RT', 1)
Q = FunctionSpace(mesh, 'DG', 0)
W = V * Q

u, p = TrialFunctions(W)
v, q = TestFunctions(W)
```

then u and v will be, respectively, trial and test functions for V, while p and q will be trial and test functions for Q.

Note: If we intend to build a variational problem on a mixed space, we cannot build the individual test and trial functions on the function spaces that were used to construct the mixed space directly. The functions that we build must "know" that they come from a mixed space or else Firedrake will not be able to assemble the correct system of equations.

A first variational form

With our test and trial functions defined, we can write down our first variational form. Let us consider solving the identity equation:

$$u = f \quad \text{on}\, \Omega$$

where Ω is the unit square, using piecewise linear polynomials for our solution. We start with a mesh and build a function space on it:

```
mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "CG", 1)
```

now we need a test function, and since u is unknown, a trial function:

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

finally we need a function to hold the right hand side f which we will populate with the x component of the coordinate field.

```
f = Function(V)
x = SpatialCoordinate(mesh)
f.interpolate(x[0])
```

For details on how *interpolate()* works, see the *appropriate section in the manual*. The variational problem is to find $u \in V$ such that

$$\int_\Omega uv \,\mathrm{d}x = \int_\Omega fv \,\mathrm{d}x \quad \forall v \in V$$

we define the variational problem in UFL with:

```
a = u*v*dx
L = f*v*dx
```

Where the `dx` indicates that the integration should be carried out over the cells of the mesh. UFL can also express integrals over the boundary of the domain, using `ds`, and the interior facets of the domain, using `dS`.

How to solve such variational problems is the subject of the *next section*, but for completeness we show how to do it here. First we define a function to hold the solution

```
s = Function(V)
```

and call *solve()* to solve the variational problem:

```
solve(a == L, s)
```

1.1.5 Forms with constant coefficients

Many PDEs will contain values that are constant over the whole mesh, but may vary in time. For example, a time-varying diffusivity, or a time-dependent forcing function. Although you can create a new form for each new value of this constant, this will not be efficient, since Firedrake must generate new code each time the value changes. A better option is to use a *Constant* coefficient. This object behaves exactly like a *Function*, except that it has a single value over the whole mesh. One may assign a new value to the *Constant* using the *assign()* method. As an example, let us consider a form which contains a time varying constant which we wish to assemble in a time loop. We can use a *Constant* to do this:

```
...
t = 0
dt = 0.1
from math import exp
c = Constant(exp(-t))
# Exponentially decaying RHS
L = f*v*c*dx
while t < tend:
    solve(a == L, ...)
    t += dt
    c.assign(exp(-t))
```

Warning: Although UFL supports computing the derivative of a form with respect to a *Constant*, the resulting form will have an unknown in the reals, which is currently unsupported by Firedrake.

1.1.6 Incorporating boundary conditions

Boundary conditions enter the variational problem in one of two ways. *Natural* (often termed *Neumann* or *weak*) boundary conditions, which prescribe values of the derivative of the solution, are incorporated into the variational form. *Essential* (often termed *Dirichlet* or *strong*) boundary conditions, which prescribe values of the solution, become prescriptions on the function space. In Firedrake, the former are naturally expressed as part of the formulation of the variational problem, the latter are represented as `DirichletBC` objects and are applied when solving the variational problem. Construction of such a strong boundary condition requires a function space (to impose the boundary condition in), a value and a subdomain to apply the boundary condition over:

```
bc = DirichletBC(V, value, subdomain_id)
```

The `subdomain_id` is an integer indicating which section of the mesh the boundary condition should be applied to. The subdomain ids for the various *utility meshes* are described in their respective constructor documentation. For externally generated meshes, Firedrake just uses whichever ids the mesh generator provided. The `value` may be either a scalar, or more generally a UFL expression, for example a `Function` or `Constant`, of the appropriate shape. You may also supply an iterable of literal constants:

```
bc = DirichletBC(V, (1.0, 2.0), 1)
```

Strong boundary conditions are applied in the solve by passing a list of boundary condition objects:

```
solve(a == L, bcs=[bc])
```

See the *next section* for a more complete description of the interface Firedrake provides to solve PDEs. The details of how Firedrake applies strong boundary conditions are slightly involved and therefore have *their own section* in the manual.

Boundary conditions on interior facets

If you wish to apply strong boundary conditions to interior facets of your mesh, this is transparently supported. You should arrange that your mesh generator marks those facets on which you wish to apply boundary conditions, and just use the subdomain ids as usual.

Special subdomain ids

As well as integer subdomain ids that come from marked portions of the mesh, Firedrake also supports the magic string `"on_boundary"` to apply a boundary condition to all exterior facets of the mesh. Further, on :doc`:extruded meshes <extruded-meshes>` the special strings `"top"` and `"bottom"` can be used to apply a boundary condition on respectively the top and bottom of the extruded domain.

Note: These special strings cannot be combined with integer ids, so if you want to apply boundary data on an extruded mesh on (say) ids `1` and `2` as well as the top of the domain you would write

```
bcs = [DirichletBC(V, ..., (1, 2)), DirichletBC(V, ..., "top")]
```

Specifying conditions on components of a space

When solving a problem defined on either a *MixedFunctionSpace* or a rank-1 *FunctionSpace*, it is common to want to specify boundary values for only some of the components. In the former case, this is the only supported method of setting boundary values, the latter also supports setting the value for all components. In both cases, the syntax is the same. When defining the *DirichletBC* we must index the function space used. For example, to specify that the third component of a *VectorFunctionSpace()* should take the boundary value 0, we write:

```
V = VectorFunctionSpace(mesh, ...)
bc = DirichletBC(V.sub(2), Constant(0), boundary_ids)
```

Note that when indexing a *MixedFunctionSpace* in this manner, one pulls out the indexed subspace, rather than a component. For example, to specify the velocity values in a Taylor-Hood discretisation we write:

```
V = VectorFunctionSpace(mesh, "CG", 2)
P = FunctionSpace(mesh, "CG", 1)
W = V*P

bcv = DirichletBC(W.sub(0), Constant((0, 0)), boundary_ids)
```

If we only wanted to specify a single component, we would have to index twice. For example, specifying that the x-component of the velocity is zero, using the same function space definitions:

```
bcv_x = DirichletBC(W.sub(0).sub(0), Constant(0), boundary_ids)
```

Boundary conditions in discontinuous spaces

Firedrake uses the topological association of nodes to facets to determine where to apply strong boundary conditions. For spaces where nodes are not topologically associated with the boundary facets, such as discontinuous Galerkin spaces, you should instead apply boundary conditions weakly.

Time dependent boundary conditions

Imposition of time-dependent boundary conditions can by carried out by modifying the value in the appropriate *DirichletBC* object. Note that if you use a literal value to initialise the boundary condition object within the timestepping loop, this will necessitate a recompilation of code every time the boundary condition changes. For this reason we either recommend using a *Constant* if the boundary condition is spatially uniform, or a UFL expression if it has both space and time-dependence. For example, a purely time-varying boundary condition might be implemented as:

```
c = Constant(sin(t))
bc = DirichletBC(V, c, 1)
while t < T:
    solve(F == 0, bcs=[bc])
    t += dt
    c.assign(sin(t))
```

If the boundary condition instead has both space and time dependence we can write:

```
c = Constant(t)
e = sin(x[0]*c)
bc = DirichletBC(V, e, 1)
while t < T:
    solve(F == 0, bcs=[bc])
    t += dt
    c.assign(t)
```

1.1.7 More complicated forms

UFL is a fully-fledged language for expressing variational problems, and hence has operators for all appropriate vector calculus operations along with special support for discontinuous galerkin methods in the form of symbolic expressions for facet averages and jumps. For an introduction to these concepts we refer the user to the UFL manual as well as the Firedrake tutorials which cover a wider variety of different problems.

1.2 Solving PDEs

1.2.1 Introduction

Now that we have learnt how to define weak variational problems, we will move on to how to actually solve them using Firedrake. Let us consider a weak variational problem

$$\begin{aligned} a(u,v) = L(v)\ \forall v \in V \text{on}\ \Omega \\ u = u_0 \text{ on } \partial\Omega \end{aligned}$$

we will call the bilinear and linear parts of this form `a` and `L` respectively. The strongly imposed boundary condition, $u = u_0 \text{ on } \partial\Omega$ will be represented by a variable of type *DirichletBC*, `bc`.

Now that we have all the pieces of our variational problem, we can move forward to solving it.

1.2.2 Solving the variational problem

The function used to solve PDEs defined as above is *solve()*. This is a unified interface for solving both linear and non-linear variational problems along with linear systems (where the arguments are already assembled matrices and vectors, rather than UFL forms). We will treat the variational interface first.

Linear variational problems

If the problem is linear, that is `a` is linear in both the test and trial functions and `L` is linear in the test function, we can use the linear variational problem interface to `solve`. To start, we need a *Function* to hold the value of the solution:

```
s = Function(V)
```

We can then solve the problem, placing the solution in `s` with:

```
solve(a == L, s)
```

To apply boundary conditions, one passes a list of *DirichletBC* objects using the `bcs` keyword argument. For example, if there are two boundary conditions, in `bc1` and `bc2`, we write:

```
solve(a == L, s, bcs=[bc1, bc2])
```

Nonlinear variational problems

For nonlinear problems, the interface is similar. In this case, we solve a problem:

$$\begin{aligned} F(u; v) &= 0 \; \forall v \in V \text{on}\, \Omega \\ u &= u_0 \text{ on } \partial\Omega \end{aligned}$$

where the *residual* $F(u; v)$ is linear in the test function v but possibly non-linear in the unknown *Function* u. To solve such a problem we write, if `F` is the residual form:

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

to apply strong boundary conditions, as before, we provide a list of `DirichletBC` objects using the `bcs` keyword:

```
solve(F == 0, u, bcs=[bc1, bc2])
```

Nonlinear problems in Firedrake are solved using Newton-like methods. That is, we compute successive approximations to the solution using

$$u_{k+1} = u_k - J(u_k)^{-1}F(u_k) \; k = 0, 1, \dots$$

where u_0 is an initial guess for the solution and $J(u_k) = \frac{\partial F(u_k)}{\partial u_k}$ is the *Jacobian* of the residual, which should be non-singular at each iteration. Notice how in the above examples, we did not explicitly supply a Jacobian. If it is not supplied, it will be computed by automatic differentiation of the residual form `F` with respect to the solution variable `u`. However, we may also supply the Jacobian explicitly, using the keyword argument `J`:

```
solve(F == 0, u, J=user_supplied_jacobian_form)
```

The initial guess for the Newton iterations is provided in `u`, for example, to provide a non-zero guess that the solution is the value of the `x` coordinate everywhere:

```
x = SpatialCoordinate(m)
u.interpolate(x[0])

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

1.2.3 Solving linear systems

Often, we might be solving a time-dependent linear system. In this case, the bilinear form `a` does not change between timesteps, whereas the linear form `L` does. Since assembly of the bilinear form is a potentially costly process, Firedrake offers the ability to "pre-assemble" forms in such systems and then reuse the assembled operator in successive linear solves. Again, we use the same `solve` interface to do this, but must build slightly different objects to pass in. In the pre-assembled case, we are solving a linear system:

$$A\vec{x} = \vec{b}$$

Where A is a known matrix, $\vec{b}$ is a known right hand side vector and $\vec{x}$ is the unknown solution vector. In Firedrake, A is represented as a *Matrix*, while $\vec{b}$ and $\vec{x}$ are both *Function*s. We build these values by calling `assemble` on the UFL forms that define our problem, which, as before are denoted `a` and `L`. Similarly to the linear variational case, we first need a function in which to place our solution:

```
x = Function(V)
```

We then *assemble()* the left hand side matrix `A` and known right hand side `b` from the bilinear and linear forms respectively:

```
A = assemble(a)
b = assemble(L)
```

Finally, we can solve the problem placing the solution in `x`:

```
solve(A, x, b)
```

to apply boundary conditions to the problem, we can assemble the linear operator `A` with boundary conditions using the `bcs` keyword argument to *assemble()* (and then not supply them in solve call):

```
A = assemble(a, bcs=[bc1, bc2])
b = assemble(L)
solve(A, x, b)
```

Warning: It is no longer possible to apply or change boundary conditions after assembling the matrix `A`; pass any necessary boundary conditions to *assemble()*.

1.2.4 Specifying solution methods

Not all linear and non-linear systems defined by PDEs are created equal, and we therefore need ways of specifying which solvers to use and options to pass to them. Firedrake uses PETSc to solve both linear and non-linear systems and presents a uniform interface in `solve` to set PETSc solver options. In all cases, we set options in the solve call by passing a dictionary to the `solver_parameters` keyword argument. To set options we use the same names that PETSc uses in its command-line option setting interface (having removed the leading `-`). For more complete details on PETSc option naming we recommend looking in the PETSc manual. We describe some of the more common options here.

Configuring solvers from the commandline

As well as specifying solver options in a parameters dict at the call site for each solve, one can configure solvers by passing options in the normal PETSc style via the commandline. To do this, we need to specify the `options_prefix` for each solver that we wish to configure via the commandline. This is done by providing a non-`None` argument as the `options_prefix` keyword argument to the solver. The separator between the prefix and the subsequent options is an underscore, which is automatically appended if the provided options prefix does end in one.

When using an options prefix, we do not need to specify the prefix in the solver parameters dictionary (it is automatically added in the appropriate way). Also to note is that command line options *override* parameters set in the dictionary. This way we can provide good defaults for solvers, and override them on a case-by-case basis.

For example, suppose we have a file `pde.py` that contains

```
...
solve(F == 0, u, options_prefix="pde",
      solver_parameters={"ksp_type": "gmres"})
```

If we run this code as:

```
python pde.py
```

Then the KSP solver will be GMRES. Conversely, when running

```
python pde.py -pde_ksp_type cg
```

we will use conjugate gradients as the KSP solver.

Linear solver options

We use a PETSc KSP object to solve linear systems. This is a uniform interface for solving linear systems using Krylov subspace methods. By default, the solve call will use GMRES using an incomplete LU factorisation to precondition the problem. To change the Krylov method used in solving the problem, we set the `'ksp_type'` option. For example, if we want to solve a modified Helmholtz equation, we know the operator is symmetric positive definite, and therefore can choose the conjugate gradient method, rather than GMRES.

```
solve(a == L, solver_parameters={'ksp_type': 'cg'})
```

To change the preconditioner used, we set the `'pc_type'` option. For example, if PETSc has been installed with the Hypre package, we can use its algebraic multigrid preconditioner, BoomerAMG, to precondition the system with:

```
solve(a == L,
      solver_parameters={'pc_type': 'hypre',
                         'pc_hypre_type': 'boomeramg'})
```

Although the *KSP* name suggests that only Krylov methods are supported, this is not the case. We may, for example, solve the system directly by computing an LU factorisation of the problem. To do this, we set the `pc_type` to `'lu'` and tell PETSc to use a "preconditioner only" Krylov method:

```
solve(a == L,
      solver_parameters={'ksp_type': 'preonly',
                         'pc_type': 'lu'})
```

In a similar manner, we can use Jacobi preconditioned Richardson iterations with:

```
solve(a == L,
      solver_parameters={'ksp_type': 'richardson',
                         'pc_type': 'jacobi'}
```

Note: We note in passing that the method Firedrake utilises internally for applying strong boundary conditions does not destroy the symmetry of the linear operator. If the system without boundary conditions is symmetric, it will continue to be so after the application of any boundary conditions.

Setting solver tolerances

In an iterative solver, such as Krylov method, we iterate until some specified tolerance is reached. The measure of how much the current solution $\vec{x}_i$ differs from the true solution is called the residual and is calculated as:

$$r = |\vec{b} - A\vec{x}_i|$$

PETSc allows us to set three different tolerance options for solving the system. The *absolute tolerance* tells us we should stop if r drops below some given value. The *relative tolerance* tells us we should stop if $\frac{r}{|\vec{b}|}$ drops below some given value. Finally, PETSc can detect divergence in a linear solve, that is, if r increases above some specified value. These values are set with the options `'ksp_atol'` for the absolute tolerance, `'ksp_rtol'` for the relative tolerance, and `'ksp_divtol'` for the divergence tolerance. The values provided to these options should be floats. For example, to set the absolute tolerance to 10^{-30}, the relative tolerance to 10^{-9} and the divergence tolerance to 10^{4} we would use:

```
solver_parameters={'ksp_atol': 1e-30,
                   'ksp_rtol': 1e-9,
                   'ksp_divtol': 1e4}
```

Note: By default, PETSc (and hence Firedrake) check for the convergence in the preconditioned norm, that is, if the system is preconditioned with a matrix P the residual is calculated as:

$$r = |P^{-1}(\vec{b} - A\vec{x}_i)|$$

to check for convergence in the unpreconditioned norm set the `'ksp_norm_type'` option to `'unpreconditioned'`.

Finally, we can set the maximum allowed number of iterations for the Krylov method by using the `'ksp_max_it'` option.

Preconditioning mixed finite element systems

PETSc provides an interface to composing "physics-based" preconditioners for mixed systems which Firedrake exploits when it assembles linear systems. In particular, for systems with two variables (for example Navier-Stokes where we solve for the velocity and pressure of the fluid), we can exploit PETSc's ability to build preconditioners from Schur complements. This is one type of preconditioner based on PETSc's fieldsplit technology. To take a concrete example, let us consider solving the dual form of the modified Helmholtz equation:

$$\langle p, q \rangle - \langle q, \mathrm{div} u \rangle + \lambda \langle v, u \rangle + \langle \mathrm{div} v, p \rangle = \langle f, q \rangle \; \forall v \in V_1, q \in V_2$$

This has a stable solution if, for example, V_1 is the lowest order Raviart-Thomas space and V_2 is the lowest order discontinuous space.

```
V1 = FunctionSpace(mesh, 'RT', 1)
V2 = FunctionSpace(mesh, 'DG', 0)
W = V1 * V2
lmbda = 1
u, p = TrialFunctions(W)
v, q = TestFunctions(W)
f = Function(V2)

a = (p*q - q*div(u) + lmbda*inner(v, u) + div(v)*p)*dx
L = f*q*dx

u = Function(W)
solve(a == L, u,
      solver_parameters={'ksp_type': 'cg',
                         'pc_type': 'fieldsplit',
                         'pc_fieldsplit_type': 'schur',
                         'pc_fieldsplit_schur_fact_type': 'FULL',
                         'fieldsplit_0_ksp_type': 'cg',
                         'fieldsplit_1_ksp_type': 'cg'})
```

We refer to section 4.5 of the PETSc manual for more complete details, but briefly describe the options in use here. The monolithic system is conceptually a 2×2 block matrix:

$$\begin{pmatrix} \lambda\langle v, u\rangle & -\langle q, \mathrm{div} u\rangle \\ \langle \mathrm{div} v, p\rangle & \langle p, q\rangle \end{pmatrix} = \begin{pmatrix} A & B \\ C & D \end{pmatrix}.$$

We can factor this block matrix in the following way:

$$\begin{pmatrix} I & 0 \\ CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} I & A^{-1}B \\ 0 & I \end{pmatrix}.$$

This is the *Schur complement factorisation* of the block system, its inverse is:

$$P = \begin{pmatrix} I & -A^{-1}B \\ 0 & I \end{pmatrix} \begin{pmatrix} A^{-1} & 0 \\ 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -CA^{-1} & I \end{pmatrix}.$$

Where S is the *Schur complement*:

$$S = D - CA^{-1}B.$$

The options in the example above use an approximation to P to precondition the system. To do so, we tell PETSc that the preconditioner should be of type `'fieldsplit'`, and the the fieldsplit's type should be `'schur'`. We then select a factorisation type for the Schur complement. The option `'FULL'` as used above preconditions using an approximation to P. We can also use `'diag'` which uses an approximation to:

$$\begin{pmatrix} A^{-1} & 0 \\ 0 & -S^{-1} \end{pmatrix}.$$

Note the minus sign in front of S^{-1} which is there such that this preconditioner is positive definite. Two other options are `'lower'`, where the preconditioner is an approximation to:

$$\begin{pmatrix} A & 0 \\ C & S \end{pmatrix}^{-1} = \begin{pmatrix} A^{-1} & 0 \\ 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -CA^{-1} & I \end{pmatrix}$$

and `'upper'` which uses:

$$\begin{pmatrix} A & B \\ 0 & S \end{pmatrix}^{-1} = \begin{pmatrix} I & -A^{-1}B \\ 0 & I \end{pmatrix} \begin{pmatrix} A^{-1} & 0 \\ 0 & S^{-1} \end{pmatrix}.$$

Note that the inverses of A and S are never formed explicitly by PETSc, instead their actions are computed approximately using a Krylov method. The choice of method is selected using the `'fieldsplit_0_ksp_type'` option (for the Krylov solver computing A^{-1}) and `'fieldsplit_1_ksp_type'` (for the Krylov solver computing S^{-1}).

Note: If you have given your *`FunctionSpace`*s names, then instead of 0 and 1, you should use the name of the function space in these options.

By default PETSc uses an approximation to D^{-1} to precondition the Krylov system solving for S, you can also use a least squares commutator, see the relevant section of the PETSc manual pages for more details.

Specifying assembled matrix types

Firedrake supports the assembly of linear operators in a number of different formats. Either as assembled sparse matrices, or as matrix-free operators that only provide matrix-vector products. Since matrix-free actions require special preconditioning, they have *their own section in the manual*. Even in the sparse matrix case there are a few options. Firedrake can build nested block matrices, or monolithic sparse matrices. The latter admit a wider range of preconditioners, but are memory-inefficient when using fieldsplit preconditioning. Even monolithic matrices have choices, specifically, if there is a block structure, whether that should be exploited or not. Again the trade-off is between memory efficiency (in the block case) and access to a slightly smaller range of preconditioners.

The default matrix type can be set with the global parameter `parameters["default_matrix_type"]`. In the case where the matrix is assembled as a nested matrix, there is a choice as to the type of the blocks (they may be "aij" or "baij"). The default choice can be controlled with `parameters["default_sub_matrix_type"]`. For finer-grained control over the matrix type, one can provide it when calling *`assemble()`* through the `mat_type` and `sub_mat_type` keyword arguments. When using variational solvers, the matrix type is controlled through use of the `solver_parameters` dictionary by specifying the `"mat_type"` entry.

Note: It is not currently possible to control the matrix type of sub-matrices through the solving interface. If you need this functionality, please *get in touch*

More block preconditioners

As well as physics-based Schur complement preconditioners for block systems, PETSc also allows us to use preconditioners formed from block Jacobi (`'pc_fieldsplit_type': 'additive'`) and block Gauss-Seidel (`'multiplicative'` or `'symmetric_multiplicative'`) inverses of the block system. These work for any number of blocks, whereas the Schur complement approach mentioned above only works for two by two blocks. There is also a *separate manual section* on specifying preconditioners that require auxiliary operators.

Recursive fieldsplits

If your system contains more than two fields, it is possible to recursively define block preconditioners by specifying the fields which should belong to each split. Note that at present this only works for "monolithically assembled" matrices, so you should set the solver parameter `"mat_type"` to `"aij"` when solving your system or assembling your matrix. To change the default assembly from nested matrices to monolithically assembled matrices, set the global parameter `parameters["default_matrix_type"] = "aij"`.

As an example, consider a three field system which we wish to precondition by forming a schur complement of the first two fields into the third, and then using a multiplicative fieldsplit with LU on each split for the approximation to A^{-1} and ILU to precondition the schur complement. The solver parameters we need are as follows:

```
parameters = {"pc_type": "fieldsplit",
              "pc_fieldsplit_type": "schur",
              # first split contains first two fields, second
              # contains the third
              "pc_fieldsplit_0_fields": "0, 1",
              "pc_fieldsplit_1_fields": "2",
              # Multiplicative fieldsplit for first field
              "fieldsplit_0_pc_type": "fieldsplit",
              "fieldsplit_0_pc_fieldsplit_type": "multiplicative",
              # LU on each field
              "fieldsplit_0_fieldsplit_0_pc_type": "lu",
              "fieldsplit_0_fieldsplit_1_pc_type": "lu",
              # ILU on the schur complement block
              "fieldsplit_1_pc_type": "ilu"}
```

In this example, none of the `FunctionSpace`s used had names, and hence we referred to the fields by number. If the function spaces are named, then any time a single field appears as a split, its options prefix is referred to by the space's *name* (rather than a number). Concretely, if the previous example had use a set of FunctionSpace definitions:

```
V = FunctionSpace(..., name="V")
P = FunctionSpace(..., name="P")
T = FunctionSpace(..., name="T")
W = V*P*T
```

Then we would have referred to the single (field 1) split using `fieldsplit_T_pc_type`, rather than `fieldsplit_1_pc_type`.

Specifying nested options blocks

For complex nested preconditioners, it can be tedious to write out the same prefix over and over. Moreover, we may have a block system where multiple blocks use the same preconditioning options. It is then error-prone to type these options out twice. To alleviate these problems, one can describe the nesting in the solver parameters dictionary by using a nested `dict` as the value. In this case, the key is used as an options prefix to all of the key-value pairs in the nested dictionary. As an example, the following two parameter sets are equivalent:

```
{"ksp_type": "cg",
 "pc_type": "fieldsplit",
 "fieldsplit_0": {"ksp_type": "gmres",
                  "pc_type": "hypre",
                   "ksp_rtol": 1e-5},
 "fieldsplit_1": {"ksp_type": "richardson",
                  "pc_type": "ilu"}}
```

and

```
{"ksp_type": "cg",
 "pc_type": "fieldsplit",
```

(continues on next page)

(continued from previous page)

```
"fieldsplit_0_ksp_type": "gmres",
"fieldsplit_0_pc_type": "hypre",
"fieldsplit_0_ksp_rtol": 1e-5,
"fieldsplit_1_ksp_type": "richardson",
"fieldsplit_1_pc_type": "ilu"}
```

PETSc uses an underscore as a separator between option names, and we do the same. For convenience, the prefix key to a nested dict can omit the trailing underscore, it will be added automatically if missing. Hence

```
{"a": {"b": "foo"}}
```

and

```
{"a_": {"b": "foo"}}
```

both expand to

```
{"a_b": "foo"}
```

Nonlinear solver options

As for linear systems, we use a PETSc object to solve nonlinear systems. This time it is a SNES. This offers a uniform interface to Newton-like and quasi-Newton solution schemes. To select the SNES type to use, we use the `'snes_type'` option. Recall that each Newton iteration is the solution of a linear system, options for the inner linear solve may be set in the same way as described above for linear problems. For example, to solve a nonlinear problem using Newton-Krylov iterations using a line search and direct factorisation to solve the linear system we would write:

```
solve(F == 0, u,
      solver_parameters={'snes_type': 'newtonls',
                         'ksp_type': 'preonly',
                         'pc_type': 'lu'}
```

Note: Not all of PETSc's SNES types are currently supported by Firedrake, since some of them require extra information which we do not currently provide.

Setting convergence criteria

In addition to setting the tolerances for the inner, linear solve in a nonlinear system, which is done in exactly the same way as for *linear problems*, we can also set convergence tolerances on the outer SNES object. These are the *absolute tolerance* (`'snes_atol'`), *relative tolerance* (`'snes_rtol'`), *step tolerance* (`'snes_stol'`) along with the maximum number of nonlinear iterations (`'snes_max_it'`) and the maximum number of allowed function evaluations (`'snes_max_func'`). The step tolerance checks for convergence due to:

$$|\Delta x_k| < \mathrm{stol}\, |x_k|$$

The maximum number of allowed function evaluations limits the number of times the residual may be evaluated before returning a non-convergence error, and defaults to 1000.

Providing an operator for preconditioning

By default, Firedrake uses the Jacobian of the residual (or equally the bilinear form for linear problems) to construct preconditioners for the linear systems it solves. That is, it does not directly solve:

$$A\vec{x} = \vec{b}$$

but rather

$$\tilde{A}^{-1}A\vec{x} = \tilde{A}^{-1}\vec{b}$$

where $\tilde{A}^{-1}$ is an approximation to A^{-1}. If we know something about the structure of our problem, we may be able to construct an operator P explicitly which is "easy" to invert, and whose inverse approximates A^{-1} well. Firedrake allows you to provide this operator when solving variational problems by passing an explicit `Jp` keyword argument to the solve call, the provided form will then be used to construct an approximate inverse when preconditioning the problem, rather than the form we're solving with.

```
a = ...
L = ...
Jp = ...
# Use the approximate inverse of Jp to precondition solves
solve(a == L, ..., Jp=Jp)
```

Default solver options

If no parameters are passed to a solve call, we use, in most cases, the defaults that PETSc supplies for solving the linear or nonlinear system. We describe the most commonly modified options (along with their defaults in Firedrake) here. For linear variational solves we use:

- `ksp_type`: GMRES, with a restart (`ksp_gmres_restart`) of 30
- `ksp_rtol`: 1e-7
- `ksp_atol`: 1e-50
- `ksp_divtol` 1e4

- `ksp_max_it`: 10000
- `pc_type`: ILU (Jacobi preconditioning for mixed problems)

For nonlinear variational solves we have:

- `snes_type`: Newton linesearch
- `ksp_type`: GMRES, with a restart (`ksp_gmres_restart`) of 30
- `snes_rtol`: 1e-8
- `snes_atol`: 1e-50
- `snes_stol`: 1e-8
- `snes_max_it`: 50
- `ksp_rtol`: 1e-5
- `ksp_atol`: 1e-50
- `ksp_divtol`: 1e4
- `ksp_max_it`: 10000
- `pc_type`: ILU (Jacobi preconditioning for mixed problems)

To see the full view that PETSc has of solver objects, you can pass a view flag to the solve call. For linear solves pass:

```
solver_parameters={'ksp_view': None}
```

For nonlinear solves use:

```
solver_parameters={'snes_view': None}
```

PETSc will then print its view of the solver objects that Firedrake has constructed. This is especially useful for debugging complicated preconditioner setups for mixed problems.

1.2.5 Solving singular systems

Some systems of PDEs, for example the Poisson equation with pure Neumann boundary conditions, have an operator which is singular. That is, we have $Ae = 0$ with $e \neq 0$. The vector space spanned by the set of vectors e for which $Ae = 0$ is termed the *null space* of A. If we wish to solve such a system, we must remove the null space from the solution. To do this in Firedrake, we first must define the null space, and then inform the solver of its existance. We use a *`VectorSpaceBasis`* to hold the vectors which span the null space. We must provide a list of *`Function`*s or *`Vector`*s spanning the space. Additionally, since removing a constant null space is such a common operation, we can pass `constant=True` to the constructor (rather than constructing the constant vector by hand). Note that the vectors we pass in must be *orthonormal*. Once the null space is built, we just need to inform the solve about it (using the `nullspace` keyword argument).

As an example, consider the Poisson equation with pure Neumann boundary conditions:

$$\begin{aligned} -\nabla^2 u &= 0 \quad \text{in } \Omega \\ \nabla u \cdot n &= g \quad \text{on } \Gamma. \end{aligned}$$

We will solve this problem on the unit square applying homogeneous Neumann boundary conditions on the planes $x = 0$ and $x = 1$. On $y = 0$ we set $g = -1$ while on $y = 1$ we set $g = 1$. The null space of the operator we form is the set of constant functions, and thus the problem has solution $u(x, y) = y + c$ where c is a constant. To solve the problem, we will inform the solver of this constant null space, fixing the solution to be $u(x, y) = y - 0.5$.

```
m = UnitSquareMesh(25, 25)
V = FunctionSpace(m, 'CG', 1)
u = TrialFunction(V)
v = TestFunction(V)

a = inner(grad(u), grad(v))*dx
L = -v*ds(3) + v*ds(4)

nullspace = VectorSpaceBasis(constant=True)
u = Function(V)
solve(a == L, u, nullspace=nullspace)
x = SpatialCoordinate(m)
exact = Function(V).interpolate(x[1] - 0.5)
print sqrt(assemble((u - exact)*(u - exact)*dx))
```

For this to work, the provided right hand side must be orthogonal to the transpose nullspace of the operator as well. In many cases, we can arrange for this to occur by careful choice of initial conditions. Sometimes this is not possible. In this case, you can ask Firedrake to remove the component of the right hand side that is in the transpose nullspace by providing a *VectorSpaceBasis* with the `transpose_nullspace` keyword argument to *solve()*.

Singular operators in mixed spaces

If you have an operator in a mixed space, you may well precondition the system using a Schur complement. If the operator is singular, you will therefore have to tell the solver about the null space of each diagonal block separately. To do this in Firedrake, we build a *MixedVectorSpaceBasis* instead of a *VectorSpaceBasis* and then inform the solver about it as before. A *MixedVectorSpaceBasis* takes a list of *VectorSpaceBasis* objects defining the null spaces of each of the diagonal blocks in the mixed operator. In addition, as a first argument, you must provide the *MixedFunctionSpace* you're building a basis for. You do not have to provide a null space for all blocks. For those you don't care about, you can pass an indexed function space at the appropriate position. For example, imagine we have a mixed space $W = V \times Q$ and an operator which has a null space of constant functions in V (this occurs, for example, for a discretisation of the mixed poisson problem on the surface of a sphere). We can specify the null space (indicating that we only really care about the constant function) as:

```
V = ...
Q = ...
W = V*Q
v_basis = VectorSpaceBasis(constant=True)
nullspace = MixedVectorSpaceBasis(W, [v_basis, W.sub(1)])
```

1.2.6 Debugging convergence failures

Occasionally, we will set up a problem and call solve only to be confronted with an error that the solve failed to converge. Here, we discuss some useful techniques to try and understand the reason. Much of the advice in the PETSc FAQ is useful here, especially the sections on SNES nonconvergence and KSP nonconvergence. We first consider linear problems.

Linear convergence failures

If the linear operator is correct, but the solve fails to converge, it is likely the case that the problem is badly conditioned (leading to slow convergence) or a symmetric method is being used (such as conjugate gradient) where the problem is non-symmetric. The first thing to check is what happened to the residual (error) term. To monitor this in the solution we pass the "flag" options `'ksp_converged_reason'` and `'ksp_monitor_true_residual'`, additionally, we pass `ksp_view` so that PETSc prints its idea of what the solver object contains (this is useful to debug the where options are not being passed in correctly):

```
solver_parameters={'ksp_converged_reason': None,
                   'ksp_monitor_true_residual': None,
                   'ksp_view': None}
```

If the problem is converging, but only slowly, it may be that it is badly conditioned. If the problem is small, we can try using a direct solve to see if the solution obtained is correct:

```
solver_parameters={'ksp_type': 'preonly', 'pc_type': 'lu'}
```

If this approach fails with a "zero-pivot" error, it is likely that the equations are singular, or nearly so, check to see if boundary conditions have been imposed correctly.

If the problem converges with a direct method to the correct solution but does not converge with a Krylov method, it's probable that the conditioning is bad. If it's a mixed problem, try using a physics-based preconditioner as described above, if not maybe try using an algebraic multigrid preconditioner. If PETSc was installed with Hypre use:

```
solver_parameters={'pc_type': 'hypre', 'pc_hypre_type': 'boomeramg'}
```

If you're using a symmetric method, such as conjugate gradient, check that the linear operator is actually symmetric, which you can compute with the following:

```
A = assemble(a)  # use bcs keyword if there are boundary conditions
print A.M.handle.isSymmetric(tol=1e-13)
```

If the problem is not symmetric, try using a method such as GMRES instead. PETSc uses restarted GMRES with a default restart of 30, for difficult problems this might be too low, in which case, you can increase the restart length with:

```
solver_parameters={'ksp_gmres_restart': 100}
```

Nonlinear convergence failures

Much of the advice for linear systems applies to nonlinear systems as well. If you have a convergence failure for a nonlinear problem, the first thing to do is run with monitors to see what is going on, and view the SNES object with `snes_view` to ensure that PETSc is seeing the correct options:

```
solver_parameters={'snes_monitor': None,
                   'snes_view': None,
                   'ksp_monitor_true_residual': None,
                   'snes_converged_reason': None,
                   'ksp_converged_reason': None}
```

If the linear solve fails to converge, debug the problem as above for linear systems. If the linear solve converges but the outer Newton iterations do not, the problem is likely a bad Jacobian. If you provided the Jacobian by hand, is it correct? If no Jacobian was provided in the solve call, it is likely a bug in Firedrake and you should report it to us.

Checking the provided Jacobian

It is possible to verify that the provided Jacobian is consistent with the residual we are trying to minimise by comparing it with a finite differenced Jacobian computed by PETSc. This is possible using only a few extra options to the call to *solve()*. We just need to specify that the nonlinear solver we want PETSc to employ should be of type `test`. PETSc will then go away, compute an approximate Jacobian by finite differencing the residual and compare it to our provided exact Jacobian. The only thing we need to be aware of is that if the problem to be solved is in a mixed space, we need to set the solver parameter "`mat_type`" to "`aij`" in the solve call.

To make things concrete, consider the following, somewhat contrived, example where we attempt to solve a Galerkin projection in a mixed space, but provide an incorrectly scaled Jacobian to the solve.

```
from firedrake import *
mesh = UnitSquareMesh(1, 1)
V = FunctionSpace(mesh, "CG", 1)
W = V*V
f = Function(W)
v = TestFunction(W)
u = TrialFunction(W)

F = dot(f, v)*dx - dot(Constant((1, 2)), v)*dx

J = Constant(4)*dot(u, v)*dx

solve(F == 0, f, J=J)
```

When run, this produces the following output:

```
pyop2:INFO Solving nonlinear variational problem...
Traceback (most recent call last):
```

(continues on next page)

(continued from previous page)

```
    solve(F == 0, u, J=J)
  File "firedrake/solving.py", line 120, in solve
    _solve_varproblem(*args, **kwargs)
  File "firedrake/solving.py", line 162, in _solve_varproblem
    solver.solve()
  File "<string>", line 2, in solve
  File "pyop2/profiling.py", line 203, in wrapper
    return f(*args, **kwargs)
  File "firedrake/variational_solver.py", line 175, in solve
    solving_utils.check_snes_convergence(self.snes)
  File "firedrake/solving_utils.py", line 62, in check_snes_convergence
    """%s""" % (snes.getIterationNumber(), msg))
RuntimeError: Nonlinear solve failed to converge after 50 nonlinear␣
↪iterations.
Reason:
    DIVERGED_MAX_IT
```

In this example we can notice by inspection of the code that the provided Jacobian is incorrect. The Gateaux derivative of F with respect to f is $\langle u, v\rangle$, not $4\langle u, v\rangle$. In the more general case, it may be that there is a bug in the assembly of the Jacobian, even if the symbolic form is correct. To verify the Jacobian we rerun the solve, but pass some additional options:

```
solve(F == 0, f, J=J,
      solver_parameters={'snes_type': 'test',
                         'mat_type': 'aij'})
```

This time we get the following output

```
pyop2:INFO Solving nonlinear variational problem...
Testing hand-coded Jacobian, if the ratio is
O(1.e-8), the hand-coded Jacobian is probably correct.
Run with -snes_test_display to show difference
of hand-coded and finite difference Jacobian.
Norm of matrix ratio 0.75, difference 1.32288 (user-defined state)
Norm of matrix ratio 0.75, difference 1.32288 (constant state -1.0)
Norm of matrix ratio 0.75, difference 1.32288 (constant state 1.0)
Traceback (most recent call last):
    solve(F == 0, u, J=J, solver_parameters={'snes_type': 'test', 'mat_type':
↪'aij'})
  File "firedrake/solving.py", line 120, in solve
    _solve_varproblem(*args, **kwargs)
  File "firedrake/solving.py", line 162, in _solve_varproblem
    solver.solve()
  File "<string>", line 2, in solve
  File "pyop2/profiling.py", line 203, in wrapper
    return f(*args, **kwargs)
  File "firedrake/variational_solver.py", line 173, in solve
    self.snes.solve(None, v)
  File "PETSc/SNES.pyx", line 520, in petsc4py.PETSc.SNES.solve (src/petsc4py.
```

(continues on next page)

(continued from previous page)

```
↪PETSc.c:165224)
petsc4py.PETSc.Error: error code 73
[0] SNESSolve() line 3907 in petsc/src/snes/interface/snes.c
[0] SNESSolve_Test() line 127 in petsc/src/snes/impls/test/snestest.c
[0] Object is in wrong state
[0] SNESTest aborts after Jacobian test: it is NORMAL behavior.
```

The important lines are:

```
Testing hand-coded Jacobian, if the ratio is
O(1.e-8), the hand-coded Jacobian is probably correct.
Run with -snes_test_display to show difference
of hand-coded and finite difference Jacobian.
Norm of matrix ratio 0.75, difference 1.32288 (user-defined state)
Norm of matrix ratio 0.75, difference 1.32288 (constant state -1.0)
Norm of matrix ratio 0.75, difference 1.32288 (constant state 1.0)
```

Here PETSc is printing information about the difference between the finite difference and provided Jacobians. We can see that these differences are large. Therefore, we conclude the the provided "exact" Jacobian is not consistent with the residual, and likely incorrect.

For comparison, here are the same relevant lines when running with the correct Jacobian:

```
solve(F == 0, f, solver_parameters={'snes_type': 'test', 'mat_type': 'aij'})

Testing hand-coded Jacobian, if the ratio is
O(1.e-8), the hand-coded Jacobian is probably correct.
Run with -snes_test_display to show difference
of hand-coded and finite difference Jacobian.
Norm of matrix ratio 4.98807e-08, difference 2.19953e-08 (user-defined state)
Norm of matrix ratio 2.91936e-08, difference 1.28732e-08 (constant state -1.0)
Norm of matrix ratio 1.51242e-08, difference 6.66915e-09 (constant state 1.0)
```

Notice how now the differences are small (within expected error tolerances) so we are happy that the Jacobian is correct.

1.3 Dirichlet boundary conditions

Strong Dirichlet boundary conditions are imposed by providing a list of `DirichletBC` objects. The class documentation provides the syntax, this document explains the mathematical formulation of the boundary conditions in Firedrake, and their implementation.

1.3.1 Mathematical background

To understand how Firedrake applies strong (Dirichlet) boundary conditions, it is necessary to write the variational problem to be solved in residual form: find $u \in V$ such that:

$$F(u; v) = 0 \quad \forall v \in V.$$

This is the natural form of a nonlinear problem. A linear problem is frequently written: find $u \in V$ such that:

$$a(u, v) = L(v) \quad \forall v \in V.$$

However, this form can trivially be rewritten in residual form by defining:

$$F(u; v) = a(u, v) - L(v).$$

In the general case, F will be always linear in v but may be nonlinear in u.

When we impose a strong (Dirichlet, essential) boundary condition on u, we are substituting the constraint:

$$u = g(x) \text{ on } \Gamma_D$$

for the original equation on Γ_D, where Γ_D is some subset of the domain boundary. To impose this constraint, we first split the function space V:

$$V = V_0 \oplus V_\Gamma$$

where V_Γ is the space spanned by those functions in the basis of V which are non-zero on Γ_D, and V_0 is the space spanned by the remaining basis functions (i.e. those basis functions which vanish on Γ_D).

In Firedrake we always have a nodal basis for V, $\phi_V = \{\phi_i\}$, and we will write ϕ^0 and ϕ^Γ for the subsets of that basis which span V_0 and V_Γ respectively.

We can similarly write $v \in V$ as $v_0 + v_\Gamma$ and use the linearity of F in v:

$$F(u; v) = F(u; v_0) + F(u; v_\Gamma)$$

If we impose a Dirichlet condition over Γ_D then we no longer impose the constraint $F(u; v_\Gamma) = 0$ for any $v_\Gamma \in V_\Gamma$. Instead, we need to impose a term which is zero when u satisfies the boundary conditions, and non-zero otherwise. So we define:

$$F_\Gamma(u; \phi_i) = u_i - g_i \quad \phi_i \in \phi^\Gamma$$

where g_i indicates the evaluation of $g(x)$ at the node associated with ϕ_i. Note that the stipulation that $F_\Gamma(u; v)$ must be linear in v is sufficient to extend the definition to any $v \in V_\Gamma$.

This means that the full statement of the problem in residual form becomes: find $u \in V$ such that:

$$\hat{F}(u; v_0 + v_\Gamma) = F(u; v_0) + F_\Gamma(u; v_\Gamma) = 0 \quad \forall v_0 \in V_0, \forall v_\Gamma \in V_\Gamma.$$

1.3.2 Solution strategy

The system of equations will be solved by a gradient-based nonlinear solver, of which a simple and illustrative example is a Newton solver. Firedrake applies this solution strategy to linear equations too, although in that case only one iteration of the nonlinear solver will ever be required or executed.

We write $u = u_i\phi_i$ as the current iteration of the solution and write U for the vector whose components are the coefficients u_i. Similarly, we write u^* for the next iterate and U^* for the vector of its coefficients. Then a single step of Newton is given by:

$$\mathrm{U}^* = \mathrm{U} - J^{-1}\mathrm{F}(u)$$

where $\mathrm{F}(u)_i = \hat{F}(u;\phi_i)$ and J is the Jacobian matrix defined by the Gâteaux derivative of F:

$$dF(u;\tilde{u},v) = \lim_{h\rightarrow 0}\frac{\hat{F}(u+h\tilde{u};v) - \hat{F}(u;v)}{h} \quad \forall v,\tilde{u}\in V$$

The actual Jacobian matrix is given by:

$$J_{ij} = dF(u;\phi_i,\phi_j)$$

where ϕ_i, ϕ_j are the ith and jth basis functions of V. Our definition of the modified residual $\hat{F}$ produces some interesting results for the boundary condition rows of J:

$$J_{ij} = \begin{cases} 1 & i = j \text{ and } \phi_i \in \phi^\Gamma \\ 0 & i \neq j \text{ and } \phi_i \in \phi^\Gamma \end{cases}$$

In other words, the rows of J corresponding to the boundary condition nodes are replaced by the corresponding rows of the identity matrix. Note that this does not depend on the *value* that the boundary condition takes, only on the set of nodes to which it applies.

This means that if, as in Newton's method, we are solving the system:

$$J\hat{\mathrm{U}} = \mathrm{F}(u)$$

then we can immediately write that part of the solution corresponding to the boundary condition rows:

$$\hat{\mathrm{U}}_i = \mathrm{F}(u)_i \quad \forall i \text{ such that } \phi_i \in \phi^\Gamma.$$

Based on this, define:

$$\hat{\mathrm{U}}^\Gamma_i = \begin{cases} \mathrm{F}(u)_i & \phi_i \in \phi^\Gamma \\ 0 & otherwise. \end{cases}$$

Next, let's consider a 4-way decomposition of J. Define:

$$J^{00}_{ij} = \begin{cases} J_{ij} & \phi_i,\phi_j \in \phi^0 \\ 0 & \text{otherwise} \end{cases}$$

$$J^{0\Gamma}_{ij} = \begin{cases} J_{ij} = 0 & \phi_i \in \phi^0, \phi_j \in \phi^\Gamma \\ 0 & \text{otherwise} \end{cases}$$

$$J^{\Gamma 0}_{ij} = \begin{cases} J_{ij} & \phi_i \in \phi^\Gamma, \phi_j \in \phi^0 \\ 0 & \text{otherwise} \end{cases}$$

$$J^{\Gamma\Gamma}_{ij} = \begin{cases} J_{ij} = \delta_{ij} & \phi_i,\phi_j \in \phi^\Gamma \\ 0 & \text{otherwise} \end{cases}$$

Clearly we may write:

$$J = J^{00} + J^{0\Gamma} + \underbrace{J^{\Gamma 0}}_{=0} + J^{\Gamma\Gamma}$$

As an illustration, assume in some example that the boundary nodes are numbered first in the global system, followed by the remaining nodes. Then (disregarding parts of the matrices which are zero), we can write:

$$J = \begin{bmatrix} J^{\Gamma\Gamma} & J^{\Gamma 0} \\ J^{0\Gamma} & J^{00} \end{bmatrix} = \begin{bmatrix} \mathrm{I} & 0 \\ J^{0\Gamma} & J^{00} \end{bmatrix}$$

Note again that this is merely illustrative: the decomposition of J works in exactly the same way for any numbering of the nodes.

Using forward substitution, this enables us to rewrite the linear system as:

$$(J^{00} + J^{\Gamma\Gamma})\hat{\mathrm{U}} = \mathrm{F}(u) - J^{0\Gamma}\hat{\mathrm{U}}^{\Gamma}$$

We can now make two observations. First, the matrix $J^{00} + J^{\Gamma\Gamma}$ preserves the symmetry of J. That is to say, if J has any of the following properties, then $J^{00} + J^{\Gamma\Gamma}$ will too:

- symmetry
- positive (semi-)definiteness
- skew-symmetry
- diagonal dominance

Second, if the initial value of u passed into the Newton iteration satisfies the Dirichlet boundary conditions, then $\hat{\mathrm{U}}^{\Gamma} = 0$ at every stage of the algorithm. Hence the system to be solved at each iteration is:

$$(J^{00} + J^{\Gamma\Gamma})\hat{\mathrm{U}} = \mathrm{F}(u)$$

A similar argument applies to other nonlinear solution algorithms such as line search Newton.

1.3.3 Implementation

Variational problems

Both linear and nonlinear PDEs are solved in residual form in Firedrake using the PETSc SNES interface. In the case of linear systems, a single step of Newton is employed.

In the following we will use `F` for the residual `Form` and `J` for the Jacobian `Form`. In both cases these forms do not include the Dirichlet boundary conditions. Additionally `u` will be the solution *`Function`*.

Strong boundary conditions are applied as follows:

1. Before the solver starts, the initial value `u` provided by the user is modified at the boundary condition nodes to satisfy the boundary conditions.
2. Each time the solver assembles the Jacobian matrix, the following happens.

   a) `J` is assembled using modified indirection maps in which the boundary condition node indices have been replaced by negative values. PETSc interprets these negative indices as an instruction to drop the corresponding entry. The result is the matrix J^{00}.

   b) The boundary node row diagonal entries of `J` are set to 1. This produces the matrix $J^{00} + J^{\Gamma\Gamma}$

3. Each time the solver assembles the residual, the following happens.

   a) `F` is assembled using unmodified indirection maps taking no account of the boundary conditions. This results in an assembled residual which is correct on the non-boundary condition nodes but contains spurious values in the boundary condition entries.

   b) The entries of `F` corresponding to boundary condition nodes are set to zero.

Linear systems

Linear systems (i.e. systems in which the matrix is pre-assembled) are solved with boundary conditions as follows:

1. When the user calls `assemble(a)` to assemble the bilinear form `a`, no actual assembly takes place. Instead, Firedrake returns a *`Matrix`* object that records the fact that it is intended to be assembled from `a`.

2. At the *`solve()`* call, Firedrake determines which boundary conditions to apply in the following priority order: first, boundary conditions supplied to the *`solve()`* call. If no boundary conditions are supplied to the *`solve()`* call, then any boundary conditions applied when *`assemble()`* was called on A are used, as are any boundary conditions subsequently added with *`apply()`*.

3. In the linear system case, the Jacobian `Form` is `a`. Using this and the boundary conditions, Firedrake assembles and solves:

$$(J^{00} + J^{\Gamma\Gamma})\hat{\mathrm{U}} = \mathrm{F}(u) - J^{\Gamma 0}\hat{\mathrm{U}}^{\Gamma}$$

4. The matrix assembled is then stored in the *`Matrix`* so that reassembly is avoided if the matrix is used in another *`solve()`* call with the same boundary conditions.

1.4 The R space

The function space R (for "Real") is the space of functions which are constant over the whole domain. It is employed to model concepts such as global constraints.

1.4.1 An example:

Warning: This section illustrates the use of the Real space using the simplest example. This is usually not the optimal approach for removing the nullspace of an operator. If that is your only goal then you are probably better placed removing the null space in the linear solver using the facilities documented in the section *Solving singular systems*.

Consider a Poisson equation in weak form, find $u \in V$ such that:

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = -\int_{\Gamma(3)} v \,\mathrm{d}s + \int_{\Gamma(4)} v \,\mathrm{d}s \qquad \forall v \in V$$

where $\Gamma(3)$ and $\Gamma(4)$ are domain boundaries over which the boundary conditions $\nabla u \cdot n = -1$ and $\nabla u \cdot n = 1$ are applied respectively. This system has a null space composed of the constant functions. One way to remove this is to add a Lagrange multiplier from the space R and use the resulting constraint equation to enforce that the integral of u is zero. The resulting system is find $u \in V$, $r \in R$ such that:

$$\begin{aligned}\int_\Omega \nabla u \cdot \nabla v + rv \,\mathrm{d}x &= -\int_{\Gamma(3)} v \,\mathrm{d}s + \int_{\Gamma(4)} v \,\mathrm{d}s \qquad \forall v \in V\\ \int_\Omega us \,\mathrm{d}x &= 0 \qquad \forall s \in R\end{aligned}$$

The corresponding Python code is:

```python
from firedrake import *

m = UnitSquareMesh(25, 25)
V = FunctionSpace(m, 'CG', 1)
R = FunctionSpace(m, 'R', 0)
W = V * R
u, r = TrialFunctions(W)
v, s = TestFunctions(W)

a = inner(grad(u), grad(v))*dx + u*s*dx + v*r*dx
L = -v*ds(3) + v*ds(4)

w = Function(W)
solve(a == L, w)
u, s = split(w)
exact = Function(V)
x, y = SpatialCoordinate(m)
exact.interpolate(y - 0.5)
print(sqrt(assemble((u - exact)*(u - exact)*dx)))
```

1.4.2 Setting and retrieving the value of a function in R

Functions in the space R are equivalent to a single floating point value. The value can be set using the `assign()` method of Firedrake functions, and the value can be accessed simply by casting it to `float()`:

```
from firedrake import *

m = UnitSquareMesh(25, 25)
R = FunctionSpace(m, 'R', 0)

f = Function(V)
f.assign(2.0)
print(float(f))
```

Note: The R space is not currently supported in complex mode.

1.4.3 Representing matrices involving R

Functions in the space R are different from other finite element functions in that their support extends to the whole domain. To illustrate the consequences of this, we can represent the matrix in the Poisson problem above as:

$$A = \begin{bmatrix} L & K \\ K^T & 0 \end{bmatrix}$$

where:

$$L_{ij} = \int_\Omega \nabla\phi_i \phi_j \, \mathrm{d}x$$
$$K_{ij} = \int_\Omega \phi_i \psi_j \, \mathrm{d}x$$

where $\{\phi_i\}$ is the basis for V and $\{\psi_i\}$ is the basis for R. Note that there is only a single basis function for R and $\psi_i \equiv 1$ hence:

$$K_{ij} = \int_\Omega \phi_i \, \mathrm{d}x$$

with the result that K is a single dense matrix column. Similiarly, K^T is a single dense matrix row.

Using the CSR matrix format typically employed by Firedrake, each matrix row is stored on a single processor. Were this carried through to K^T, both the assembly and action of this row would require the entire system state to be gathered onto one MPI process. This is clearly a horribly non-performant option.

Instead, we observe that a dense matrix row (or column) is isomorphic to a *`Function`* and implement these matrix blocks accordingly.

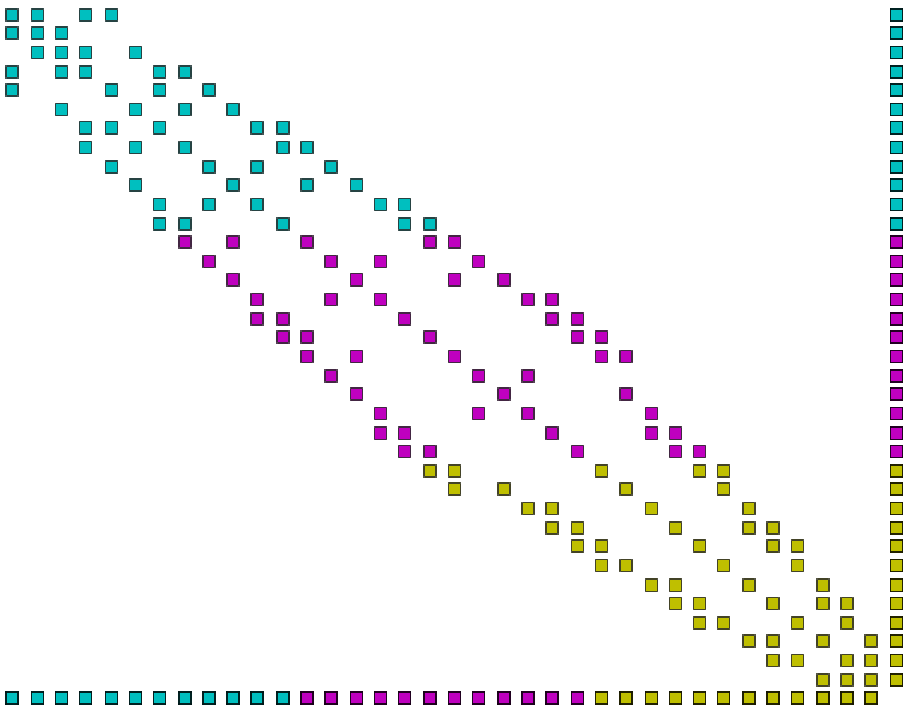


Figure1: Example parallel distribution of the matrix A. Colours indicate the processor on which the data is stored. Notice the dense row and column, and that the dense row is distributed across the processors.

1.4.4 Assembling matrices involving R

Assembling the column block is implemented by replacing the trial function with the constant 1, thereby transforming a 2-form into a 1-form, and assembling. Similarly, assembling the row block simply requires the replacement of the test function with the constant 1, and assembling.

The one by one block in the corner is assembled by replacing both the test and trial functions of the corresponding form with 1 and assembling. The remaining block does not involve R and is assembled as usual.

1.4.5 Using R space with extruded meshes

On extruded meshes it is possible to construct tensor product function spaces with the R space. Using the R space in the extruded direction provides a convenient way of expressing fields that are constant along the extrusion.

The example below illustrates how the R space can be used to compute a vertical average of a three-dimensional DG1 field by projecting the source field on a DG1 x R space.

```
from firedrake import *

mesh2d = UnitSquareMesh(10, 10)
mesh = ExtrudedMesh(mesh2d, 10, 0.1)

V = FunctionSpace(mesh, 'DG', 1, vfamily='DG', vdegree=1)
f = Function(V)
x, y, z = SpatialCoordinate(mesh)
f.interpolate(sin(2*pi*z))

U = FunctionSpace(mesh, 'DG', 1, vfamily='R', vdegree=0)
g = Function(U, name='g')
g.project(f)

print('f min: {:.3g}, max: {:.3g} '.format(f.dat.data.min(), f.dat.data.
↪max()))
print('g min: {:.3g}, max: {:.3g} '.format(g.dat.data.min(), g.dat.data.
↪max()))
```

1.5 Extruded Meshes in Firedrake

1.5.1 Introduction

Firedrake provides several utility functions for the creation of semi-structured meshes from an unstructured base mesh. Firedrake also provides a wide range of finite element spaces, both simple and sophisticated, for use with such meshes.

These meshes may be particularly appropriate when carrying out simulations on high aspect ratio domains. More mundanely, they allow a two-dimensional mesh to be built from square or rectangular cells.

The partial structure can be exploited to give performance advantages when iterating over the mesh, relative to a fully unstructured traversal of the same mesh. Firedrake exploits these benefits when extruded meshes are used.

Structured, Unstructured and Semi-Structured Meshes

Structured and unstructured meshes differ in the way the topology of the mesh is specified.

In a *fully structured* mesh, the array indices of mesh entities can be computed directly. For example, given the index of the current cell, the indices of the cell's vertices can be computed using a simple mathematical expression. This means that data can be directly addressed, using expressions of the form A[i].

In a *fully unstructured* mesh, there is no simple relation between the indices of different mesh entities. Instead, the relationships have to be explicitly stored. For example, given the index of the current cell, the indices of the cell's vertices can only be found by looking up the information in a separate array. It follows that data must be indirectly addressed, using expressions of the form A[B[i]].

Memory access latency makes indirect addressing more expensive than direct addressing: it is usually more efficient to compute the array index directly than to look it up from memory.

The characteristics of a *semi-structured* or *extruded* mesh lie somewhere between the two extremes above. An extruded mesh has an unstructured *base* mesh. Each cell of the base mesh corresponds to a *column* of cells in the extruded mesh. Visiting the first cell in each column requires indirect addressing. However, visiting subsequent cells in the column can be done using direct addressing. As the number of cells in the column increases, the performance should approach that of a fully structured mesh.

1.5.2 Generating Extruded Meshes in Firedrake

Extruded meshes are built using *`ExtrudedMesh()`*. There are several built-in extrusion types that generate commonly-used extruded meshes. To create a more complicated extruded mesh, one can either pass a hand-written kernel to *`ExtrudedMesh()`*, or one can use a built-in extrusion type and modify the coordinate field afterwards.

The following information may be passed in to the constructor:

- a `Mesh` object, which will be used as the base mesh.
- the desired number of cell layers in the extruded mesh. One may also specify layers per column, see below for more information.
- the `extrusion_type`, which can be one of the built-in "uniform", "radial" or "radial_hedgehog" – these are described below – or "custom". If this argument is omitted, the "uniform" extrusion type will be used.
- the `layer_height`, which is needed for the built-in extrusion types.
- a `kernel`, only if the custom extrusion type is used
- the appropriate `gdim`, describing the geometric dimension of the mesh, only if the custom extrusion type is used.

Uniform Extrusion

Uniform extrusion adds another spatial dimension to the mesh. For example, a 2D base mesh becomes a 3D extruded mesh. The coordinates of the extruded mesh are computed on the assumption that the layers are evenly spaced (hence the word 'uniform').

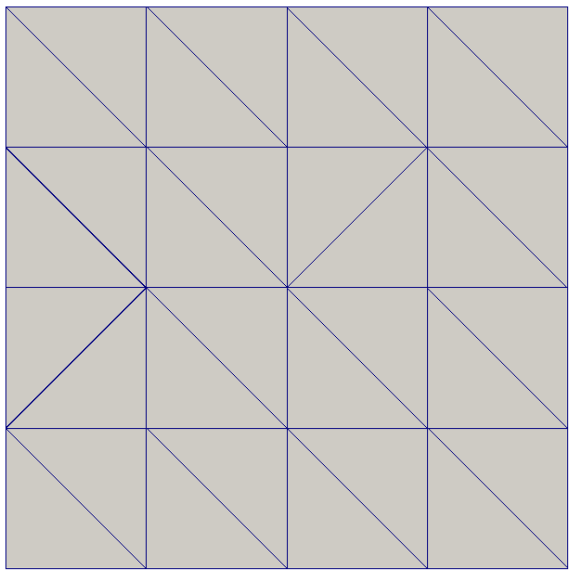
Let `m` be a standard *UnitSquareMesh()*. The following code produces the extruded mesh, whose base mesh is `m`, with 5 mesh layers and a layer thickness of 0.2:

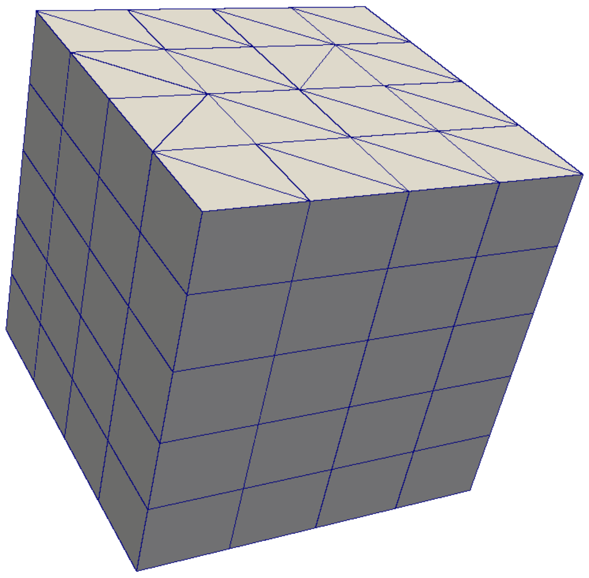
```
m = UnitSquareMesh(4, 4)
mesh = ExtrudedMesh(m, 5, layer_height=0.2, extrusion_type='uniform')
```

This can be simplified slightly. The extrusion_type defaults to 'uniform', so this can be omitted. Furthermore, the layer_height, if omitted, defaults to the reciprocal of the number of layers. The following code therefore has the same effect:

```
m = UnitSquareMesh(4, 4)
mesh = ExtrudedMesh(m, 5)
```

The base mesh and extruded mesh are shown below.



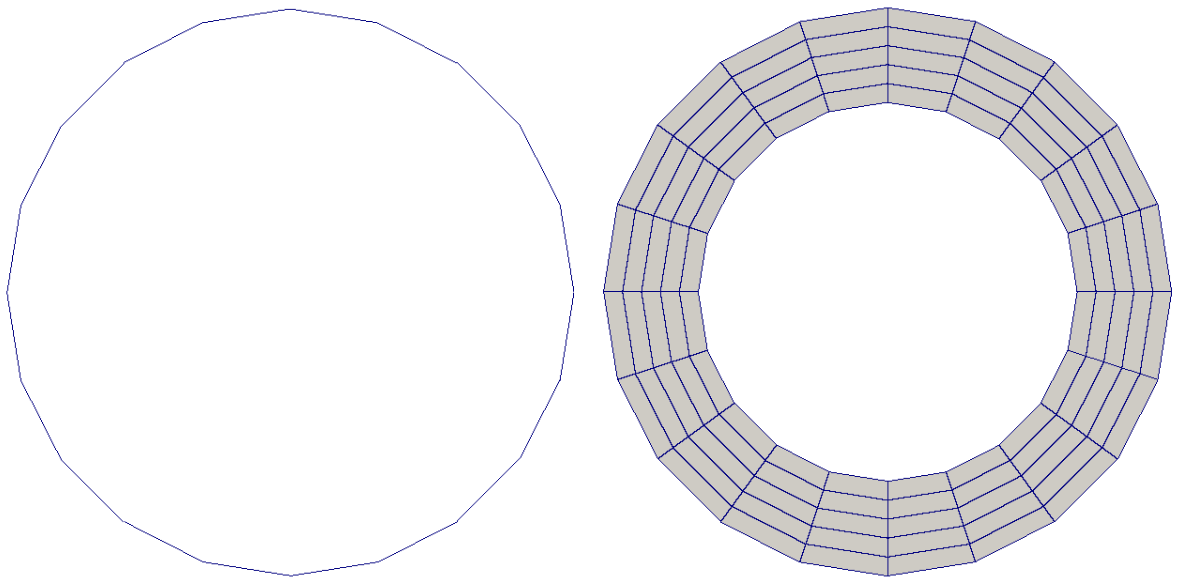


Radial Extrusion

Radial extrusion extrudes cells radially outwards from the origin, without increasing the number of spatial dimensions. An example in 2 dimensions, in which a circle is extruded into an annulus, is:

```
m = CircleManifoldMesh(20, radius=2)
mesh = ExtrudedMesh(m, 5, extrusion_type='radial')
```

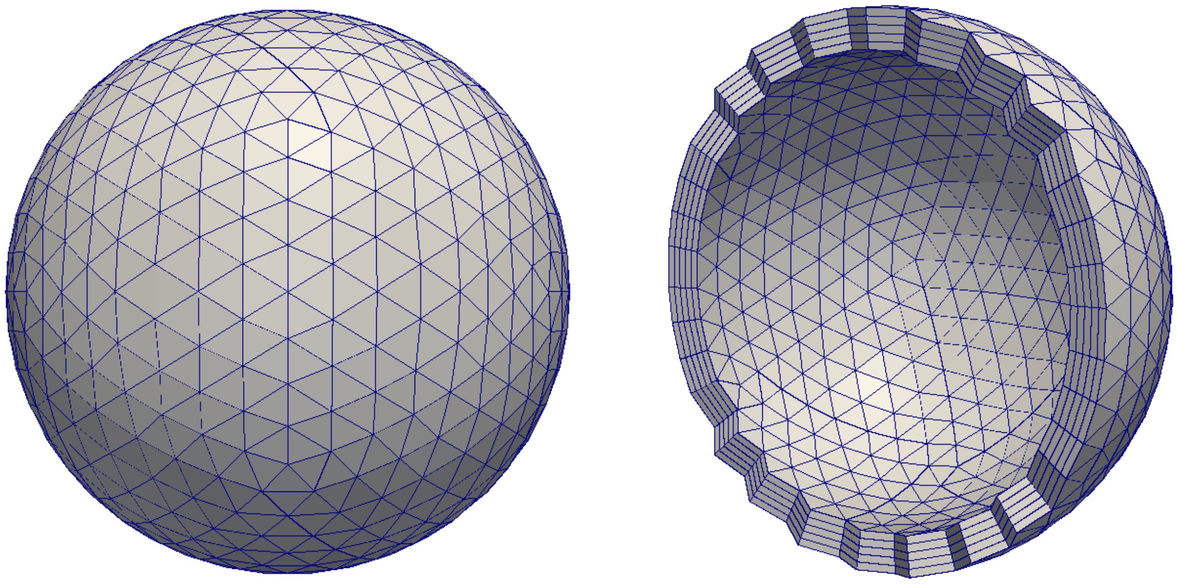
The base mesh and extruded mesh are shown below.



An example in 3 dimensions, in which a sphere is extruded into a spherical annulus, is:

```
m = IcosahedralSphereMesh(radius=3, refinement_level=3)
mesh = ExtrudedMesh(m, 5, layer_height=0.1, extrusion_type='radial')
```

The base mesh and part of the extruded mesh are shown below.



Hedgehog Extrusion

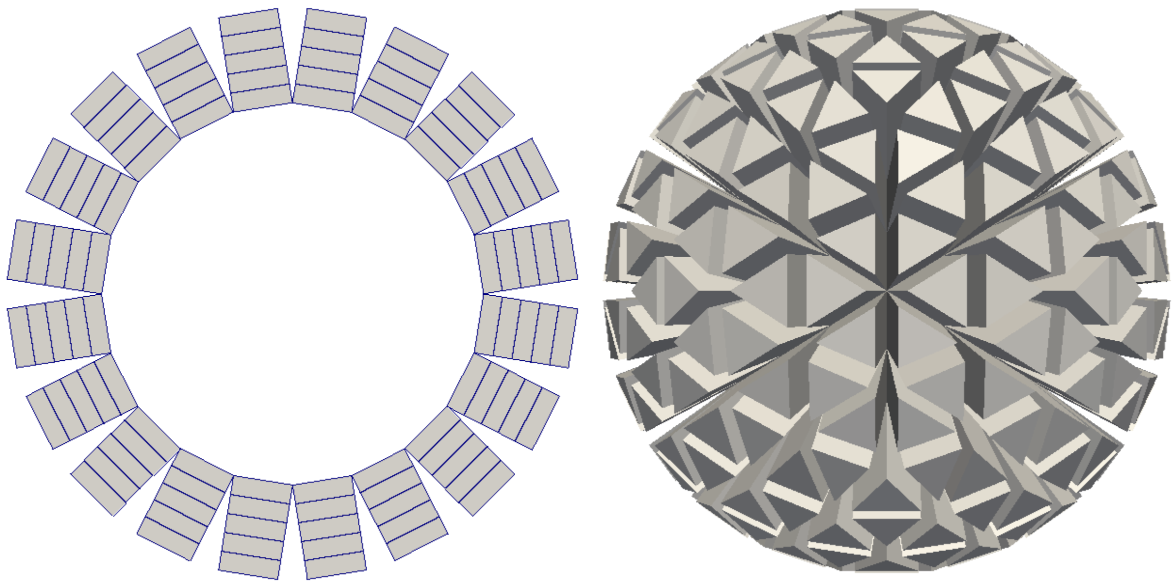
Hedgehog extrusion is similar to radial extrusion, but the cells are extruded outwards in a direction normal to the base cell. This produces a discontinuous coordinate field.

```
m = CircleManifoldMesh(20, radius=2)
mesh = ExtrudedMesh(m, 5, extrusion_type='radial_hedgehog')
```

An example in 3 dimensions, in which a sphere is extruded into a spherical annulus, is:

```
m = UnitIcosahedralSphereMesh(refinement_level=2)
mesh = ExtrudedMesh(m, 5, layer_height=0.1, extrusion_type='radial_hedgehog')
```

The 2D and 3D hedgehog-extruded meshes are shown below.



Custom Extrusion

For a more complicated extruded mesh, a custom *kernel* can be given by the user. Since this is a mesh-wide operation, a PyOP2 parallel loop is constructed by Firedrake.

```
m = UnitSquareMesh(5, 5)
kernel = op2.Kernel("""
    void extrusion_kernel(double **base_coords, double **ext_coords,
                          double *layer_height, int layer) {
        for (int i=0; i<6; i++) {
            ext_coords[i][0] = base_coords[i / 2][0]; // X
            ext_coords[i][1] = base_coords[i / 2][1]; // Y
            ext_coords[i][2] = 0.1 * (layer + (i % 2)) + 0.5 * base_coords[i /
↪ 2][1]; // Z
        }
    }
""", "extrusion_kernel")
mesh = ExtrudedMesh(m, 5, extrusion_type='custom', kernel=kernel, gdim=3)
```

Variable numbers of mesh cell layers

The simplest method of creating an extruded mesh is to provide a constant number of cell layers for every cell in the base mesh. For some applications, this may not provide sufficient flexibility. Firedrake therefore also allows creation of extruded meshes with a different number of cells in each cell column. To do this, we provide an array with two values for each cell in the mesh. The first entry is the number cells offset from the "bottom" (zero) level, the second is the number of cells in the column.

For example, we might create this extruded mesh:

```
mesh = UnitIntervalMesh(3)
extmesh = ExtrudedMesh(mesh, layers=[[0, 2], [1, 1], [2, 1]],
                              layer_height=0.25)
```

which results in the following mesh topology.:

```
                    x--------x
                    |        |
                    |        |
                    |        |
                    |        |
x--------x--------x--------x
|        |        |
|        |        |
|        |        |
|        |        |
x--------x--------x
|        |
|        |
|        |
|        |
x--------x
```

To simplify the implementation, we never iterate over the interior facets that only have cells on one side. When you construct the mesh, you should arrange that these facets have zero area, by squashing the coordinates together. In addition, we require that the resulting extruded mesh does not contain topologically disconnected columns: offset cells must, at least, share a vertex with some other cell.

Note: When running in parallel, the base mesh will be distributed before the extruded mesh is created. So you should arrange that the layers array that you provide is specified accordingly (matching the parallel distribution).

For more details on the implementation, see `firedrake.extrusion_numbering`.

1.5.3 Function Spaces on Extruded Meshes

The syntax for building a *FunctionSpace* on an extruded mesh is an extension of the existing syntax used with normal meshes. On a non-extruded mesh, the following syntax is used:

```
mesh = UnitSquareMesh(4, 4)
V = FunctionSpace(mesh, "RT", 1)
```

To allow maximal flexibility in constructing function spaces, Firedrake supports a more general syntax:

```
V = FunctionSpace(mesh, element)
```

where `element` is a UFL `FiniteElement` object. This requires generation and manipulation of `FiniteElement` objects.

Geometrically, an extruded mesh cell is the *product* of a base, "horizontal", cell with a "vertical" interval. The construction of function spaces on extruded meshes makes use of this. Firedrake supports all function spaces whose local element can be expressed as the product of an element defined on the base cell with an element defined on an interval.

We will now introduce the new operators which act on `FiniteElement` objects.

The `TensorProductElement` operator

To create an element compatible with an extruded mesh, one should use the `TensorProductElement` operator. For example,

```
horiz_elt = FiniteElement("CG", triangle, 1)
vert_elt = FiniteElement("CG", interval, 1)
elt = TensorProductElement(horiz_elt, vert_elt)
V = FunctionSpace(mesh, elt)
```

will give a continuous, scalar-valued function space. The resulting space contains functions which vary linearly in the horizontal direction and linearly in the vertical direction.

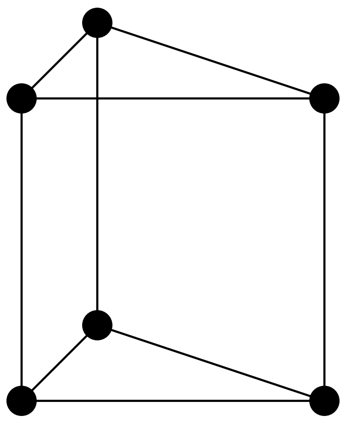


Figure2: The product of a CG1 triangle element with a CG1 interval element

The degree and continuity may differ; for example

```
horiz_elt = FiniteElement("DG", triangle, 0)
vert_elt = FiniteElement("CG", interval, 2)
elt = TensorProductElement(horiz_elt, vert_elt)
V = FunctionSpace(mesh, elt)
```

will give a function space which is continuous between cells in a column, but discontinuous between horizontally-neighbouring cells. In addition, the function may vary piecewise-quadratically in the vertical direction, but is piecewise constant horizontally.

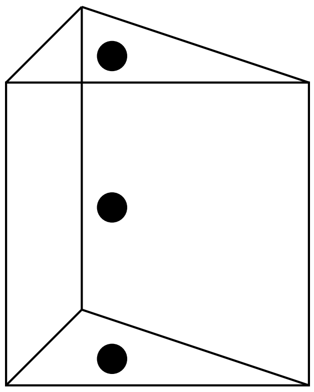


Figure3: The product of a DG0 triangle element with a CG2 interval element

A more complicated element, like a Mini horizontal element with linear variation in the vertical direction, may be built using the `EnrichedElement` functionality in either of the following ways:

```
mini_horiz_1 = FiniteElement("CG", triangle, 1)
mini_horiz_2 = FiniteElement("B", triangle, 3)
mini_horiz = mini_horiz_1 + mini_horiz_2  # Enriched element
mini_vert = FiniteElement("CG", interval, 1)
mini_elt = TensorProductElement(mini_horiz, mini_vert)
V = FunctionSpace(mesh, mini_elt)
```

or

```
mini_horiz_1 = FiniteElement("CG", triangle, 1)
mini_horiz_2 = FiniteElement("B", triangle, 3)
mini_vert = FiniteElement("CG", interval, 1)
mini_elt_1 = TensorProductElement(mini_horiz_1, mini_vert)
mini_elt_2 = TensorProductElement(mini_horiz_2, mini_vert)
mini_elt = mini_elt_1 + mini_elt_2  # Enriched element
V = FunctionSpace(mesh, mini_elt)
```

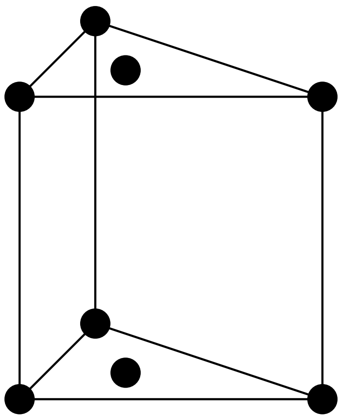


Figure4: The product of a Mini triangle element with a CG1 interval element

The `HDivElement` and `HCurlElement` operators

For moderately complicated vector-valued elements, `TensorProductElement` does not give enough information to unambiguously produce the desired space. As an example, consider the lowest-order *Raviart-Thomas* element on a quadrilateral. The degrees of freedom live on the facets, and consist of a single evaluation of the component of the vector field normal to each facet.

The following element is closely related to the desired Raviart-Thomas element:

```
CG_1 = FiniteElement("CG", interval, 1)
DG_0 = FiniteElement("DG", interval, 0)
P1P0 = TensorProductElement(CG_1, DG_0)
P0P1 = TensorProductElement(DG_0, CG_1)
elt = P1P0 + P0P1
```

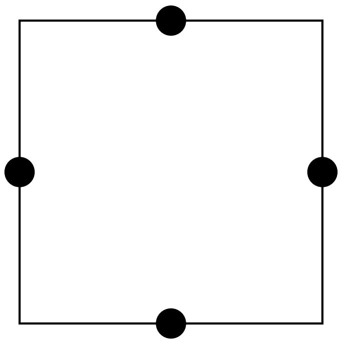


Figure5: The element created above

However, this is only scalar-valued. There are two natural vector-valued elements that can be generated from this: one of them preserves tangential continuity between elements, and the other preserves normal continuity between elements. To obtain the Raviart-Thomas element, we must use the `HDivElement` operator:

```
CG_1 = FiniteElement("CG", interval, 1)
DG_0 = FiniteElement("DG", interval, 0)
P1P0 = TensorProductElement(CG_1, DG_0)
```

(continues on next page)

(continued from previous page)

```
RT_horiz = HDivElement(P1P0)
P0P1 = TensorProductElement(DG_0, CG_1)
RT_vert = HDivElement(P0P1)
elt = RT_horiz + RT_vert
```

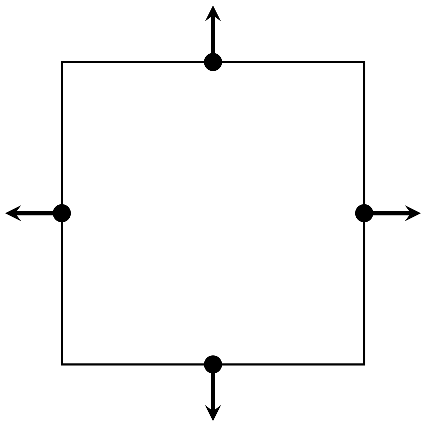


Figure6: The RT quadrilateral element, requiring the use of `HDivElement`

Another reason to use these operators is when expanding a vector into a higher dimensional space. Consider the lowest-order Nedelec element of the 2nd kind on a triangle:

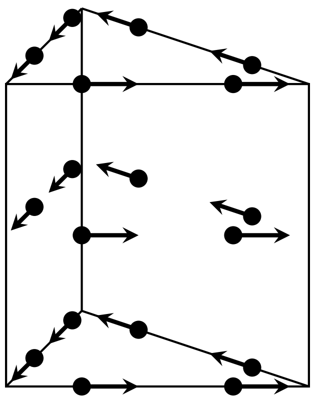
```
N2_1 = FiniteElement("N2curl", triangle, 1)
```

This is naturally vector-valued, and has two components. Suppose we form the product of this with a continuous element on an interval:

```
CG_2 = FiniteElement("CG", interval, 2)
N2CG = TensorProductElement(N2_1, CG_2)
```

This element still only has two components. To expand this into a three-dimensional curl-conforming element, we must use the `HCurlElement` operator; the syntax is:

```
Ned_horiz = HCurlElement(N2CG)
```



This gives the horizontal part of a Nedelec edge element on a triangular prism. The full element can be built as follows:

```
N2_1 = FiniteElement("N2curl", triangle, 1)
CG_2 = FiniteElement("CG", interval, 2)
N2CG = TensorProductElement(N2_1, CG_2)
Ned_horiz = HCurlElement(N2CG)
P2tr = FiniteElement("CG", triangle, 2)
P1dg = FiniteElement("DG", interval, 1)
P2P1 = TensorProductElement(P2tr, P1dg)
Ned_vert = HCurlElement(P2P1)
Ned_wedge = Ned_horiz + Ned_vert
V = FunctionSpace(mesh, Ned_wedge)
```

Shortcuts for simple spaces

Simple scalar-valued spaces can be created using a variation on the existing syntax, if the `HDivElement`, `HCurlElement` and enrichment operations are not required. To create a function space of degree 2 in the horizontal direction, degree 1 in the vertical direction and possibly discontinuous between layers, the short syntax is

```
fspace = FunctionSpace(mesh, "CG", 2, vfamily="DG", vdegree=1)
```

If the horizontal and vertical parts have the same `family` and `degree`, the `vfamily` and `vdegree` arguments may be omitted. If `mesh` is an `ExtrudedMesh` then the following are equivalent:

```
fspace = FunctionSpace(mesh, "Lagrange", 1)
```

and

```
fspace = FunctionSpace(mesh, "Lagrange", 1, vfamily="Lagrange", vdegree=1)
```

1.5.4 Solving Equations on Extruded Meshes

Once the mesh and function spaces have been declared, extruded meshes behave almost identically to normal meshes. However, there are some small differences, which are listed below.

1. Surface integrals are no longer denoted by `ds`. Since extruded meshes have multiple types of surfaces, the following notation is used:

    - `ds_v` is used to denote an integral over *side* facets of the mesh. This can be combined with boundary markers from the base mesh, such as `ds_v(1)`.
    - `ds_t` is used to denote an integral over the *top* surface of the mesh.
    - `ds_b` is used to denote an integral over the *bottom* surface of the mesh.
    - `ds_tb` is used to denote an integral over both the *top* and *bottom* surfaces of the mesh.

2. Interior facet integrals are no longer denoted by `dS`. The *horizontal* and *vertical* interior facets may require different numerical treatment. To facilitate this, the following notation is used:

   - `dS_h` is used to denote an integral over *horizontal* interior facets (between cells that are vertically-adjacent).
   - `dS_v` is used to denote an integral over *vertical* interior facets (between cells that are horizontally-adjacent).

3. When setting strong boundary conditions, the boundary markers from the base mesh can be used to set boundary conditions on the relevant side of the extruded mesh. To set boundary conditions on the top or bottom, the label is replaced by:

   - `top`, to set a boundary condition on the top surface.
   - `bottom`, to set a boundary condition on the bottom surface.

   Note that for extruded meshes, the label `on_boundary` only refers to the side boundaries that take their labels from the base mesh, and not the top or bottom boundaries.

1.6 Changing mesh coordinates

Users may want to change the coordinates of an existing mesh object for certain reasons. The coordinates can be accessed as a *Function* through `mesh.coordinates` where `mesh` is a mesh object. For example,

```
mesh.coordinates.dat.data[:, 1] *= 2.0
```

streches the mesh in the y-direction. Another possibility is to use *assign()*:

```
Vc = mesh.coordinates.function_space()
x, y = SpatialCoordinate(mesh)
f = Function(Vc).interpolate(as_vector([x, y*2.0]))
mesh.coordinates.assign(f)
```

This can also be used if *f* is a solution to a PDE.

Warning: Features which rely on the coordinates field of a mesh's PETSc `DM` (usually a `DMPlex`) such as *VertexOnlyMesh()* and *MeshHierarchy()* will not work as expected if the `mesh.coordinates` field has been modified: at present, the this does not correspondingly update the coordinates field of the `DM`. This will be fixed in a future Firedrake update.

1.6.1 Changing the coordinate function space

For more complicated situations, one might wish to replace the mesh coordinates with a field which lives on a different *FunctionSpace* (e.g. higher-order meshes).

Note: Re-assigning the `coordinates` property of a mesh used to be an undocumented feature. However, this no longer works:

```
mesh.coordinates = f  # Raises an exception
```

Instead of re-assigning the coordinates of a mesh, one can create new mesh object from a field *f*:

```
new_mesh = Mesh(f)
```

`new_mesh` has the same mesh topology as the original mesh, but its coordinate values and coordinate function space are from *f*. The coordinate function space must be a rank-1 *FunctionSpace*, constructed either with *VectorFunctionSpace()*, or by providing a `VectorElement` to *FunctionSpace()*. For efficiency, the new mesh object shares data with *f*. That is, changing the values of *f* will change the coordinate values of the mesh, and *vice versa*. If this behaviour is undesired, one should explicitly copy:

```
g = Function(f)  # creates a copy of f
new_mesh = Mesh(g)
```

Or simply:

```
new_mesh = Mesh(Function(f))
```

1.6.2 Replacing the mesh geometry of an existing function

Creating a new mesh geometry object, as described above, leaves any existing *Function*s untouched – they continue to live on their original mesh geometries. One may wish to move these functions over to the new mesh. To move *f* over to `mesh`, use:

```
g = Function(functionspaceimpl.WithGeometry.create(f.function_space(), mesh),
             val=f.topological)
```

This creates a *Function* *g* which shares data with *f*, but its mesh geometry is `mesh`.

Warning: The example above uses Firedrake internal APIs, which might change in the future.

1.7 Interpolation

Firedrake offers various ways to interpolate expressions onto fields (*Function*s). Interpolation is often used to set up initial conditions and/or boundary conditions. The basic syntax for interpolation is:

```
# create new function f on function space V
f = interpolate(expression, V)

# alternatively:
f = Function(V).interpolate(expression)

# setting the values of an existing function
f.interpolate(expression)
```

Note: Interpolation is supported for most, but not all, of the elements that Firedrake provides. In particular, higher-continuity elements such as Argyris and Hermite do not presently support interpolation.

The recommended way to specify the source expression is UFL. UFL produces clear error messages in case of syntax or type errors, yet UFL expressions have good run-time performance, since they are translated to C interpolation kernels using TSFC technology. Moreover, UFL offers a rich language for describing expressions, including:

- The coordinates: in physical space as `SpatialCoordinate`, and in reference space as `ufl.geometry.CellCoordinate`.
- Firedrake *Function*s, derivatives of *Function*s, and *Constant*s.
- Literal numbers, basic arithmetic operations, and also mathematical functions such as `sin`, `cos`, `sqrt`, `abs`, etc.
- Conditional expressions using UFL `conditional`.
- Compound expressions involving any of the above.

Here is an example demonstrating some of these features:

```
# g is a vector-valued Function, e.g. on an H(div) function space
f = interpolate(sqrt(3.2 * div(g)), V)
```

This also works as expected when interpolating into a a space defined on the facets of the mesh:

```
# where trace is a trace space on the current mesh:
f = interpolate(expression, trace)
```

1.7.1 Interpolator objects

Firedrake is also able to generate reusable *Interpolator* objects which provide caching of the interpolation operation. The following line creates an interpolator which will interpolate the current value of *expression* into the space *V*:

```
interpolator = Interpolator(expression, V)
```

If *expression* does not contain a `TestFunction()` then the interpolation can be performed with:

```
f = interpolator.interpolate()
```

Alternatively, one can use the interpolator to set the value of an existing *Function*:

```
f = Function(V)
interpolator.interpolate(output=f)
```

If *expression* does not contain a `TestFunction()` then the interpolator acts to interpolate *Function*s in the test space to those in the target space. For example:

```
w = TestFunction(W)
interpolator = Interpolator(w, V)
```

Here, *interpolator* acts as the interpolation matrix from the *FunctionSpace()* W into the *FunctionSpace()* V. Such that if *f* is a *Function* in *W* then *interpolator(f)* is its interpolation into *g*. As before, the *output* parameter can be used to write into an existing *Function*. Passing the *transpose=True* option to *interpolate()* will cause the transpose interpolation to occur. This is equivalent to the multigrid restriction operation which interpolates assembled 1-forms in the dual space to *V* to assembled 1-forms in the dual space to *W*.

1.7.2 Interpolation from external data

Unfortunately, UFL interpolation is not applicable if some of the source data is not yet available as a Firedrake *Function* or UFL expression. Here we describe a recipe for moving external to Firedrake fields.

Let us assume that there is some function `mydata(X)` which takes as input an $n \times d$ array, where n is the number of points at which the data values are needed, and d is the geometric dimension of the mesh. `mydata(X)` shall return a n long vector of the scalar values evaluated at the points provided. (Assuming that the target *FunctionSpace* is scalar valued, although this recipe can be extended to vector or tensor valued fields.) Presumably `mydata` works by interpolating the external data source, but the precise details are not relevant now. In this case, interpolation into a target function space `V` proceeds as follows:

```
# First, grab the mesh.
m = V.ufl_domain()

# Now make the VectorFunctionSpace corresponding to V.
W = VectorFunctionSpace(m, V.ufl_element())

# Next, interpolate the coordinates onto the nodes of W.
X = interpolate(m.coordinates, W)

# Make an output function.
f = Function(V)

# Use the external data function to interpolate the values of f.
f.dat.data[:] = mydata(X.dat.data_ro)
```

This will also work in parallel, as the interpolation will occur on each process, and Firedrake will take care of the halo updates before the next operation using `f`.

1.7.3 C string expressions

Warning: C string expressions were a FEniCS compatibility feature which has now been removed. Users should use UFL expressions instead. This section only remains to assist in the transition of existing code.

Here are a couple of old-style C string expressions, and their modern replacements.

```
# Expression:
f = interpolate(Expression("sin(x[0]*pi)"), V)

# UFL equivalent:
x = SpatialCoordinate(V.mesh())
f = interpolate(sin(x[0] * math.pi), V)

# Expression with a Constant parameter:
f = interpolate(Expression('sin(x[0]*t)', t=t), V)

# UFL equivalent:
x = SpatialCoordinate(V.mesh())
f = interpolate(sin(x[0] * t), V)
```

1.7.4 Python expression classes

Warning: Python expression classes were a FEniCS compatibility feature which has now been removed. Users should use UFL expressions instead. This section only remains to assist in the transition of existing code.

Since Python `Expression` classes expressions are deprecated, below are a few examples on how to replace them with UFL expressions:

```
# Python expression:
class MyExpression(Expression):
    def eval(self, value, x):
        value[:] = numpy.dot(x, x)

    def value_shape(self):
        return ()

f.interpolate(MyExpression())

# UFL equivalent:
x = SpatialCoordinate(f.function_space().mesh())
f.interpolate(dot(x, x))
```

1.7.5 Generating Functions with randomised values

The *randomfunctiongen* module wraps the external package randomgen, which gives Firedrake users an easy access to many stochastically sound random number generators, including `PCG64`, `Philox`, and `ThreeFry`, which are parallel-safe. All distribution methods defined in randomgen are made available, and one can pass a *FunctionSpace* to most of these methods to generate a randomised *Function*.

```
mesh = UnitSquareMesh(2,2)
V = FunctionSpace(mesh, "CG", 1)
# PCG64 random number generator
pcg = PCG64(seed=123456789)
rg = RandomGenerator(pcg)
# beta distribution
f_beta = rg.beta(V, 1.0, 2.0)

print(f_beta.dat.data)

# produces:
# [0.56462514 0.11585311 0.01247943 0.398984 0.19097059 0.5446709 0.1078666 0.
↪2178807 0.64848515]
```

1.8 Point evaluation

Firedrake can evaluate *Function*s at arbitrary physical points. This feature can be useful for the evaluation of the result of a simulation. Two APIs are offered to this feature: a Firedrake-specific one, and one from UFL.

1.8.1 Firedrake API

Firedrake offers a convenient API for evaluating functions at arbitrary points via *at()*:

```
# evaluate f at a 1-dimensional point
f.at(0.3)

# evaluate f at two 1-dimensional points, or at one 2-dimensional point
# (depending on f's geometric dimension)
f.at(0.2, 0.4)

# evaluate f at one 2-dimensional point
f.at([0.2, 0.4])

# evaluate f at two 2-dimensional point
f.at([0.2, 0.4], [1.2, 0.5])

# evaluate f at two 2-dimensional point (same as above)
f.at([[0.2, 0.4], [1.2, 0.5]])
```

While in these examples we have only shown lists, other *iterables* such as tuples and `numpy` arrays are also accepted. The following are equivalent:

```
f.at(0.2, 0.4)
f.at((0.2, 0.4))
f.at([0.2, 0.4])
f.at(numpy.array([0.2, 0.4]))
```

For a single point, the result is a `numpy` array, or a tuple of `numpy` arrays in case of *mixed* functions. When evaluating multiple points, the result is a list of values for each point. To summarise:

- Single point, non-mixed: `numpy` array
- Single point, mixed: tuple of `numpy` arrays
- Multiple points, non-mixed: list of `numpy` arrays
- Multiple points, mixed: list of tuples of `numpy` arrays

Points outside the domain

When any point is outside the domain of the function, *`PointNotInDomainError`* exception is raised. If `dont_raise=True` is passed to *`at()`*, the result is `None` for those points which fall outside the domain.

```
mesh = UnitIntervalMesh(8)
f = mesh.coordinates

f.at(1.2)                    # raises exception
f.at(1.2, dont_raise=True)   # returns None

f.at(0.5, 1.2)                    # raises exception
f.at(0.5, 1.2, dont_raise=True)   # returns [0.5, None]
```

Warning: Point evaluation on *immersed manifolds* is not supported yet, due to the difficulty of specifying a physical point on the manifold.

Evaluation on a moving mesh

If you move the mesh, by *changing the mesh coordinates*, then the bounding box tree that Firedrake maintains to ensure fast point evaluation must be rebuilt. To do this, after moving the mesh, call `clear_spatial_index()` on the mesh you have just moved.

Evaluation with a distributed mesh

There is limited support for point evaluation when running Firedrake in parallel. There is no special API, but there are some restrictions:

- Point evaluation is a *collective* operation.
- Each process must ask for the same list of points.
- Each process will get the same values.

1.8.2 UFL API

UFL reserves the function call operator for evaluation:

```
f([0.2, 0.4])
```

will evaluate f at $(0.2, 0.4)$. UFL does not accept multiple points at once, and cannot configure what to do with a point which is not in the domain. The advantage of this syntax is that it works on any `Expr`, for example:

```
(f*sin(f)([0.2, 0.4])
```

will evaluate $f \cdot \sin(f)$ at $(0.2, 0.4)$.

Note: The expression itself is not translated into C code. While the evaluation of a function uses the same infrastructure as the Firedrake API, which uses generated C code, the expression tree is evaluated by UFL in Python.

1.9 Visualising the results of simulations

Having run a simulation, it is likely that we will want to look at the results. To do this, Firedrake supports saving data in VTK format, suitable for visualisation in Paraview (amongst others).

In addition, 1D and 2D function could be plotted and displayed using the python library of matplotlib (an optional dependency of firedrake)

1.9.1 Creating output files

Output for visualisation purposes is managed with a *`File`* object. To create one, we just need to pass the name of the output file on disk. The file Firedrake creates is in PVD and therefore the requested file name must end in `.pvd`.

```
outfile = File("output.pvd")
# The following raises an error
badfile = File("output.vtu")
```

To save functions to the *`File`* we use the *`write()`* method.

```
mesh = UnitSquareMesh(1, 1)
V = FunctionSpace(mesh, "DG", 0)
f = Function(V)
f.interpolate(sin(SpatialCoordinate(mesh)[0]))

outfile = File("output.pvd")
outfile.write(f)
```

Note: Output created for visualisation purposes is not intended for purposes other than visualisation. If you need to save data for checkpointing purposes, you should instead use Firedrake's *checkingpointing capabilities*.

1.9.2 Saving time-dependent data

Often, we have a time-dependent simulation and would like to save the same function at multiple timesteps. This is straightforward, we must create the output *File* outside the time loop and call *write()* inside.

```
...
outfile = File("timesteps.pvd")

while t < T:
    ...
    outfile.write(f)
    t += dt
```

The PVD data format supports specifying the timestep value for time-dependent data. We do not have to provide it to *write()*, by default an integer counter is used that is incremented by 1 each time *write()* is called. It is possible to override this by passing the keyword argument `time`.

```
...
outfile = File("timesteps.pvd")

while t < T:
    ...
    outfile.write(f, time=t)
    t += dt
```

1.9.3 Visualising high-order data

The file format Firedrake outputs to currently supports the visualisation of scalar-, vector-, or tensor-valued fields represented with an arbitrary order (possibly discontinuous) Lagrange basis. Furthermore, the fields must be in an isoparametric function space, meaning the *mesh coordinates* associated to a field must be represented with the same basis as the field. To visualise fields in anything other than these spaces we must transform the data to this format first. One option is to do so by hand before outputting. Either by *interpolating* or else `projecting` the *mesh coordinates* and then the field. Since this is such a common operation, the *File* object is set up to manage these operations automatically, we just need to choose whether we want data to be interpolated or projected. The default is to use interpolation. For example, assume we wish to output a vector-valued function that lives in an $H(\mathrm{div})$ space. If we want it to be interpolated in the output file we can use

```
V = FunctionSpace(mesh, "RT", 2)
f = Function(V)
...
outfile = File("output.pvd")
outfile.write(f)
```

If instead we want projection, we use

```
projected = File("proj_output.pvd", project_output=True)
projected.write(f)
```

Note: This feature requires Paraview version 5.5.0 or better. If you must use an older version of Paraview, you must manually interpolate mesh coordinates and field coordinates to a piecewise linear function space, represented with either a Lagrange (H1) or discontinuous Lagrange (L2) basis. The *File* is also setup to manage this issue. For instance, we can force the output to be discontinuous piecewise linears via

```
projected = File("proj_output.pvd", target_degree=1, target_continuity=H1)
projected.write(f)
```

Using Paraview on higher order data

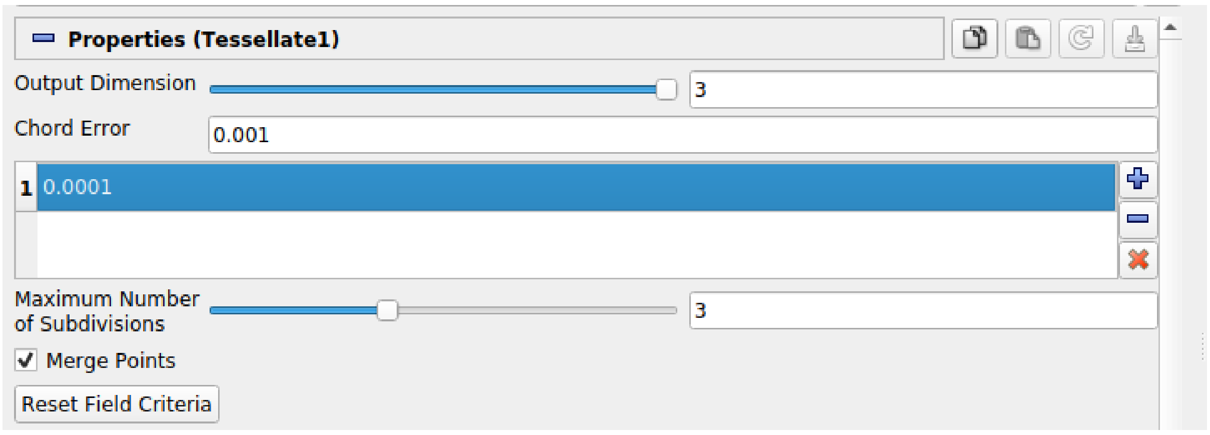
Paraview's visualisation algorithims are typically exact on piecewise linear data, but if you write higher order data, Paraview will produce an approximate visualisation. This approximation can be controlled in at least two ways:

1. Under the display properties of an unstructured grid, the Nonlinear Subdivision Level can be increased; this option controls the display of unstructured grid data and can be used to present a plausible curved geometry. Further, the Nonlinear Subdivision Level can also be changed after applying filters such as Extract Surface.
2. The Tessellate filter can be applied to unstructured grid data and has three parameters: Chord Error, Maximum Number of Subdivisions, and Field Error. Tessellation is the process of approximating a higher order geometry via subdividing cells into smaller linear cells. Chord Error is a tessellation error metric, the distance between the midpoint of any edge on the tessellated geometry and a corresponding point in the original geometry.

> Field Error is analogous to Chord Error: the error of the field on the tessellated data is compared pointwise to the original data at the midpoints of the edges of the tessellated geometry and the corresponding points on the original geometry. The Maximum Number of Subdivisions is the maximum number of times an edge in the original geometry can be subdivided.

Besides the two tools listed above, Paraview provides many other tools (filters) that might be applied to the original data or composed with the tools listed above. Documentation on these interactions is sparse, but tessellation can be used to understand this issue: the Tessellate filter produces another unstructured grid from its inputs so algorithms can be applied to both the tessellated and input unstructured grid. The tessellated data can also be saved for future reference.

Note: Field Error is hidden in the current Paraview UI (5.7) so we include a visual guide wherein the field error is set via the highlighted field directly below Chord Error:


Properties (Tessellate1)

Output Dimension 3

Chord Error 0.001

1 0.0001

Maximum Number of Subdivisions 3

✓ Merge Points

Reset Field Criteria

We also note that the Tessellate filter (and other filters) can be more clearly controlled via the Paraview Python shell (under the View menu). For instance, Field Error can be more clearly specified via an argument to the Tessellate filter constructor.

```python
from paraview.simple import *
pvd = PVDReader(FileName="Example.pvd")
tes = Tessellate(pvd, FieldError=0.001)
```

1.9.4 Saving multiple functions

Often we will want to save, and subsequently visualise, multiple different fields from a simulation. For example the velocity and pressure in a fluids models. This is possible either by having a separate output file for each field, or by saving multiple fields to the same output file. The latter may be more convenient for subsequent analysis. To do this, we just need to pass multiple *Functions* to *write()*.

```python
u = Function(V, name="Velocity")
p = Function(P, name="Pressure")

outfile = File("output.pvd")
```

(continues on next page)

(continued from previous page)

```
outfile.write(u, p, time=0)

# We can happily do this in a timeloop as well.
while t < t:
    ...
    outfile.write(u, p, time=t)
```

Note: Subsequent writes to the same file *must* use the same number of functions, and the functions must have the *same* names. The following example results in an error.

```
u = Function(V, name="Velocity")
p = Function(P, name="Pressure")

outfile = File("output.pvd")

outfile.write(u, p, time=0)
...
# This raises an error
outfile.write(u, time=1)
# as does this
outfile.write(p, u, time=1)
```

Selecting the output space when outputting multiple functions

All functions, including the mesh coordinates, that are output to the same file must be represented in the same space, the rules for selecting the output space are as follows. First, all functions must be defined via the same cell type otherwise an exception will be thrown. Second, if all functions are continuous (i.e. they live in H^1), then the output space will be a piecewise continuous space. If any of the functions are at least partially discontinuous, again including the coordinate field (this occurs when using periodic meshes), then the output will use a piecewise discontinuous space. Third, the degree of the basis will be the maximum degree used over the spaces of all input functions. For elements where the degree is a tuple (this occurs when using tensor product elements), the the maximum will be over the elements of the tuple too, meaning a tensor product of elements of degree 4 and 2 will be turned into a tensor product of elements of degree 4 and 4.

1.9.5 Plotting with *matplotlib*

Firedrake includes support for plotting meshes and functions using matplotlib. The API for plotting mimics that of matplotlib as much as possible. For example the functions *tripcolor*, *tricontour*, and so forth, all behave more or less like their counterparts in matplotlib, and actually call them under the hood. The only difference is that the Firedrake functions include an extra optional argument `axes` to specify the matplotlib `Axes` object to draw on. When using matplotlib by itself these methods are methods of the Axes object. Otherwise the usage is identical. For example, the following code would make a filled contour plot of the function `u` using the inferno colormap, with contours drawn at 0.0, 0.02, . . . , 1.0, and add a colorbar to the figure.

```python
import matplotlib.pyplot as plt
import numpy as np
mesh = UnitSquareMesh(10, 10)
V = FunctionSpace(mesh, "CG", 1)
u = Function(V)
x = SpatialCoordinate(mesh)
u.interpolate(x[0] + x[1])
fig, axes = plt.subplots()
levels = np.linspace(0, 1, 51)
contours = tricontourf(u, levels=levels, axes=axes, cmap="inferno")
axes.set_aspect("equal")
fig.colorbar(contours)
fig.show()
```

For vector fields, triplot and tricontour will show the magnitude of function. To see the direction as well, you can instead call the *quiver* function, which again works the same as its counterpart in matplotlib.

The function *triplot* has one major departure from matplotlib to make finite element analysis easier. The different segments of the boundary are shown with different colors in order to make it easy to determine the numeric ID of each boundary segment. Mistaking which segments of the boundary should have Dirichlet or Neumann boundary conditions is a common source of errors in applications. To see a legend explaining the colors, you can add a legend like so:

```python
mesh = Mesh(mesh_filename)
import matplotlib.pyplot as plt
fig, axes = plt.subplots()
triplot(mesh, axes=axes)
axes.legend()
fig.show()
```

The numeric IDs shown in the legend are the same as those stored internally in the mesh, so for example if you added physical lines using gmsh the numbering is the same.

For 1D functions with degree less than 4, the plot of the function would be exact using Bezier curves. For higher order 1D functions, the plot would be the linear approximation by sampling points of the function. The number of sample points per element could be specfied to when calling *plot*.

To install matplotlib, please look at the installation instructions of matplotlib.

1.10 Checkpointing state

In addition to the ability to *write field data to vtu files*, suitable for visualisation in Paraview, Firedrake has support for checkpointing state to disk. This enables pausing, and subsequently resuming, a simulation at a later time.

1.10.1 Checkpointing with CheckpointFile

CheckpointFile class facilitates saving/loading meshes and *Function* s to/from an HDF5 file. The implementation is scalable in that *Function* s are saved to and loaded from the file entirely in parallel without needing to pass through a single process. It also supports flexible checkpointing, where one can save meshes and *Function* s on N processes and later load them on P processes. If $P == N$, the parallel distribution and entity permutation (reordering) of the saved mesh is recovered on the loaded mesh by default. The distribution and permutation data are by default stored under names automatically generated by Firedrake.

Warning: If the mesh has a non-standard distribution, e.g., generated by a partitioner with some non-standard parameters, it is recommended that the user set the distribution name explicitly when constructing a mesh; see, e.g., *Mesh()*.

Warning: If $P! = N$ or *P* == *N* but *distribution_parameters* dict and/or *reorder* parameter are passed when loading a mesh, the saved mesh and the loaded mesh (and thus the saved function and the loaded function) will in general be represented by different global numbering systems and they are merely guaranteed to be analytically the same; as a consequence, it is currently not allowed to save the once loaded mesh or function back to the same file under the same name as this would cause conflict with other data stored using incompatible global numbering system. We plan to remove this restriction in the future.

Saving

In the following example we save in “example.h5” file two *Function* s, along with the mesh on which they are defined.

```
mesh = UnitSquareMesh(10, 10, name="meshA")
V = FunctionSpace(mesh, "CG", 2)
W = FunctionSpace(mesh, "CG", 1)
Z = V * W
f = Function(V, name="f")
g = Function(Z, name="g")
with CheckpointFile("example.h5", 'w') as afile:
    afile.save_mesh(mesh)  # optional
    afile.save_function(f)
    afile.save_function(g)
```

If the mesh name is not provided by the user when constructing the mesh, the default mesh name, *DEFAULT_MESH_NAME*, is assigned, which is then used when saving in the file. We, however, strongly encourage users to name each mesh.

Inspecting saved data

Now “example.h5” file has been created and the mesh and *Function* data have been saved. One can view the contents of the HDF5 file with “h5dump” utility shipped with the HDF5 installation; “h5dump -n example.h5”, for instance, shows:

```
HDF5 "example.h5" {
FILE_CONTENTS {
 group      /
 group      /topologies
 group      /topologies/firedrake_mixed_meshes
 group      /topologies/firedrake_mixed_meshes/meshA
 group      /topologies/firedrake_mixed_meshes/meshA/firedrake_mixed_function_
↪spaces
 group      /topologies/firedrake_mixed_meshes/meshA/firedrake_mixed_function_
↪spaces/firedrake_function_space_meshA_CG2(None,None)_meshA_CG1(None,None)
 group      /topologies/firedrake_mixed_meshes/meshA/firedrake_mixed_function_
↪spaces/firedrake_function_space_meshA_CG2(None,None)_meshA_CG1(None,None)/0
 group      /topologies/firedrake_mixed_meshes/meshA/firedrake_mixed_function_
↪spaces/firedrake_function_space_meshA_CG2(None,None)_meshA_CG1(None,None)/1
 group      /topologies/firedrake_mixed_meshes/meshA/firedrake_mixed_function_
↪spaces/firedrake_function_space_meshA_CG2(None,None)_meshA_CG1(None,None)/
↪firedrake_functions
 group      /topologies/firedrake_mixed_meshes/meshA/firedrake_mixed_function_
↪spaces/firedrake_function_space_meshA_CG2(None,None)_meshA_CG1(None,None)/
↪firedrake_functions/g
 group      /topologies/firedrake_mixed_meshes/meshA/firedrake_mixed_function_
↪spaces/firedrake_function_space_meshA_CG2(None,None)_meshA_CG1(None,None)/
↪firedrake_functions/g/0
 group      /topologies/firedrake_mixed_meshes/meshA/firedrake_mixed_function_
↪spaces/firedrake_function_space_meshA_CG2(None,None)_meshA_CG1(None,None)/
↪firedrake_functions/g/1
 group      /topologies/meshA_topology
 group      /topologies/meshA_topology/distributions
 group      /topologies/meshA_topology/distributions/firedrake_default_1_True_
↪None_(FACET,1)
 dataset    /topologies/meshA_topology/distributions/firedrake_default_1_True_
↪None_(FACET,1)/chart_sizes
 dataset    /topologies/meshA_topology/distributions/firedrake_default_1_True_
↪None_(FACET,1)/global_point_numbers
 dataset    /topologies/meshA_topology/distributions/firedrake_default_1_True_
↪None_(FACET,1)/owners
 group      /topologies/meshA_topology/distributions/firedrake_default_1_True_
↪None_(FACET,1)/permutations
 group      /topologies/meshA_topology/distributions/firedrake_default_1_True_
↪None_(FACET,1)/permutations/firedrake_default_True
 dataset    /topologies/meshA_topology/distributions/firedrake_default_1_True_
```

(continues on next page)

(continued from previous page)

```
↪None_(FACET,1)/permutations/firedrake_default_True/permutation
 group      /topologies/meshA_topology/dms
 group      /topologies/meshA_topology/dms/coordinateDM
 dataset    /topologies/meshA_topology/dms/coordinateDM/order
 group      /topologies/meshA_topology/dms/coordinateDM/section
 dataset    /topologies/meshA_topology/dms/coordinateDM/section/atlasDof
 dataset    /topologies/meshA_topology/dms/coordinateDM/section/atlasOff
 group      /topologies/meshA_topology/dms/coordinateDM/section/field0
 dataset    /topologies/meshA_topology/dms/coordinateDM/section/field0/
↪atlasDof
 dataset    /topologies/meshA_topology/dms/coordinateDM/section/field0/
↪atlasOff
 group      /topologies/meshA_topology/dms/coordinateDM/section/field0/
↪component0
 group      /topologies/meshA_topology/dms/coordinateDM/section/field0/
↪component1
 group      /topologies/meshA_topology/dms/coordinateDM/vecs
 group      /topologies/meshA_topology/dms/coordinateDM/vecs/coordinates
 dataset    /topologies/meshA_topology/dms/coordinateDM/vecs/coordinates/
↪coordinates
 group      /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_1
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_1/order
 group      /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_1/section
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_1/section/
↪atlasDof
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_1/section/
↪atlasOff
 group      /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_1/vecs
 group      /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_1/vecs/
↪g[1]
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_1/vecs/
↪g[1]/g[1]
 group      /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_2
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_2/order
 group      /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_2/section
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_2/section/
↪atlasDof
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_2/section/
↪atlasOff
 group      /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_2/vecs
 group      /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_2/vecs/
↪meshA_coordinates
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_0_0_False_2/vecs/
↪meshA_coordinates/meshA_coordinates
 group      /topologies/meshA_topology/dms/firedrake_dm_1_1_0_False_1
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_1_0_False_1/order
 group      /topologies/meshA_topology/dms/firedrake_dm_1_1_0_False_1/section
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_1_0_False_1/section/
↪atlasDof
```

(continues on next page)

(continued from previous page)

```
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_1_0_False_1/section/
↪atlasOff
 group      /topologies/meshA_topology/dms/firedrake_dm_1_1_0_False_1/vecs
 group      /topologies/meshA_topology/dms/firedrake_dm_1_1_0_False_1/vecs/f
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_1_0_False_1/vecs/f/f
 group      /topologies/meshA_topology/dms/firedrake_dm_1_1_0_False_1/vecs/
↪g[0]
 dataset    /topologies/meshA_topology/dms/firedrake_dm_1_1_0_False_1/vecs/
↪g[0]/g[0]
 group      /topologies/meshA_topology/firedrake_meshes
 group      /topologies/meshA_topology/firedrake_meshes/meshA
 group      /topologies/meshA_topology/firedrake_meshes/meshA/firedrake_
↪function_spaces
 group      /topologies/meshA_topology/firedrake_meshes/meshA/firedrake_
↪function_spaces/firedrake_function_space_meshA_CG1(None,None)
 group      /topologies/meshA_topology/firedrake_meshes/meshA/firedrake_
↪function_spaces/firedrake_function_space_meshA_CG1(None,None)/firedrake_
↪functions
 group      /topologies/meshA_topology/firedrake_meshes/meshA/firedrake_
↪function_spaces/firedrake_function_space_meshA_CG1(None,None)/firedrake_
↪functions/g[1]
 group      /topologies/meshA_topology/firedrake_meshes/meshA/firedrake_
↪function_spaces/firedrake_function_space_meshA_CG2(None,None)
 group      /topologies/meshA_topology/firedrake_meshes/meshA/firedrake_
↪function_spaces/firedrake_function_space_meshA_CG2(None,None)/firedrake_
↪functions
 group      /topologies/meshA_topology/firedrake_meshes/meshA/firedrake_
↪function_spaces/firedrake_function_space_meshA_CG2(None,None)/firedrake_
↪functions/f
 group      /topologies/meshA_topology/firedrake_meshes/meshA/firedrake_
↪function_spaces/firedrake_function_space_meshA_CG2(None,None)/firedrake_
↪functions/g[0]
 group      /topologies/meshA_topology/labels
 group      /topologies/meshA_topology/labels/...
 ...
 group      /topologies/meshA_topology/topology
 dataset    /topologies/meshA_topology/topology/cells
 dataset    /topologies/meshA_topology/topology/cones
 dataset    /topologies/meshA_topology/topology/order
 dataset    /topologies/meshA_topology/topology/orientation
 }
}
```

Loading

We can load the mesh and *Function* s in “example.h5” as in the following.

```
with CheckpointFile("example.h5", 'r') as afile:
    mesh = afile.load_mesh("meshA")
    f = afile.load_function(mesh, "f")
    g = afile.load_function(mesh, "g")
```

Note that one needs to load the mesh before loading the *Function* s that are defined on it. If the default mesh name, *DEFAULT_MESH_NAME*, was used when saving, the mesh name can be ommitted when loading.

Extrusion

Extruded meshes can be saved and loaded seamlessly as the following:

```
mesh = UnitSquareMesh(10, 10, name="meshA")
extm = ExtrudedMesh(mesh, layers=4)
V = FunctionSpace(extm, "CG", 2)
f = Function(V, name="f")
with CheckpointFile("example_extrusion.h5", 'w') as afile:
    afile.save_mesh(mesh)  # optional
    afile.save_function(f)
with CheckpointFile("example_extrusion.h5", 'r') as afile:
    extm = afile.load_mesh("meshA_extruded")
    f = afile.load_function(extm, "f")
```

Note that if the name was not directly provided by the user, the base mesh’s name postfixed by “_extruded” is given to the extruded mesh.

Timestepping

The following demonstrates how a *Function* can be saved and loaded at each timestep in a time-series simulation by setting the *idx* parameter:

```
mesh = UnitSquareMesh(2, 2, name="meshA")
V = FunctionSpace(mesh, "CG", 1)
f = Function(V, name="f")
x, y = SpatialCoordinate(mesh)
with CheckpointFile("example_timestepping.h5", 'w') as afile:
    afile.save_mesh(mesh)  # optional
    for i in range(4):
        f.interpolate(x * i)
        afile.save_function(f, idx=i)
with CheckpointFile("example_timestepping.h5", 'r') as afile:
    mesh = afile.load_mesh("meshA")
    for i in range(4):
        f = afile.load_function(mesh, "f", idx=i)
```

Note that each `Function` can either be saved in the timestepping mode with *idx* parameter always set or in the normal mode (non-timestepping mode) with *idx* parameter always unset, and the same `Function` can only be loaded using the same mode.

1.10.2 Using disk checkpointing in adjoint simulations

When adjoint annotation is active, the result of every Firedrake operation is stored in memory. For some simulations, this can result in a very large memory footprint. As an alternative, it is possible to specify that those intermediate results in forward evaluations of the tape which have type `Function` be written to disk. This is usually the bulk of the data stored on the tape so this largely alleviates the memory problem, at the cost of the time taken to read to and write from disk.

Having imported *firedrake_adjoint*, there are two steps required to enable disk checkpointing of the forward tape state.

1. Call `enable_disk_checkpointing()`.
2. Wrap all mesh constructors in `checkpointable_mesh()`.

See the documentation of those functions for more detail.

1.10.3 Checkpointing with DumbCheckpoint

Warning: `DumbCheckpoint` will be deprecated after 01/01/2023. Instead, users are encouraged to use `CheckpointFile`, which is more robust and scalable.

The support for `DumbCheckpoint` is somewhat limited. One may only store `Function`s in the checkpoint object. Moreover, no remapping of data is performed. This means that resuming the checkpoint is only possible on the same number of processes as used to create the checkpoint file. Additionally, the *same* `Mesh` must be used: that is a `Mesh` constructed identically to the mesh used to generate the saved checkpoint state.

Opening a checkpoint

A checkpoint file is created using the `DumbCheckpoint` constructor. We pass a filename argument, and an access mode. Available modes are:

FILE_READ

Open the checkpoint file for reading. Raises `OSError` if the file does not already exist.

FILE_CREATE

Open the checkpoint file for reading and writing, creating the file if it does not exist, and *erasing* any existing contents if it does.

FILE_UPDATE

Open the checkpoint file for reading and writing, creating it if it does not exist, without erasing any existing contents.

For example, to open a checkpoint file for writing solution state, truncating any existing contents we use:

```
chk = DumbCheckpoint("dump", mode=FILE_CREATE)
```

note how we only provide the base name of the on-disk file, `".h5"` is appended automatically.

Storing data

Once a checkpoint file is opened, *`Function`* data can be stored in the checkpoint using *`store()`*. A *`Function`* is referenced in the checkpoint file by its `name()`, but this may be overridden by explicitly passing an optional *name* argument. For example, to store a *`Function`* using its default name use:

```
f = Function(V, name="foo")
chk.store(f)
```

If instead we want to override the name we use:

```
chk.store(f, name="bar")
```

Warning: No warning is provided when storing multiple *`Function`*s with the same name, existing values are overwritten.

Moreover, attempting to store a *`Function`* with a different number of degrees of freedom into an existing name will cause an error.

Loading data

Once a checkpoint is created, we can use it to load saved state into *`Function`*s to resume a simulation. To load data into a *`Function`* from a checkpoint, we pass it to *`load()`*. As before, the data is looked up by its `name()`, although once again this may be overridden by optionally specifying the `name` as an argument.

For example, assume we had previously saved a checkpoint containing two different *`Function`*s with names `"A"` and `"B"`. We can load these as follows:

```
chk = DumbCheckpoint("dump.h5", mode=FILE_READ)

a = Function(V, name="A")

b = Function(V)

# Use a.name() to look up value
chk.load(a)

# Look up value by explicitly specifying name="B"
chk.load(b, name="B")
```

Note: Since Firedrake does not currently support reading data from a checkpoint file on a different number of processes from that it was written with, whenever a *Function* is stored, an attribute is set recording the number of processes used. When loading data from the checkpoint, this value is validated against the current number of processes and an error is raised if they do not match.

Closing a checkpoint

The on-disk file inside a checkpoint object is automatically closed when the checkpoint object is garbage-collected. However, since this may not happen at a predictable time, it is possible to manually close a checkpoint file using *close()*. To facilitate this latter usage, checkpoint objects can be used as context managers which ensure that the checkpoint file is closed as soon as the object goes out of scope. To use this approach, we use the python `with` statement:

```
# Normal code here
with DumbCheckpoint("dump.h5", mode=FILE_UPDATE) as chk:
    # Checkpoint file open for reading and writing
    chk.store(...)
    chk.load(...)

# Checkpoint file closed, continue with normal code
```

Writing attributes

In addition to storing *Function* data, it is also possible to store metadata in *DumbCheckpoint* files using HDF5 attributes. This is carried out using h5py to manipulate the file. The interface allows setting attribute values, reading them, and checking if a file has a particular attribute:

write_attribute()

Write an attribute, specifying the object path the attribute should be set on, the name of the attribute and its value.

read_attribute()

Read an attribute with specified name from at a given object path.

has_attribute()

Check if a particular attribute exists. Does not raise an error if the object also does not exist.

Support for multiple timesteps

The checkpoint object supports multiple timesteps in the same on-disk file. The primary interface to this is via *set_timestep()*. If never called on a checkpoint file, no timestep support is enabled, and storing a *Function* with the same name as an existing object overwrites it (data is stored in the HDF5 group `"/fields"`). If one wishes to store multiple timesteps, one should call *set_timestep()*, providing the timestep value (and optionally a timestep "index"). Storing a *Function* will now write to the group `"/fields/IDX"`. To store the same function at a different time level, we just call *set_timestep()* again with a new timestep value.

Inspecting available time levels

The stored time levels in the checkpoint object are available as attributes in the file. They may be inspected by calling *get_timesteps()*. This returns a list of the timesteps stored in the file, along with the indices they map to. In addition, the timestep value is available as an attribute on the appropriate field group: reading the attribute `"/fields/IDX/timestep"` returns the timestep value corresponding to `IDX`.

Support for multiple on-disk files

For large simulations, it may not be expedient to store all timesteps in the same on-disk file. To this end, the *DumbCheckpoint* object offers the facility to retain the same checkpoint object, but change the on-disk file used to store the data. To switch to a new on-disk file one uses *new_file()*. There are two method of choosing the new file name. If the *DumbCheckpoint* object was created passing `single_file=False` then calling *new_file()* without any additional arguments will use an internal counter to create file names by appending this counter to the provided base name. This selection can be overridden by explicitly passing the optional `name` argument.

As an example, consider the following sequence:

```
with DumbCheckpoint("dump", single_file=False, mode=FILE_CREATE) as chk:
    chk.store(a)
    chk.store(b)
    chk.new_file()
    chk.store(c)
    chk.new_file(name="special")
    chk.store(d)
    chk.new_file()
    chk.store(e)
```

Will create four on-disk files:

`dump_0.h5`

  Containing `a` and `b`;

`dump_1.h5`

  Containing `c`;

`special.h5`

  Containing `d`;

`dump_2.h5`

Containing `e`.

1.10.4 Implementation details

The on-disk representation of checkpoints is as HDF5 files. Firedrake uses the PETSc HDF5 Viewer object to write and read state. As such, writing data is collective across processes. h5py is used for attribute manipulation. To this end, h5py *must* be linked against the same version of the HDF5 library that PETSc was built with. The `firedrake-install` script automates this, however, if you build PETSc manually, you will need to ensure that h5py is linked correctly following the instructions for custom installation here.

Warning: Calling `h5py:File.close()` on the h5py representation will likely result in errors inside PETSc (since it is not aware that the file has been closed). So don't do that!

1.11 Matrix-free operators

In addition to supporting computation with the workhorse of sparse linear algebra, an assembled sparse matrix, Firedrake also supports computing "matrix-free". In this case, the matrix returned from *`assemble()`* implements matrix-vector multiplication by the assembly of a 1-form subject to boundary conditions rather than direct construction of a sparse matrix ("aij" format) followed by traditional CSR algorithms. This functionality is documented in more detail in [KM18].

There are two ways of accessing this functionality. One can either request a matrix-free operator by passing `mat_type="matfree"` to *`assemble()`*. In this case, the returned object is an *`ImplicitMatrix`*. This object can be used in the normal way with a *`LinearSolver`*. Alternately, when solving a variational problem, an *`ImplicitMatrix`* is requested through the `solver_parameters` dict, by setting the option `mat_type` to `matfree`. The type of the preconditioning matrix can be controlled separately by setting `pmat_type`.

Generically, one can expect such a matrix to be cheaper to "assemble" and to use less memory, especially for high-order discretizations or complicated systems. The downside is that traditional algebraic preconditioners will not work with such unassembled matrices. To take advantage of these features, we need to configure our solvers correctly. To expedite this, the matrix-free operator, implemented as a PETSc shell matrix, contains an application context of type *`ImplicitMatrixContext`*. This context provides some important features to enabled advanced solver configuration.

1.11.1 Splitting unassembled matrices

For the purposes of fieldsplit preconditioners, the PETSc matrix object must be able to extract submatrices. For unassembled matrices, this is achieved through a specialized `ImplicitMatrixContext.getSubMatrix()` method that partitions the UFL form defining the operator into pieces corresponding to the integer labels of the unknown fields. This is in contrast to the normal splitting of assembled matrices which operates at a purely algebraic level. With unassembled operators, the PDE description is available in the matrix, and is therefore propagated down to the split operators.

1.11.2 Preconditioning unassembled matrices

As well as providing symbolic field splitting, the *ImplicitMatrixContext* object is available to preconditioners. Since it contains a complete UFL description of the bilinear form, preconditioners can query or manipulate it as desired. As a particularly simple example, the class *AssembledPC* simply passes the UFL into *assemble()* to produce an explicit matrix during set up. It also sets up a new PETSc PC context acting on this assembled matrix so that the user can configure it at run-time via the options database. This allows the use of matrix-free actions in the Krylov solve, preconditioned using an assembled matrix.

Providing application context to preconditioners

Frequently, such custom preconditioners require some additional information that will not be fully available from the UFL description in *ImplicitMatrixContext*. For example, it is not possible to extract physical parameters such as the Reynolds number from a UFL bilinear form. In this case, the solver accepts a dictionary "appctx" as an optional keyword argument, the same argument may also be passed to *assemble()* in the case of preassembled solves. Firedrake passes that down into the *ImplicitMatrixContext* so that it is accessible to preconditioners.

1.11.3 Example usage

To demonstrate basic usage of matrix-free operators and preconditioners, we show a simple Poisson problem, introducing some of the additional solver options.

1.11.4 Poisson equation

It is what it is, a conforming discretization on a regular mesh using piecewise quadratic elements.

As usual we start by importing firedrake and setting up the problem.:

```
from firedrake import *

N = 128

mesh = UnitSquareMesh(N, N)

V = FunctionSpace(mesh, "CG", 2)

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

a = inner(grad(u), grad(v)) * dx

x = SpatialCoordinate(mesh)
F = Function(V)
F.interpolate(sin(x[0]*pi)*sin(2*x[1]*pi))
L = F*v*dx
```

(continues on next page)

(continued from previous page)

```
bcs = [DirichletBC(V, Constant(2.0), (1,))]

uu = Function(V)
```

With the setup out of the way, we now demonstrate various ways of configuring the solver. First, a direct solve with an assembled operator.:

```
solve(a == L, uu, bcs=bcs, solver_parameters={"ksp_type": "preonly",
                                              "pc_type": "lu"})
```

Next, we use unpreconditioned conjugate gradients using matrix-free actions. This is not very efficient due to the h^{-2} conditioning of the Laplacian, but demonstrates how to request an unassembled operator using the "mat_type" solver parameter.:

```
uu.assign(0)
solve(a == L, uu, bcs=bcs, solver_parameters={"mat_type": "matfree",
                                              "ksp_type": "cg",
                                              "pc_type": "none",
                                              "ksp_monitor": None})
```

Finally, we demonstrate the use of a *AssembledPC* preconditioner. This uses matrix-free actions but preconditions the Krylov iterations with an incomplete LU factorisation of the assembled operator.:

```
uu.assign(0)
solve(a == L, uu, bcs=bcs, solver_parameters={"mat_type": "matfree",
                                              "ksp_type": "cg",
                                              "ksp_monitor": None,
```

To use the assembled matrix for the preconditioner we select a "python" type:

```
"pc_type": "python",
```

and set its type, by providing the name of the class constructor to PETSc.:

```
"pc_python_type": "firedrake.AssembledPC",
```

Finally, we set the preconditioner type for the assembled operator:

```
"assembled_pc_type": "ilu"})
```

This demo is available as a runnable python file here.

1.12 Preconditioning infrastructure

Firedrake has tight coupling with the PETSc library which provides support for a wide range of preconditioning strategies, see the relevant PETSc documentation for an overview.

In addition to these algebraic approaches, Firedrake offers a flexible framework for defining preconditioners that need to construct or apply auxiliary operators. The basic approach is described in [KM18]. Here we provide a brief overview of the preconditioners available in Firedrake that use this approach. To use these preconditioners, one sets `"pc_type": "python"` and `"pc_python_type": "fully_qualified.NameOfPC"` in the `solver_parameters`.

1.12.1 Additive Schwarz methods

Small-block overlapping additive Schwarz preconditioners built on top of PCASM that can be used as components of robust multigrid schemes when using geometric multigrid.

ASMPatchPC
: Abstract base class for which one must implement *ASMPatchPC.get_patches()* to extract sets of degrees of freedom. Needs to be used with assembled sparse matrices (`"mat_type": "aij"`).

ASMStarPC
: Constructs patches by gathering degrees of freedom in the star of specified mesh entities.

ASMVankaPC
: Constructs patches using the Vanka scheme by gathering degrees of freedom in the closure of the star of specified mesh entities.

ASMLinesmoothPC
: Constructs patches gathering degrees of freedom in vertical columns on *extruded meshes*.

ASMExtrudedStarPC
: Like *ASMStarPC* but on extruded meshes.

In addition to these algebraic approaches to constructing patches, Firedrake also interfaces with PCPATCH for both linear and nonlinear overlapping Schwarz methods. The approach is described in detail in [FKMW21]. These preconditioners can be used with both sparse matrices and Firedrake's *matrix-free operators*, and can be applied either additively or multiplicatively within an MPI rank and additively between ranks.

PatchPC
: Small-block overlapping Schwarz smoother with topological definition of patches. Does not support extruded meshes.

PatchSNES
: Nonlinear overlapping Schwarz smoother with topological definition of patches. Does not support extruded meshes.

PlaneSmoother
: A Python construction class for *PatchPC* and *PatchSNES* that approximately groups mesh entities into lines or planes (useful for advection-dominated problems).

1.12.2 Multigrid methods

Firedrake has support for rediscretised geometric multigrid on both normal and extruded meshes, with regular refinement. This is obtained by constructing a *mesh hierarchy* and then using `"pc_type": "mg"`. In addition to this basic support, it also has out of the box support for a number of problem-specific preconditioners.

HypreADS
: An interface to Hypre's auxiliary space divergence solver. Currently only implemented for lowest-order Raviart-Thomas elements.

HypreAMS
: An interface to Hypre's auxiliary space Maxwell solver. Currently only implemented for lowest order Nedelec elements of the first kind.

PMGPC
: Generic p-coarsening rediscretised linear multigrid. If the problem is built on a mesh hierarchy then the coarse grid can do further h-multigrid with geometric coarsening.

P1PC
: Coarsening directly to linear elements.

PMGSNES
: Generic p-coarsening nonlinear multigrid. If the problem is built on a mesh hierarchy then the coarse grid can do further h-multigrid with geometric coarsening.

P1SNES
: Coarsening directly to linear elements.

GTMGPC
: A two-level non-nested multigrid scheme for the hybridised mixed method that operates on the trace variable, using the approach of [GT09]. The Firedrake implementation is described in [BGGMuller21].

1.12.3 Assembled and auxiliary operators

Many preconditioning schemes call for auxiliary operators, these are facilitated by variations on Firedrake's *AssembledPC* which can be used to deliver an assembled operator inside a nested solver where the outer matrix is a matrix-free operator. Matrix-free operators can be used "natively" with PETSc's `"jacobi"` preconditioner, since they can provide their diagonal cheaply. For more complicated things, one must assemble an operator instead.

AssembledPC
: Assemble an operator as a sparse matrix and then apply an inner preconditioner. For example, this might be used to assemble a coarse grid in an (otherwise matrix-free) multigrid solver.

AuxiliaryOperatorPC
: Abstract base class for preconditioners built from assembled auxiliary operators. One should subclass this preconditioner and implement the *AuxiliaryOperatorPC.form()* method. This can be used to provided bilinear forms to the solver that were not there in the original problem, for example, the pressure mass matrix for block preconditioners of the Stokes equations.

FDMPC
: An auxiliary operator that uses piecewise-constant coefficients that is assembled in the

basis of shape functions that diagonalize separable problems in the interior of each cell. Currently implemented for quadrilateral and hexahedral cells. The assembled matrix becomes as sparse as a low-order refined preconditioner, to which one may apply other preconditioners such as *ASMStarPC* or *ASMExtrudedStarPC*. See details in [BF21].

MassInvPC
Preconditioner for applying an inverse mass matrix.

PCDPC
A preconditioner providing the Pressure-Convection-Diffusion approximation to the Schur complement for the Navier-Stokes equations. Note that this implementation only treats problems with characteristic velocity boundary conditions correctly.

1.12.4 Hybridisation and element-wise static condensation

Firedrake has a number of preconditioners that use the *Slate* facility for element-wise linear algebra on assembled tensors. These are described in detail in [GMHC20].

HybridizationPC
A preconditioner for hybridisable H(div) mixed methods that breaks the vector-valued space, and enforces continuity through introduction of a trace variable. The (now-broken) problem is eliminated element-wise onto the trace space to leave a single-variable global problem, whose solver can be configured.

SCPC
A preconditioner that performs element-wise static condensation onto a single field.

1.13 Interfacing directly with PETSc

1.13.1 Introduction

Sometimes, the system we wish to solve can not be described purely in terms of a sum of weak forms that we can then assemble. Or else, it might be, but the resulting assembled operator would be dense. In this chapter, we will see how to solve such problems in a “matrix-free” manner, using Firedrake to assemble the pieces and then providing a matrix object to PETSc which is unassembled. Note that this is a lower-level interface than that described in *Matrix-free operators*, so you should try that first to see if it suits your needs.

To take a concrete example, let us consider a linear system obtained from a normal variational problem, augmented with a rank-1 perturbation:

$$B := A + \vec{u}\vec{v}^T.$$

Such operators appear, for example, in limited memory quasi-Newton methods such as L-BFGS or Broyden.

The matrix B is dense, however its action on a vector may be computed in only marginally more work than computing the action of A since

$$B\vec{x} \equiv A\vec{x} + \vec{u}(\vec{v} \cdot \vec{x}).$$

1.13.2 Accessing PETSc objects

Firedrake builds on top of PETSc for its linear algebra, and therefore all assembled forms provide access to the underlying PETSc object. For assembled bilinear forms, the PETSc object is a `Mat`; for assembled linear forms, it is a `Vec`. The ways we access these are different. For a bilinear form, the matrix is obtained with:

```
petsc_mat = assemble(bilinear_form).M.handle
```

For a linear form, we need to use a context manager. There are two options available here, depending on whether we want read-only or read-write access to the PETSc object. For read-only access, we use:

```
with assemble(linear_form).dat.vec_ro as v:
    petsc_vec_ro = v
```

For write-only access, use `.vec_wo`, and for read-write access, use:

```
with assemble(linear_form).dat.vec as v:
    petsc_vec = v
```

These context managers ensure that if PETSc writes to the vector, Firedrake sees the modification of the values.

Plotting the sparsity of a PETSc `Mat`

Given a PETSc matrix of type `'seqaij'`, we may access its compressed sparse row format and convert to that used in SciPy in the following way:

```
import scipy.sparse as sp

indptr, indices, data = petsc_mat.getValuesCSR()
scipy_mat = sp.csr_matrix((data, indices, indptr), shape=petsc_mat.getSize())
```

The sparsity pattern may then be straightforwardly plotted using matplotlib:

```
import matplotlib.pyplot as plt

plt.spy(scipy_mat)
```

1.13.3 Building an operator

To solve the linear system $Bx = b$ we need to define the operator B such that PETSc can use it. To do this, we build a Python class that provides a `mult` method:

```
class MatrixFreeB(object):

    def __init__(self, A, u, v):
        self.A = A
        self.u = u
```

(continues on next page)

(continued from previous page)

```
        self.v = v

    def mult(self, mat, x, y):
        # y <- A x
        self.A.mult(x, y)

        # alpha <- v^T x
        alpha = self.v.dot(x)

        # y <- y + alpha*u
        y.axpy(alpha, self.u)
```

Now we must build a PETSc `Mat` and indicate that it should use this newly defined class to compute the matrix action:

```
# Import petsc4py namespace
from firedrake.petsc import PETSc

B = PETSc.Mat().create()

# Assemble the bilinear form that defines A and get the concrete
# PETSc matrix
A = assemble(bilinear_form).M.handle

# Now do the same for the linear forms for u and v, making a copy

with assemble(u_form).dat.vec_ro as u_vec:
    u = u_vec.copy()

with assemble(v_form).dat.vec_ro as v_vec:
    v = v_vec.copy()


# Build the matrix "context"
Bctx = MatrixFreeB(A, u, v)

# Set up B
# B is the same size as A
B.setSizes(A.getSizes())

B.setType(B.Type.PYTHON)
B.setPythonContext(Bctx)
B.setUp()
```

The next step is to build a linear solver object to solve the system. For this we need a PETSc `KSP`:

```
ksp = PETSc.KSP().create()
```

(continues on next page)

(continued from previous page)

```
ksp.setOperators(B)

ksp.setFromOptions()
```

Now we can solve a system using this `ksp` object:

```
solution = Function(V)

rhs = assemble(rhs_form)

with rhs.dat.vec_ro as b:
    with solution.dat.vec as x:
        ksp.solve(b, x)
```

1.13.4 Defining a preconditioner

Note: In many cases it is not necessary to drop to this low a level to construct problem-specific preconditioners. More details on this approach are discussed in the manual section on *Preconditioning infrastructure*.

Since PETSc only knows how to compute the action of B, and does not have access to any of the entries, it will not be able to build a preconditioner for the linear solver. To use a preconditioner, we have to provide PETSc with one. We can do this in one of two ways.

1. Provide an assembled matrix to the `KSP` object to be used as a preconditioning matrix. For example, we might use the matrix A. In this case, we merely have to call `ksp.setOperators` with two arguments:

   ```
   ksp.setOperators(B, A)
   ```

   Now we solve the system $Bx = b$, using A to build a preconditioner.

2. Provide our own `PC` object to be used as the preconditioner. This is somewhat more involved. As we did to define the matrix-free action of B, we need to build an object that applies the action of our chosen preconditioner. If we know that our matrix B has some special structure, this can be more efficient than the previous method.

Providing a custom preconditioner

Recall that we do not explicitly form B since it is dense, and subsequently its inverse is as well. However, since we know that B is formed of a full-rank invertible matrix, A, plus a rank-1 update, it is possible to compute its inverse reasonably cheaply using the Sherman-Morrison formula. Let A be invertible and u and v be column vectors such that $1 + v^T A^{-1} u \neq 0$ then:

$$B^{-1} = (A + uv^T)^{-1} = A^{-1} - \frac{A^{-1}uv^TA^{-1}}{1 + v^TA^{-1}u}.$$

Hence, we see that we can apply the action of B^{-1} on a vector using only the action of A^{-1} and some dot products.

With these mathematical preliminaries out of the way, let us move on to the implementation. We need to define an object which has an `apply` method which applies the action of our preconditioner to a vector. The PETSc `PC` object will be created with access to the operators we have provided to our solver, so for this class, we won't pass A, u and v explicitly, but rather extract them from the operators in a `setUp` method:

```
class MatrixFreePC(object):

    def setUp(self, pc):
        B, P = pc.getOperators()
        # extract the MatrixFreeB object from B
        ctx = B.getPythonContext()
        self.A = ctx.A
        self.u = ctx.u
        self.v = ctx.v
        # Here we build the PC object that uses the concrete,
        # assembled matrix A.  We will use this to apply the action
        # of A^{-1}
        self.pc = PETSc.PC().create()
        self.pc.setOptionsPrefix("mf_")
        self.pc.setOperators(self.A)
        self.pc.setFromOptions()
        # Since u and v do not change, we can build the denominator
        # and the action of A^{-1} on u only once, in the setup
        # phase.
        tmp = self.A.createVecLeft()
        self.pc.apply(self.u, tmp)
        self._Ainvu = tmp
        self._denom = 1 + self.v.dot(self._Ainvu)

    def apply(self, pc, x, y):
        # y <- A^{-1}x
        self.pc.apply(x, y)
        # alpha <- (v^T A^{-1} x) / (1 + v^T A^{-1} u)
        alpha = self.v.dot(y) / self._denom
        # y <- y - alpha * A^{-1}u
        y.axpy(-alpha, self._Ainvu)
```

Now we extract the `PC` object from the `KSP` linear solver and indicate that it should use our matrix free preconditioner

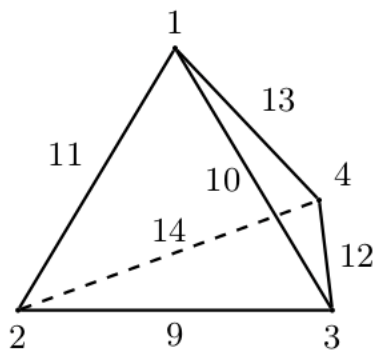
```
ksp = PETSc.KSP().create()
ksp.setOperators(B)
ksp.setUp()
pc = ksp.pc
pc.setType(pc.Type.PYTHON)
pc.setPythonContext(MFPC())
ksp.setFromOptions()
```

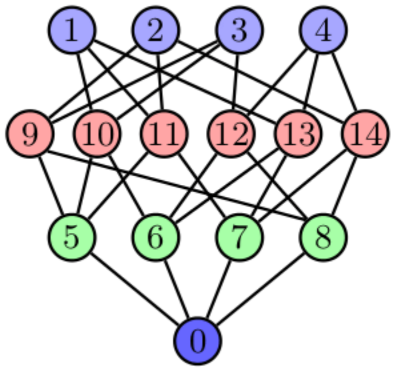
before going on to solve the system as before:

```
solution = Function(V)

rhs = assemble(rhs_form)

with rhs.dat.vec_ro as b:
    with solution.dat.vec as x:
        ksp.solve(b, x)
```

1.13.5 Accessing the PETSc mesh representation

Under the hood, Firedrake uses PETSc's DMPlex unstructured mesh representation. It uses a hierarchical approach, where entities of different dimension are put on different levels of the hierarchy. The single tetrahedral element shown on the left below may be interpreted using the graph representation on the right. Entities of dimension zero (vertices) are shown at the top. Entities of dimension one (edges) are shown on the next level down. Entities of dimension two (faces) are shown on the penultimate level and the (dimension three) element itself is on the bottom level. Edges in the graph indicate which entities own/are owned by others.





The DMPlex associated with a given `mesh` may be accessed via its `topology_dm` attribute:

```
plex = mesh.topology_dm
```

All entities in a DMPlex are given a unique number. The range of these numbers may be deduced using the method `plex.getDepthStratum`, whose only argument is the entity dimension sought. For example, 0 for vertices, 1 for edges, etc. Similarly, the method `plex.getHeightStratum` can be used for codimension access. For example, height 0 corresponds to cells. The hierarchical DMPlex structure may be traversed using other methods, such as `plex.getCone`, `plex.getSupport` and `plex.getTransitiveClosure`. See the Firedrake DMPlex paper and the PETSc manual for details.

If vertex coordinate information is to be accessed from the DMPlex then we must first establish a mapping between its numbering and the coordinates in the Firedrake mesh. This is done by establishing a 'section'. A section provides a way of associating data with the mesh - in this case, coordinate field data. For a \$d\$-dimensional mesh, we seek to establish offsets to recover \$d\$-tuple coordinates for the degrees of freedom.

For a linear mesh, we seek \$d\$ values at each vertex and no values for entities of higher dimension. In 2D, for example, this corresponds to the array

$$(d, 0, 0).$$

For an order p Lagrange mesh, it is a little more complicated. In the 2D triangular case, we require the following entities:

$$(d, d(p-1), d(p-1)(p-2)/2).$$

Accordingly, set

```
dim = mesh.topological_dimension()
gdim = mesh.geometrical_dimension()
entity_dofs = np.zeros(dim+1, dtype=np.int32)
entity_dofs[0] = gdim
entity_dofs[1] = gdim*(p-1)
entity_dofs[2] = gdim*((p-1)*(p-2))//2
```

We then use Firedrake's helper function for creating a PETSc section to establish the mapping:

```
from firedrake.cython.dmcommon import create_section

coord_section = create_section(mesh, entity_dofs)
plex = mesh.topology_dm
plex_coords = plex.getCoordinateDM()
plex_coords.setDefaultSection(coord_section)
coords_local = plex_coords.createLocalVec()
coords_local.array[:] = np.reshape(mesh.coordinates.dat.data_ro_with_halos,␣
↪coords_local.array.shape)
plex.setCoordinatesLocal(coords_local)
```

We can then extract coordinates for node `i` belonging to entity `d` (according to the DMPlex numbering) by

```
dofs = coord_section.getDof(d)
offset = coord_section.getOffset(d)//dim + i
coord = mesh.coordinates.dat.data_ro_with_halos[offset]
print(f"Node {i} belonging to entity {d} has coordinates {coord}")
```

1.14 Parallelism in Firedrake

Firedrake uses MPI for distributed memory parallelism. This is carried out transparently as long as your usage of Firedrake is only through the public API. To run your code in parallel you need you use the MPI job launcher available on your system. Often this program is called `mpiexec`. For example, to run a simulation in a file named `simulation.py` on 16 processes we might use.

```
mpiexec -n 16 python simulation.py
```

1.14.1 Installing for parallel use

By default, Firedrake makes use of an MPICH library that is downloaded, configured, and installed in the virtual environment as part of the PETSc installation procedure. If you do not intend to use parallelism, or only use it in a limited way, this will be sufficient for your needs. The default MPICH installation uses `nemesis` as the MPI channel, which is reasonably fast, but imposes a hard limit on the maximum number of concurrent MPI threads equal to the number of cores on your machine. If you would like to be able to *oversubscribe* your machine, and run more threads than cores, you need to change the MPICH device at install time to `sock`, by setting an environment variable before you run `firedrake-install`:

```
export PETSC_CONFIGURE_OPTIONS="--download-mpich-device=ch3:sock"
```

If parallel performance is important to you (e.g., for generating reliable timings or using a supercomputer), then you should probably be using an MPICH library tuned for your system. If you have a system-wide install already available, then you can simply tell the firedrake installer to use it, by running:

```
python3 firedrake-install --mpiexec=mpiexec --mpicc=mpicc --mpicxx=mpicxx --
↪mpif90=mpif90
```

where `mpiexec`, `mpicc`, `mpicxx`, and `mpif90` are the commands to run an MPI job and to compile C, C++, and Fortran 90 code, respectively.

1.14.2 Printing in parallel

The MPI execution model is that of single program, multiple data. As a result, printing output requires a little bit of care: just using `print()` will result in every process producing output. A sensible approach is to use PETSc's printing facilities to handle this, as *covered in this short demo*.

1.14.3 Expected performance improvements

Without detailed analysis, it is difficult to say precisely how much performance improvement should be expected from running in parallel. As a rule of thumb, it is worthwhile adding more processes as long as the number of degrees of freedom per process is more than around 50000. This is explored in some depth in the main Firedrake paper. Additionally, most of the finite element calculations performed by Firedrake are limited by the *memory bandwidth* of the machine. You can measure how the achieved memory bandwidth changes depending on the number of processes used on your machine using STREAMS.

1.14.4 Parallel garbage collection

As of the PETSc v3.18 release (which Firedrake started using October 2022), there should no longer be any issue with MPI distributed PETSc objects and Python's internal garbage collector. If you previously disabled the Python garbage collector in your Firedrake scripts, we now recommend you turn garbage collection back on. Randomly hanging or deadlocking parallel code should be debugged and any suspected issues reported by *getting in touch*.

1.14.5 Using MPI Communicators

By default, Firedrake parallelises across `MPI_COMM_WORLD`. If you want to perform a simulation in which different subsets of processes perform different computations (perhaps solving the same PDE for multiple different initial conditions), this can be achieved by using sub-communicators. The mechanism to do so is to provide a communicator when building the *`Mesh()`* you will perform the simulation on, using the optional `comm` keyword argument. All subsequent operations using that mesh are then only collective over the supplied communicator, rather than `MPI_COMM_WORLD`. For example, to split the global communicator into two and perform two different simulations on the two halves we would write.

```
from firedrake import *

comm = COMM_WORLD.Split(COMM_WORLD.rank % 2)

if COMM_WORLD.rank % 2 == 0:
   # Even ranks create a quad mesh
   mesh = UnitSquareMesh(N, N, quadrilateral=True, comm=comm)
else:
   # Odd ranks create a triangular mesh
   mesh = UnitSquareMesh(N, N, comm=comm)

...
```

To access the communicator a mesh was created on, we can use the `comm` property, or the function `mpi_comm()`.

Warning: Do not use the internal `_comm` attribute for communication. This communicator is for internal Firedrake MPI communication only.

1.14.6 Ensemble parallelism

Ensemble parallelism means solving simultaneous copies of a model with different coefficients, RHS or initial data, in situations that require communication between the copies. Use cases include ensemble data assimilation, uncertainty quantification, and time parallelism.

In ensemble parallelism, we split the MPI communicator into a number of subcommunicators, each of which we refer to as an ensemble member. Within each ensemble member, existing Firedrake functionality allows us to specify the FE problem solves which use spatial parallelism across the subcommunicator in the usual way. Another set of subcommunicators then allow communication between ensemble members.

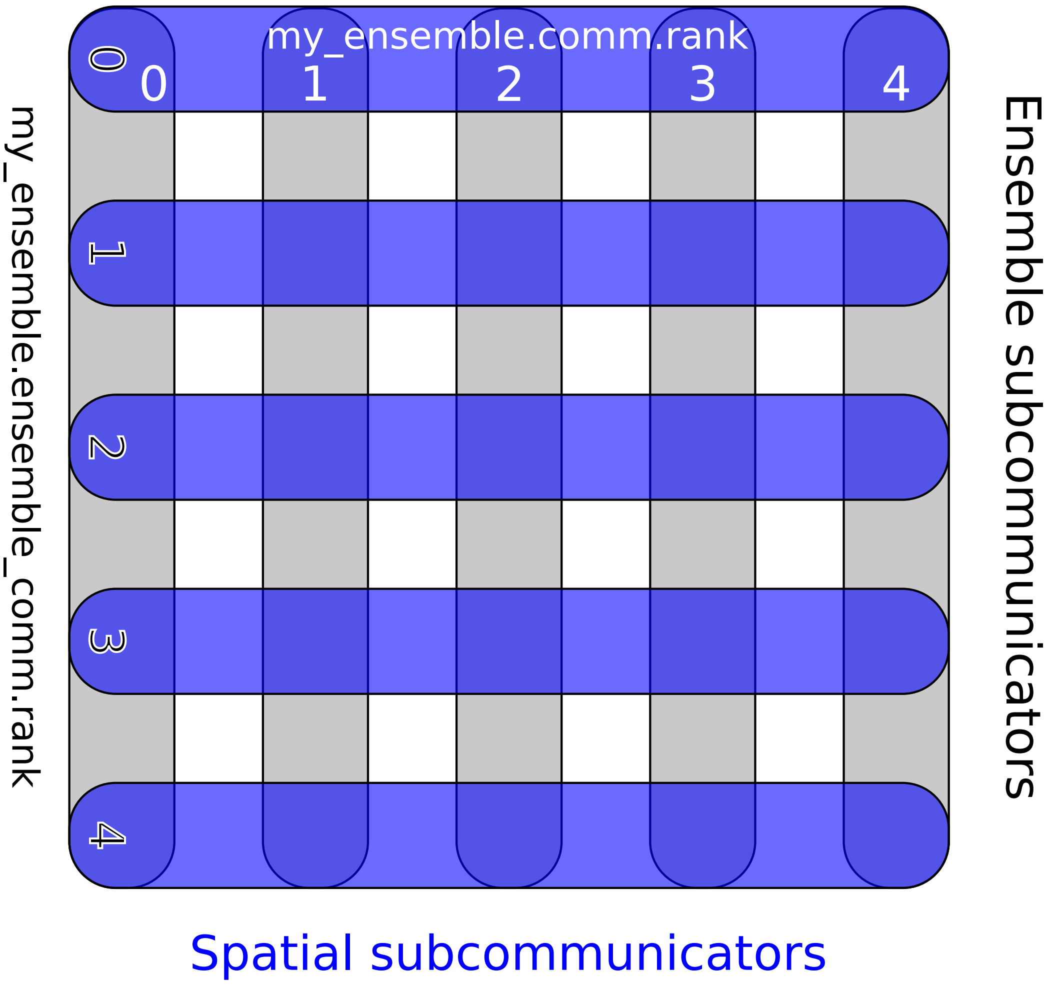


Figure7: Spatial and ensemble paralellism for an ensemble with 5 members, each of which is executed in parallel over 5 processors.

The additional functionality required to support ensemble parallelism is the ability to send instances of *Function* from one ensemble to another. This is handled by the *Ensemble* class. Instantiating an ensemble requires a communicator (usually `MPI_COMM_WORLD`) plus the number of MPI processes to be used in each member of the ensemble (5, in the case of the example below). Each ensemble member will have the same spatial parallelism with the number of ensemble members given by dividing the size of the original communicator by the number processes in each ensemble member. The total number of processes launched by `mpiexec` must therefore be equal to the product of number of ensemble members with the number of processes to be used for each ensemble member.

```
from firedrake import *

my_ensemble = Ensemble(COMM_WORLD, 5)
```

Then, the spatial sub-communicator must be passed to *Mesh()* (or via inbuilt mesh generators in *utility_meshes*), so that it will then be used by function spaces and functions derived from the mesh.

```
mesh = UnitSquareMesh(20, 20, comm=my_ensemble.comm)
x, y = SpatialCoordinate(mesh)
V = FunctionSpace(mesh, "CG", 1)
u = Function(V)
```

The ensemble sub-communicator is then available through the method `ensemble_comm`.

```
q = Constant(my_ensemble.ensemble_comm.rank + 1)
u.interpolate(sin(q*pi*x)*cos(q*pi*y))
```

MPI communications across the spatial sub-communicator (i.e., within an ensemble member) are handled automatically by Firedrake, whilst MPI communications across the ensemble sub-communicator (i.e., between ensemble members) are handled through methods of *Ensemble*. Currently send/recv, reductions and broadcasts are supported, as well as their non-blocking variants.

```
my_ensemble.send(u, dest)
my_ensemble.recv(u, source)

my_ensemble.reduce(u, usum, root)
my_ensemble.allreduce(u, usum)

my_ensemble.bcast(u, root)
```

1.15 Firedrake Zenodo integration: tools for reproducible science

Zenodo provides a facility for archiving scientific data, such as software. Zenodo provides secure archiving and referability, including digital object identifiers (DOIs). Firedrake integrates with Zenodo and GitHub to provide Firedrake users with the ability to generate a set of DOIs corresponding to the exact set of Firedrake components which were used to conduct a particular simulation.

These DOIs can be used in citations in publications to provide a reference to the exact version of the software used, and thereby to improve the reproducibility of your computational science.

1.15.1 How to register DOIs for a version of Firedrake

This section assumes that you have a Firedrake installation which you have used to conduct some numerical experiment and which you wish to publish or otherwise record for posterity. It is assumed that your virtualenv is activated or that you otherwise have the firedrake scripts in your path.

1. Use `firedrake-zenodo` to generate a JSON file containing the versions of Firedrake components you are using, as well as a title describing what this version was used for (this will appear online on Zenodo). For example:

   ```
   firedrake-zenodo -t "My paper title"
   ```

   You can additionally provide a single file that contains any extra (free-form) information that you want to appear in the uploaded Zenodo record:

   ```
   firedrake-zenodo -t "My paper title" --info-file README.txt
   ```

   This file could, for example, contain DOIs of any archived simulation code that you used over and above the core Firedrake components. It can also be a single python script or a tarball (or other archive) of your code and any data required to reproduce your results.

   This will create a file `firedrake.json` containing the required information.

2. Create an issue on the Firedrake GitHub page asking that a Zenodo release be created. Attach the `firedrake.json` file to the issue. You can create the issue here.

3. The Firedrake developers will generate a bespoke Firedrake release containing exactly the set of versions your JSON file specifies, as well as creating a Zenodo record collating these. You will be provided with a firedrake release tag of the form `Firedrake_YYYYMMDD.N`.

   You can see an example such a collated record here.

4. You can use this release tag to generate a BibTeX entry (including the DOI) for the collated “meta”-record, which in turn links to all the individual components:

   ```
   firedrake-zenodo --bibtex Firedrake_YYYYMMDD.N
   ```

   Obviously, you substitute in your Firedrake release tag.

You can explore the full set of options for `firedrake-zenodo` with:

```
firedrake-zenodo -h
```

Installing an archived release

firedrake-install has support for installing a Zenodo-archived release. If you have a DOI for a particular Zenodo release, you can install the matching set of components with:

```
firedrake-install --doi MY_ZENODO_DOI
```

Note: `firedrake-update` will not work out of the box in this scenario, because the components are checked out in a detached head state.

1.15.2 What else do you need to do?

Archive your code

`firedrake-zenodo` produces citable DOIs which point to the versions of Firedrake components you used. This covers your bases as far as concerns Firedrake, but doesn’t cover your code which uses Firedrake. Best practice in computational science also demands that you provide the code which you used to conduct your experiments. You could attach a tarball as a supplement to your paper, embed a tarball (or single script) in the Zenodo release generated to record your Firedrake components, or you could also use Zenodo to generate a DOI directly from your GitHub source repository. Using Zenodo in combination with GitHub for this purpose is documented by github here.

Note: If you archive your code before running `firedrake-zenodo`, you can ensure that the eventual release also references these DOIs by providing them in a text file via the `--info-file` argument. You can also directly attach your code (either a single script or a single archive containing it) to the Firedrake Zenodo release using the same argument.

Cite your sources

Citing custom DOIs for particular versions of Firedrake and its dependencies aids readers of your papers in reproducing your science. However it's a supplement to, and not a replacement for, citing the published resources for the computational methods you are employing. Firedrake also offers support for citing the papers on which your computations depend. This is documented on the *Citing Firedrake* page.

1.16 Optimising Firedrake Performance

"Premature optimisation is the root of all evil"

—Donald Knuth

Performance of a Firedrake script is rarely optimal from the outset. Choice of solver options, discretisation and variational form all have an impact on the amount of time your script takes to run. More general programming considerations such as not repeating unnecessary work inside of a loop can also be signficant.

It is always a bad idea to attempt to optimise your code without a solid understanding of where the bottlenecks are, else you could spend vast amounts of developer time resulting in little to no improvement in performance. The best strategy for performance optimisation should therefore always be to start at the highest level possible with an overview of the entire problem before drilling down into specific hotspots. To get this high level understanding of your script we strongly recommend that you first profile your script using a flame graph (see *below*).

1.16.1 Automatic flame graph generation with PETSc

Flame graphs are a very useful entry point when trying to optimise your application since they make hotspots easy to find. PETSc can generate a flame graph input file using its logging infrastructure that Firedrake has extended by annotating many of its own functions with PETSc events. This allows users to easily generate informative flame graphs giving a lot of insight into the internals of Firedrake and PETSc.

As an example, here is a flame graph showing the performance of the scalar wave equation with higher-order mass lumping demo. It is interactive and you can zoom in on functions by clicking.

One can immediately see that the dominant hotspots for this code are assembly and writing to output so any optimisation effort should be spent in those. Some time is also spent in `firedrake.__init__` but this corresponds to the amount of time spent importing Firedrake and would be amortized for longer-running problems.

Flame graphs can also be generated for codes run in parallel with the reported times in the graph given by the maximum value across all ranks.

Generating the flame graph

To generate a flame graph from your Firedrake script you need to:

1. Run your code with the extra flag `-log_view :foo.txt:ascii_flamegraph`. For example:

   ```
   $ python myscript.py -log_view :foo.txt:ascii_flamegraph
   ```

   This will run your program as usual but output an additional file called `foo.txt` containing the profiling information.

2. Visualise the results. This can be done in one of two ways:

   - Generate an SVG file using the `flamegraph.pl` script from this repository with the command:

     ```
     $ ./flamegraph.pl foo.txt > foo.svg
     ```

     You can then view `foo.svg` in your browser.

   - Upload the file to speedscope and view it there.

Adding your own events

It is very easy to add your own events to the flame graph and there are a few different ways of doing it. The simplest methods are:

- With a context manager:

  ```
  from firedrake.petsc import PETSc

  with PETSc.Log.Event("foo"):
      do_something_expensive()
  ```

- With a decorator:

  ```
  from firedrake.petsc import PETSc

  @PETSc.Log.EventDecorator("foo")
  def do_something_expensive():
      ...
  ```

  If no arguments are passed to `PETSc.Log.EventDecorator` then the event name will be the same as the function.

Caveats

- The `flamegraph.pl` script assumes by default that the values in the stack traces are sample counts. This means that if you hover over functions in the SVG it will report the count in terms of 'samples' rather than the correct unit of microseconds. A simple fix to this is to include the command line option `--countname us` when you generate the SVG. For example:

  ```
  $ ./flamegraph.pl --countname us foo.txt > foo.svg
  ```

- If you use PETSc stages in your code these will be ignored in the flame graph.
- If you call `PETSc.Log.begin()` as part of your script/package then profiling will not work as expected. This is because this function starts PETSc's default (flat) logging while we need to use nested logging instead.

  This issue can be avoided with the simple guard:

  ```
  from firedrake.petsc import OptionsManager

  # If the -log_view flag is passed you don't need to call
  # PETSc.Log.begin because it is done automatically.
  if "log_view" not in OptionsManager.commandline_options:
      PETSc.Log.begin()
  ```

1.16.2 Common performance issues

Calling `solve` repeatedly

When solving PDEs, Firedrake uses a PETSc `SNES` (nonlinear solver) under the hood. Every time the user calls *`solve()`* a new `SNES` is created and used to solve the problem. This is a convenient shorthand for scripts that only need to solve a problem once, but it is fairly expensive to set up a new `SNES` and so repeated calls to *`solve()`* will introduce some overhead.

To get around this problem, users should instead instantiate a variational problem (e.g. *`NonlinearVariationalProblem`*) and solver (e.g. *`NonlinearVariationalSolver`*) outside of the loop body. An example showing how this is done can be found in this demo.

1.16.3 Other useful tools

Here we present a handful of performance analysis tools that users may find useful to run with their codes.

py-spy

py-spy is a great sampling profiler that outputs directly to SVG flame graphs. It allows users to see the entire stack trace of the program rather than just the annotated PETSc events and unlike most Python profilers it can also profile native code.

A flame graph for your Firedrake script can be generated from py-spy with:

```
$ py-spy record -o foo.svg --native -- python myscript.py
```

Beyond the inherent uncertainty that comes from using a sampling profiler, one substantial limitation of py-spy is that it does not work when run in parallel.

pyinstrument

pyinstrument is a great sample-based profiling tool that you can use to easily identify hotspots in your code. To use the profiler simply run:

```
$ pyinstrument myscript.py
```

This will print out a timed callstack to the terminal. To instead generate an interactive graphic you can view in your browser pass the `-r html` flag.

Unfortunately, pyinstrument cannot profile native code. This means that information about the code's execution inside of PETSc is largely lost.

memory_profiler

memory_profiler is a useful tool that you can use to monitor the memory usage of your script. After installing it you can simply run:

```
$ mprof run python myscript.py
$ mprof plot
```

The former command will run your script and generate a file containing the profiling information. The latter then displays a plot of the memory usage against execution time for the whole script.

memory_profiler also works in parallel. You can pass either of the `--include-children` or `--multiprocess` flags to `mprof` depending on whether or not you want to accumulate the memory usage across ranks or plot them separately. For example:

```
$ mprof run --include-children mpiexec -n 4 python myscript.py
```

Score-P

Score-P is a tool aimed at HPC users. We found it to provide some useful insight into MPI considerations such as load balancing and communication overhead.

To use it with Firedrake, users will also need to install Score-P's Python bindings.

CHAPTER

TWO

INTRODUCTORY TUTORIALS

2.1 Simple Helmholtz equation

Let's start by considering the modified Helmholtz equation on a unit square, Ω, with boundary Γ:

$$\begin{aligned}-\nabla^2 u + u &= f \\ \nabla u \cdot \vec{n} &= 0 \quad \text{on } \Gamma\end{aligned}$$

for some known function f. The solution to this equation will be some function $u \in V$, for some suitable function space V, that satisfies these equations. Note that this is the Helmholtz equation that appears in meteorology, rather than the indefinite Helmholtz equation $\nabla^2 u + u = f$ that arises in wave problems.

We transform the equation into weak form by multiplying by an arbitrary test function in V, integrating over the domain and then integrating by parts. The variational problem so derived reads: find $u \in V$ such that:

$$\int_\Omega \nabla u \cdot \nabla v + uv \,\mathrm{d}x = \int_\Omega vf \,\mathrm{d}x + \cancel{\int_\Gamma v \nabla u \cdot \vec{n} \mathrm{d}s}$$

Note that the boundary condition has been enforced weakly by removing the surface term resulting from the integration by parts.

We can choose the function f, so we take:

$$f = (1.0 + 8.0\pi^2)\cos(2\pi x)\cos(2\pi y)$$

which conveniently yields the analytic solution:

$$u = \cos(2\pi x)\cos(2\pi y)$$

However we wish to employ this as an example for the finite element method, so lets go ahead and produce a numerical solution.

First, we always need a mesh. Let's have a 10×10 element unit square:

```
from firedrake import *
mesh = UnitSquareMesh(10, 10)
```

We need to decide on the function space in which we'd like to solve the problem. Let's use piecewise linear functions continuous between elements:

```
V = FunctionSpace(mesh, "CG", 1)
```

We'll also need the test and trial functions corresponding to this function space:

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

We declare a function over our function space and give it the value of our right hand side function:

```
f = Function(V)
x, y = SpatialCoordinate(mesh)
f.interpolate((1+8*pi*pi)*cos(x*pi*2)*cos(y*pi*2))
```

We can now define the bilinear and linear forms for the left and right hand sides of our equation respectively:

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

Finally we solve the equation. We redefine *u* to be a function holding the solution:

```
u = Function(V)
```

Since we know that the Helmholtz equation is symmetric, we instruct PETSc to employ the conjugate gradient method and do not worry about preconditioning for the purposes of this demo

```
solve(a == L, u, solver_parameters={'ksp_type': 'cg', 'pc_type': 'none'})
```

For more details on how to specify solver parameters, see the section of the manual on *solving PDEs*.

Next, we might want to look at the result, so we output our solution to a file:

```
File("helmholtz.pvd").write(u)
```

This file can be visualised using paraview.

We could use the built-in plotting functions of firedrake by calling `tripcolor` to make a pseudo-color plot. Before that, matplotlib.pyplot should be installed and imported:

```
try:
  import matplotlib.pyplot as plt
except:
  warning("Matplotlib not imported")

try:
  fig, axes = plt.subplots()
  colors = tripcolor(u, axes=axes)
  fig.colorbar(colors)
except Exception as e:
  warning("Cannot plot figure. Error msg: '%s'" % e)
```

The plotting functions in Firedrake mimic those of matplotlib; to produce a contour plot instead of a pseudocolor plot, we can call *tricontour* instead:

```
try:
  fig, axes = plt.subplots()
  contours = tricontour(u, axes=axes)
  fig.colorbar(contours)
except Exception as e:
  warning("Cannot plot figure. Error msg: '%s'" % e)
```

Don't forget to show the image:

```
try:
  plt.show()
except Exception as e:
  warning("Cannot show figure. Error msg: '%s'" % e)
```

Alternatively, since we have an analytic solution, we can check the L_2 norm of the error in the solution:

```
f.interpolate(cos(x*pi*2)*cos(y*pi*2))
print(sqrt(assemble(dot(u - f, u - f) * dx)))
```

A python script version of this demo can be found here.

2.2 Burgers equation

The Burgers equation is a non-linear equation for the advection and diffusion of momentum. Here we choose to write the Burgers equation in two dimensions to demonstrate the use of vector function spaces:

$$\frac{\partial u}{\partial t} + (u \cdot \nabla)u - \nu\nabla^2 u = 0$$
$$(n \cdot \nabla)u = 0 \text{ on } \Gamma$$

where Γ is the domain boundary and ν is a constant scalar viscosity. The solution u is sought in some suitable vector-valued function space V. We take the inner product with an arbitrary test function $v \in V$ and integrate the viscosity term by parts:

$$\int_\Omega \frac{\partial u}{\partial t} \cdot v + ((u \cdot \nabla)u) \cdot v + \nu\nabla u \cdot \nabla v \,\mathrm{d}x = 0.$$

The boundary condition has been used to discard the surface integral. Next, we need to discretise in time. For simplicity and stability we elect to use a backward Euler discretisation:

$$\int_\Omega \frac{u^{n+1} - u^n}{dt} \cdot v + ((u^{n+1} \cdot \nabla)u^{n+1}) \cdot v + \nu\nabla u^{n+1} \cdot \nabla v \,\mathrm{d}x = 0.$$

We can now proceed to set up the problem. We choose a resolution and set up a square mesh:

```
from firedrake import *
n = 30
mesh = UnitSquareMesh(n, n)
```

We choose degree 2 continuous Lagrange polynomials. We also need a piecewise linear space for output purposes:

```
V = VectorFunctionSpace(mesh, "CG", 2)
V_out = VectorFunctionSpace(mesh, "CG", 1)
```

We also need solution functions for the current and the next timestep. Note that, since this is a nonlinear problem, we don't define trial functions:

```
u_ = Function(V, name="Velocity")
u = Function(V, name="VelocityNext")

v = TestFunction(V)
```

For this problem we need an initial condition:

```
x = SpatialCoordinate(mesh)
ic = project(as_vector([sin(pi*x[0]), 0]), V)
```

We start with current value of u set to the initial condition, but we also use the initial condition as our starting guess for the next value of u:

```
u_.assign(ic)
u.assign(ic)
```

ν is set to a (fairly arbitrary) small constant value:

```
nu = 0.0001
```

The timestep is set to produce an advective Courant number of around 1. Since we are employing backward Euler, this is stricter than is required for stability, but ensures good temporal resolution of the system's evolution:

```
timestep = 1.0/n
```

Here we finally get to define the residual of the equation. In the advection term we need to contract the test function v with $(u \cdot \nabla)u$, which is the derivative of the velocity in the direction u. This directional derivative can be written as `dot(u,nabla_grad(u))` since `nabla_grad(u)[i, j]`$= \partial_i u_j$. Note once again that for a nonlinear problem, there are no trial functions in the formulation. These will be created automatically when the residual is differentiated by the nonlinear solver:

```
F = (inner((u - u_)/timestep, v)
     + inner(dot(u,nabla_grad(u)), v) + nu*inner(grad(u), grad(v)))*dx
```

We now create an object for output visualisation:

```
outfile = File("burgers.pvd")
```

Output only supports visualisation of linear fields (either P1, or P1DG). In this example we project to a linear space by hand. Another option is to let the *`File`* object manage the decimation. It supports both interpolation to linears (the default) or projection (by passing project_output=True when creating the *File*). Outputting data is carried out using the *write()* method of *File* objects:

```
outfile.write(project(u, V_out, name="Velocity"))
```

Finally, we loop over the timesteps solving the equation each time and outputting each result. Firedrake's default solver parameters are used, which amount to applying a full LU decomposition as a preconditioner.

```
t = 0.0
end = 0.5
while (t <= end):
    solve(F == 0, u)
    u_.assign(u)
    t += timestep
    outfile.write(project(u, V_out, name="Velocity"))
```

A python script version of this demo can be found here.

2.3 Mixed formulation for the Poisson equation

We're considering the Poisson equation $\nabla^2 u = -f$ using a mixed formulation on two coupled fields. We start by introducing the negative flux $\sigma = \nabla u$ as an auxiliary vector-valued variable. This leaves us with the PDE on a unit square $\Omega = [0,1] \times [0,1]$ with boundary Γ

$$\begin{aligned} \sigma - \nabla u &= 0 \text{ on } \Omega \\ \nabla \cdot \sigma &= -f \text{ on } \Omega \\ u &= u_0 \text{ on } \Gamma_D \\ \sigma \cdot n &= g \text{ on } \Gamma_N \end{aligned}$$

for some known function f. The solution to this equation will be some functions $u \in V$ and $\sigma \in \Sigma$ for some suitable function space V and Σ that satisfy these equations. We multiply by arbitrary test functions $\tau \in \Sigma$ and $\nu \in V$, integrate over the domain and then integrate by parts to obtain a weak formulation of the variational problem: find $\sigma \in \Sigma$ and $\nu \in V$ such that:

$$\int_{\Omega} (\sigma \cdot \tau + \nabla \cdot \tau \, u) \, \mathrm{d}x = \int_{\Gamma} \tau \cdot n \, u \, \mathrm{d}s \quad \forall \, \tau \in \Sigma,$$

$$\int_{\Omega} \nabla \cdot \sigma v \, \mathrm{d}x = - \int_{\Omega} f \, v \, \mathrm{d}x \quad \forall \, v \in V.$$

The flux boundary condition $\sigma \cdot n = g$ becomes an *essential* boundary condition to be enforced on the function space, while the boundary condition $u = u_0$ turn into a *natural* boundary condition which enters into the variational form, such that the variational problem can be written as: find $(\sigma, u) \in \Sigma_g \times V$ such that

$$a((\sigma, u), (\tau, v)) = L((\tau, v)) \quad \forall \, (\tau, v) \in \Sigma_0 \times V$$

with the variational forms a and L defined as

$$\begin{aligned} a((\sigma, u), (\tau, v)) &= \int_{\Omega} \sigma \cdot \tau + \nabla \cdot \tau \, u + \nabla \cdot \sigma \, v \, \mathrm{d}x \\ L((\tau, v)) &= - \int_{\Omega} f v \, \mathrm{d}x + \int_{\Gamma_D} u_0 \tau \cdot n \, \mathrm{d}s \end{aligned}$$

The essential boundary condition is reflected in function spaces $\Sigma_g = \{\tau \in H(\mathrm{div})$ such that $\tau \cdot n|_{\Gamma_N} = g\}$ and $V = L^2(\Omega)$.

We need to choose a stable combination of discrete function spaces $\Sigma_h \subset \Sigma$ and $V_h \subset V$ to form a mixed function space $\Sigma_h \times V_h$. One such choice is Brezzi-Douglas-Marini elements of polynomial order k for Σ_h and discontinuous elements of polynomial order $k-1$ for V_h.

For the remaining functions and boundaries we choose:

$$\Gamma_D = \{(0, y) \cup (1, y) \in \partial\Omega\}, \Gamma_N = \{(x, 0) \cup (x, 1) \in \partial\Omega\}$$

$$u_0 = 0, g = \sin(5x)$$

$$f = 10\ e^{-\frac{(x-0.5)^2+(y-0.5)^2}{0.02}}$$

To produce a numerical solution to this PDE in Firedrake we procede as follows:

The mesh is chosen as a 32×32 element unit square.

```
from firedrake import *
mesh = UnitSquareMesh(32, 32)
```

As argued above, a stable choice of function spaces for our problem is the combination of order k Brezzi-Douglas-Marini (BDM) elements and order $k-1$ discontinuous Galerkin elements (DG). We use $k = 1$ and combine the BDM and DG spaces into a mixed function space `W`.

```
BDM = FunctionSpace(mesh, "BDM", 1)
DG = FunctionSpace(mesh, "DG", 0)
W = BDM * DG
```

We obtain test and trial functions on the subspaces of the mixed function spaces as follows:

```
sigma, u = TrialFunctions(W)
tau, v = TestFunctions(W)
```

Next we declare our source function `f` over the DG space and initialise it with our chosen right hand side function value.

```
x, y = SpatialCoordinate(mesh)
f = Function(DG).interpolate(
    10*exp(-(pow(x - 0.5, 2) + pow(y - 0.5, 2)) / 0.02))
```

After dropping the vanishing boundary term on the right hand side, the bilinear and linear forms of the variational problem are defined as:

```
a = (dot(sigma, tau) + div(tau)*u + div(sigma)*v)*dx
L = - f*v*dx
```

The strongly enforced boundary conditions on the BDM space on the top and bottom of the domain are declared as:

```
bc0 = DirichletBC(W.sub(0), as_vector([0.0, -sin(5*x)]), 3)
bc1 = DirichletBC(W.sub(0), as_vector([0.0, sin(5*x)]), 4)
```

Note that it is necessary to apply these boundary conditions to the first subspace of the mixed function space using `W.sub(0)`. This way the association with the mixed space is preserved.

Declaring it on the BDM space directly is *not* the same and would in fact cause the application of the boundary condition during the later solve to fail.

Now we're ready to solve the variational problem. We define *w* to be a function to hold the solution on the mixed space.

```
w = Function(W)
```

Then we solve the linear variational problem `a == L` for `w` under the given boundary conditions `bc0` and `bc1` using Firedrake's default solver parameters. Afterwards we extract the components `sigma` and `u` on each of the subspaces with `split`.

```
solve(a == L, w, bcs=[bc0, bc1])
sigma, u = w.split()
```

Lastly we write the component of the solution corresponding to the primal variable on the DG space to a file in VTK format for later inspection with a visualisation tool such as ParaView

```
File("poisson_mixed.pvd").write(u)
```

We could use the built in plot function of firedrake by calling *`plot`* to plot a surface graph. Before that, matplotlib.pyplot should be installed and imported:

```
try:
  import matplotlib.pyplot as plt
except:
  warning("Matplotlib not imported")

try:
  fig, axes = plt.subplots()
  colors = tripcolor(u, axes=axes)
  fig.colorbar(colors)
except Exception as e:
  warning("Cannot plot figure. Error msg '%s'" % e)
```

Don't forget to show the image:

```
try:
  plt.show()
except Exception as e:
  warning("Cannot show figure. Error msg '%s'" % e)
```

This demo is based on the corresponding DOLFIN mixed Poisson demo and can be found as a script in poisson_mixed.py.

2.4 DG advection equation with upwinding

We next consider the advection equation

$$\frac{\partial q}{\partial t} + (\vec{u} \cdot \nabla)q = 0$$

in a domain Ω, where $\vec{u}$ is a prescribed vector field, and $q(\vec{x}, t)$ is an unknown scalar field. The value of q is known initially:

$$q(\vec{x}, 0) = q_0(\vec{x}),$$

and the value of q is known for all time on the subset of the boundary Γ in which $\vec{u}$ is directed towards the interior of the domain:

$$q(\vec{x}, t) = q_{\mathrm{in}}(\vec{x}, t) \quad \text{on } \Gamma_{\mathrm{inflow}}$$

where Γ_{inflow} is defined appropriately.

We will look for a solution q in a space of *discontinuous* functions V. A weak form of the continuous equation in each element e is

$$\int_e \phi_e \frac{\partial q}{\partial t} \,\mathrm{d}x + \int_e \phi_e (\vec{u} \cdot \nabla) q \,\mathrm{d}x = 0, \qquad \forall \phi_e \in V_e,$$

where we explicitly introduce the subscript e since the test functions ϕ_e are local to each element. Using integration by parts on the second term, we get

$$\int_e \phi_e \frac{\partial q}{\partial t} \,\mathrm{d}x = \int_e q \nabla \cdot (\phi_e \vec{u}) \,\mathrm{d}x - \int_{\partial e} \phi_e q \vec{u} \cdot \vec{n}_e \,\mathrm{d}S, \qquad \forall \phi_e \in V_e,$$

where $\vec{n}_e$ is an outward-pointing unit normal.

Since q is discontinuous, we have to make a choice about how to define q on facets when we assemble the equations globally. We will use upwinding: we choose the *upstream* value of q on facets, with respect to the velocity field $\vec{u}$. We note that there are three types of facets that we may encounter:

1. Interior facets. Here, the value of q from the upstream side, denoted $\widetilde{q}$, is used.
2. Inflow boundary facets, where $\vec{u}$ points towards the interior. Here, the upstream value is the prescribed boundary value q_{in}.
3. Outflow boundary facets, where $\vec{u}$ points towards the outside. Here, the upstream value is the interior solution value q.

We must now express our problem in terms of integrals over the entire mesh and over the sets of interior and exterior facets. This is done by summing our earlier expression over all elements e. The cell integrals are easy to handle, since $\sum_e \int_e \cdot \,\mathrm{d}x = \int_\Omega \cdot \,\mathrm{d}x$. The interior facet integrals are more difficult to express, since each facet in the set of interior facets Γ_{int} appears twice in the $\sum_e \int_{\partial e}$. In other words, contributions arise from both of the neighbouring cells.

In Firedrake, the separate quantities in the two cells neighbouring an interior facet are denoted by + and -. These markings are arbitrary – there is no built-in concept of upwinding, for example – and the user is responsible for providing a form that works in all cases. We will give an example shortly. The exterior facet integrals are easier to handle, since each facet in the set of exterior facets Γ_{ext} appears exactly once in $\sum_e \int_{\partial e}$. The full equations are then

$$\begin{aligned}\int_\Omega \phi \frac{\partial q}{\partial t}\,\mathrm{d}x = &\int_\Omega q\nabla\cdot(\phi\vec{u})\,\mathrm{d}x\\ &- \int_{\Gamma_{\text{int}}} \widetilde{q}(\phi_+\vec{u}\cdot\vec{n}_+ + \phi_-\vec{u}\cdot\vec{n}_-)\,\mathrm{d}S\\ &- \int_{\Gamma_{\text{ext,inflow}}} \phi q_{\text{in}}\vec{u}\cdot\vec{n}\,\mathrm{d}s\\ &- \int_{\Gamma_{\text{ext,outflow}}} \phi q\vec{u}\cdot\vec{n}\,\mathrm{d}s \qquad \forall\phi\in V.\end{aligned}$$

As a timestepping scheme, we use the three-stage strong-stability-preserving Runge-Kutta (SSPRK) scheme from [SO88]: to discretise $\frac{\partial q}{\partial t} = \mathcal{L}(q)$, we set

$$\begin{aligned}q^{(1)} &= q^n + \Delta t\mathcal{L}(q^n)\\ q^{(2)} &= \frac{3}{4}q^n + \frac{1}{4}(q^{(1)} + \Delta t\mathcal{L}(q^{(1)}))\\ q^{n+1} &= \frac{1}{3}q^n + \frac{2}{3}(q^{(2)} + \Delta t\mathcal{L}(q^{(2)}))\end{aligned}$$

In this worked example, we reproduce the classic cosine-bell–cone–slotted-cylinder advection test case of [LeV96]. The domain Ω is the unit square $\Omega = [0,1]\times[0,1]$, and the velocity field corresponds to solid body rotation $\vec{u} = (0.5 - y, x - 0.5)$. Each side of the domain has a section of inflow and a section of outflow boundary. We therefore perform both the inflow and outflow integrals over the entire boundary, but construct them so that they only contribute in the correct places.

As usual, we start by importing Firedrake. We also import the math library to give us access to the value of pi. We use a 40-by-40 mesh of squares.

```
from firedrake import *
import math
import matplotlib.pyplot as plt
from matplotlib.animation import FuncAnimation

mesh = UnitSquareMesh(40, 40, quadrilateral=True)
```

We set up a function space of discontinous bilinear elements for q, and a vector-valued continuous function space for our velocity field.

```
V = FunctionSpace(mesh, "DQ", 1)
W = VectorFunctionSpace(mesh, "CG", 1)
```

We set up the initial velocity field using a simple analytic expression.

```
x, y = SpatialCoordinate(mesh)

velocity = as_vector((0.5 - y, x - 0.5))
u = Function(W).interpolate(velocity)
```

Now, we set up the cosine-bell–cone–slotted-cylinder initial coniditon. The first four lines declare various parameters relating to the positions of these objects, while the analytic expressions appear in the last three lines.

```
bell_r0 = 0.15; bell_x0 = 0.25; bell_y0 = 0.5
cone_r0 = 0.15; cone_x0 = 0.5; cone_y0 = 0.25
cyl_r0 = 0.15; cyl_x0 = 0.5; cyl_y0 = 0.75
slot_left = 0.475; slot_right = 0.525; slot_top = 0.85

bell = 0.25*(1+cos(math.pi*min_value(sqrt(pow(x-bell_x0, 2) + pow(y-bell_y0,␣
↪2))/bell_r0, 1.0)))
cone = 1.0 - min_value(sqrt(pow(x-cone_x0, 2) + pow(y-cone_y0, 2))/cyl_r0, 1.
↪0)
slot_cyl = conditional(sqrt(pow(x-cyl_x0, 2) + pow(y-cyl_y0, 2)) < cyl_r0,
             conditional(And(And(x > slot_left, x < slot_right), y < slot_
↪top),
               0.0, 1.0), 0.0)
```

We then declare the inital condition of q to be the sum of these fields. Furthermore, we add 1 to this, so that the initial field lies between 1 and 2, rather than between 0 and 1. This ensures that we can't get away with neglecting the inflow boundary condition. We also save the initial state so that we can check the L^2-norm error at the end.

```
q = Function(V).interpolate(1.0 + bell + cone + slot_cyl)
q_init = Function(V).assign(q)
```

Next we'll create a list to store the function values at every timestep so that we can make a movie of them later.

```
qs = []
```

We will run for time 2π, a full rotation. We take 600 steps, giving a timestep close to the CFL limit. We declare an extra variable `dtc`; for technical reasons, this means that Firedrake does not have to compile new C code if the user tries different timesteps. Finally, we define the inflow boundary condition, q_{in}. In general, this would be a `Function`, but here we just use a `Constant` value.

```
T = 2*math.pi
dt = T/600.0
dtc = Constant(dt)
q_in = Constant(1.0)
```

Now we declare our variational forms. Solving for Δq at each stage, the explicit timestepping scheme means that the left hand side is just a mass matrix.

```
dq_trial = TrialFunction(V)
phi = TestFunction(V)
a = phi*dq_trial*dx
```

The right-hand-side is more interesting. We define `n` to be the built-in `FacetNormal` object; a unit normal vector that can be used in integrals over exterior and interior facets. We next define un to be an object which is equal to $\vec{u} \cdot \vec{n}$ if this is positive, and zero if this is negative. This will be useful in the upwind terms.

```
n = FacetNormal(mesh)
un = 0.5*(dot(u, n) + abs(dot(u, n)))
```

We now define our right-hand-side form `L1` as Δt times the sum of four integrals.

The first integral is a straightforward cell integral of $q\nabla\cdot(\phi\vec{u})$. The second integral represents the inflow boundary condition. We only want this to contribute on the inflow part of the boundary, where $\vec{u}\cdot\vec{n} < 0$ (recall that $\vec{n}$ is an outward-pointing normal). Where this is true, the condition gives the desired expression $\phi q_{\text{in}}\vec{u}\cdot\vec{n}$, otherwise the condition gives zero. The third integral operates in a similar way to give the outflow boundary condition. The last integral represents the integral $\tilde{q}(\phi_+\vec{u}\cdot\vec{n}_+ + \phi_-\vec{u}\cdot\vec{n}_-)$ over interior facets. We could again use a conditional in order to represent the upwind value $\tilde{q}$ by the correct choice of q_+ or q_-, depending on the sign of $\vec{u}\cdot\vec{n_+}$, say. Instead, we make use of the quantity `un`, which is either $\vec{u}\cdot\vec{n}$ or zero, in order to avoid writing explicit conditionals. Although it is not obvious at first sight, the expression given in code is equivalent to the desired expression, assuming $\vec{n}_- = -\vec{n}_+$.

```
L1 = dtc*(q*div(phi*u)*dx
          - conditional(dot(u, n) < 0, phi*dot(u, n)*q_in, 0.0)*ds
          - conditional(dot(u, n) > 0, phi*dot(u, n)*q, 0.0)*ds
          - (phi('+') - phi('-'))*(un('+')*q('+') - un('-')*q('-'))*dS)
```

In our Runge-Kutta scheme, the first step uses q^n to obtain $q^{(1)}$. We therefore declare similar forms that use $q^{(1)}$ to obtain $q^{(2)}$, and $q^{(2)}$ to obtain q^{n+1}. We make use of UFL's `replace` feature to avoid writing out the form repeatedly.

```
q1 = Function(V); q2 = Function(V)
L2 = replace(L1, {q: q1}); L3 = replace(L1, {q: q2})
```

We now declare a variable to hold the temporary increments at each stage.

```
dq = Function(V)
```

Since we want to perform hundreds of timesteps, ideally we should avoid reassembling the left-hand-side mass matrix each step, as this does not change. We therefore make use of the `LinearVariationalProblem` and `LinearVariationalSolver` objects for each of our Runge-Kutta stages. These cache and reuse the assembled left-hand-side matrix. Since the DG mass matrices are block-diagonal, we use the 'preconditioner' ILU(0) to solve the linear systems. As a minor technical point, we in fact use an outer block Jacobi preconditioner. This allows the code to be executed in parallel without any further changes being necessary.

```
params = {'ksp_type': 'preonly', 'pc_type': 'bjacobi', 'sub_pc_type': 'ilu'}
prob1 = LinearVariationalProblem(a, L1, dq)
solv1 = LinearVariationalSolver(prob1, solver_parameters=params)
prob2 = LinearVariationalProblem(a, L2, dq)
solv2 = LinearVariationalSolver(prob2, solver_parameters=params)
prob3 = LinearVariationalProblem(a, L3, dq)
solv3 = LinearVariationalSolver(prob3, solver_parameters=params)
```

We now run the time loop. This consists of three Runge-Kutta stages, and every 20 steps we write out the solution to file and print the current time to the terminal.

```
t = 0.0
step = 0
output_freq = 20
while t < T - 0.5*dt:
    solv1.solve()
    q1.assign(q + dq)

    solv2.solve()
    q2.assign(0.75*q + 0.25*(q1 + dq))

    solv3.solve()
    q.assign((1.0/3.0)*q + (2.0/3.0)*(q2 + dq))

    step += 1
    t += dt

    if step % output_freq == 0:
        qs.append(q.copy(deepcopy=True))
        print("t=", t)
```

To check our solution, we display the normalised L^2 error, by comparing to the initial condition.

```
L2_err = sqrt(assemble((q - q_init)*(q - q_init)*dx))
L2_init = sqrt(assemble(q_init*q_init*dx))
print(L2_err/L2_init)
```

Finally, we'll animate our solution using matplotlib. We'll need to evaluate the solution at many points in every frame of the animation, so we'll employ a helper class that pre-computres some relevant data in order to speed up the evaluation.

```
nsp = 16
fn_plotter = FunctionPlotter(mesh, num_sample_points=nsp)
```

We first set up a figure and axes and draw the first frame.

```
fig, axes = plt.subplots()
axes.set_aspect('equal')
colors = tripcolor(q_init, num_sample_points=nsp, vmin=1, vmax=2, axes=axes)
fig.colorbar(colors)
```

Now we'll create a function to call in each frame. This function will use the helper object we created before.

```
def animate(q):
    colors.set_array(fn_plotter(q))
```

The last step is to make the animation and save it to a file.

```
interval = 1e3 * output_freq * dt
animation = FuncAnimation(fig, animate, frames=qs, interval=interval)
try:
```

(continues on next page)

(continued from previous page)

```
    animation.save("DG_advection.mp4", writer="ffmpeg")
except:
    print("Failed to write movie! Try installing `ffmpeg`.")
```

This demo can be found as a script in DG_advection.py.

References

2.5 Steady-state continuity equation on an extruded mesh

This demo showcases the use of extruded meshes, including the new regions of integration and the construction of sophisticated finite element spaces.

We now consider the equation

$$\begin{aligned} \nabla \cdot (\vec{u} q) &= 0 \\ q &= q_{\mathrm{in}} \quad \text{on } \Gamma_{\mathrm{inflow}}, \end{aligned}$$

in a domain Ω, where $\vec{u}$ is a prescribed vector field, and q is an unknown scalar field. The value of q is known on the 'inflow' part of the boundary Γ, where $\vec{u}$ is directed towards the interior of the domain. q can be interpreted as the steady-state distribution of a passive tracer carried by a fluid with velocity field $\vec{u}$.

We apply an upwind DG method, as we saw in the *previous example*. Denoting the upwind value of q on interior facets by $\widetilde{q}$, the full set of equations are then

$$-\int_\Omega q\vec{u_0} \cdot \nabla\phi \, \mathrm{d}x + \int_{\Gamma_{\mathrm{ext,outflow}}} \phi q \vec{u} \cdot \vec{n} \, \mathrm{d}s + \int_{\Gamma_{\mathrm{int}}} (\phi_+ \vec{u} \cdot \vec{n}_+ + \phi_- \vec{u} \cdot \vec{n}_-) \widetilde{q} \, \mathrm{d}S \quad = \quad -\int_{\Gamma_{\mathrm{ext,inflow}}} \phi q_{\mathrm{in}} \vec{u} \cdot \vec{n} \, \mathrm{d}s \quad \forall\, \phi \in V,$$

We will take the domain Ω to be the cuboid $\Omega = [0,1] \times [0,1] \times [0,0.2]$. We will use the uniform velocity field $\vec{u} = (0,0,1)$. Γ_{inflow} is therefore the base of the cuboid, while $\Gamma_{\mathrm{outflow}}$ is the top. The four vertical sides can be ignored, since $\vec{u} \cdot \vec{n} = 0$ on these faces.

We use an *extruded* mesh, where the base mesh is a 20 by 20 unit square, divided into triangles, with 10 evenly-spaced vertical layers. This gives prism-shaped cells.

```
from firedrake import *
m = UnitSquareMesh(20, 20)
mesh = ExtrudedMesh(m, layers=10, layer_height=0.02)
```

We will use a simple piecewise-constant function space for the unknown scalar q:

```
V = FunctionSpace(mesh, "DG", 0)
```

Our velocity will live in a low-order Raviart-Thomas space. The construction of this is more complicated than element spaces that have appeared previously. The horizontal and vertical components of the field are specified separately. They are combined into a single element which is used to build a FunctionSpace.

```
# RT1 element on a prism
W0_h = FiniteElement("RT", "triangle", 1)
W0_v = FiniteElement("DG", "interval", 0)
W0 = HDivElement(TensorProductElement(W0_h, W0_v))
W1_h = FiniteElement("DG", "triangle", 0)
W1_v = FiniteElement("CG", "interval", 1)
W1 = HDivElement(TensorProductElement(W1_h, W1_v))
W_elt = W0 + W1
W = FunctionSpace(mesh, W_elt)
```

As an aside, since our prescibed velocity is purely in the vertical direction, a simpler space would have sufficed:

```
# Vertical part of RT1 element
# W_h = FiniteElement("DG", "triangle", 0)
# W_v = FiniteElement("CG", "interval", 1)
# W_elt = HDivElement(TensorProductElement(W_h, W_v))
# W = FunctionSpace(mesh, W_elt)
```

Or even:

```
# Why can't everything in life be this easy?
# W = VectorFunctionSpace(mesh, "CG", 1)
```

Next, we set the prescribed velocity field:

```
velocity = as_vector((0.0, 0.0, 1.0))
u = project(velocity, W)

# if we had used W = VectorFunctionSpace(mesh, "CG", 1), we could have done
# u = Function(W)
# u.interpolate(velocity)
```

Next, we will set the boundary value on our scalar to be a simple indicator function over part of the bottom of the domain:

```
x, y, z = SpatialCoordinate(mesh)
inflow = conditional(And(z < 0.02, x > 0.5), 1.0, -1.0)
q_in = Function(V)
q_in.interpolate(inflow)
```

Now we will define our forms. We use the same trick as in the *previous example* of defining `un` to aid with the upwind terms:

```
n = FacetNormal(mesh)
un = 0.5*(dot(u, n) + abs(dot(u, n)))
```

We define our trial and test functions in the usual way:

```
q = TrialFunction(V)
phi = TestFunction(V)
```

Since we are on an extruded mesh, we have several new integral types at our disposal. An integral over the cells of the domain is still denoted by `dx`. Boundary integrals now come in several varieties: `ds_b` denotes an integral over the base of the mesh, while `ds_t` denotes an integral over the top of the mesh. `ds_v` denotes an integral over the sides of a mesh, though we will not use that here.

Similiarly, interior facet integrals are split into `dS_h` and `dS_v`, over *horizontal* interior facets and *vertical* interior facets respectively. Since our velocity field is purely in the vertical direction, we will omit the integral over vertical interior facets, since we know $\vec{u} \cdot \vec{n}$ is zero for these.

```
a1 = -q*dot(u, grad(phi))*dx
a2 = dot(jump(phi), un('+')*q('+') - un('-')*q('-'))*dS_h
a3 = dot(phi, un*q)*ds_t  # outflow at top wall
a = a1 + a2 + a3

L = -q_in*phi*dot(u, n)*ds_b  # inflow at bottom wall
```

Finally, we will compute the solution:

```
out = Function(V)
solve(a == L, out)
```

By construction, the exact solution is quite simple:

```
exact = Function(V)
exact.interpolate(conditional(x > 0.5, 1.0, -1.0))
```

We finally compare our solution to the expected solution:

```
assert max(abs(out.dat.data - exact.dat.data)) < 1e-10
```

This demo can be found as a script in extruded_continuity.py.

2.6 Double slit experiment

Here we solve a linear wave equation using an explicit timestepping scheme. This example demonstrates the use of an externally generated mesh, pointwise operations on Functions, and a time varying boundary condition. The strong form of the equation we set out to solve is:

$$\frac{\partial^2\phi}{\partial t^2} - \nabla^2\phi = 0$$
$$\nabla\phi \cdot n = 0 \text{ on } \Gamma_N$$
$$\phi = \frac{1}{10\pi}\cos(10\pi t) \text{ on } \Gamma_D$$

To facilitate our choice of time integrator, we make the substitution:

$$\frac{\partial\phi}{\partial t} = -p$$
$$\frac{\partial p}{\partial t} + \nabla^2\phi = 0$$
$$\nabla\phi \cdot n = 0 \text{ on } \Gamma_N$$
$$p = \sin(10\pi t) \text{ on } \Gamma_D$$

We then form the weak form of the equation for p. Find $p \in V$ such that:

$$\int_\Omega \frac{\partial p}{\partial t} v \, \mathrm{d}x = \int_\Omega \nabla\phi \cdot \nabla v \, \mathrm{d}x \quad \forall v \in V$$

For a suitable function space V. Note that the absence of spatial derivatives in the equation for ϕ makes the weak form of this equation equivalent to the strong form so we will solve it pointwise.

In time we use a simple symplectic method in which we offset p and ϕ by a half timestep.

This time we created the mesh with Gmsh:

```
gmsh -2 wave_tank.geo
```

We can then start our Python script and load this mesh:

```
from firedrake import *
mesh = Mesh("wave_tank.msh")
```

We choose a degree 1 continuous function space, and set up the function space and functions. Setting the *name* parameter when constructing `Function` objects will set the name used in the output file:

```
V = FunctionSpace(mesh, 'Lagrange', 1)
p = Function(V, name="p")
phi = Function(V, name="phi")

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

Output the initial conditions:

```
outfile = File("out.pvd")
outfile.write(phi)
```

We next establish a boundary condition object. Since we have time-dependent boundary conditions, we first create a `Constant` to hold the value and use that:

```
bcval = Constant(0.0)
bc = DirichletBC(V, bcval, 1)
```

Now we set the timestepping variables:

```
T = 10.
dt = 0.001
t = 0
step = 0
```

Finally we set a flag indicating whether we wish to perform mass-lumping in the timestepping scheme:

```
lump_mass = True
```

Now we are ready to start the timestepping loop:

```
while t <= T:
    step += 1
```

Update the boundary condition value for this timestep:

```
bcval.assign(sin(2*pi*5*t))
```

Step forward ϕ by half a timestep. Since this does not involve a matrix inversion, this is implemented as a pointwise operation:

```
phi -= dt / 2 * p
```

Now step forward p. This is an explicit timestepping scheme which only requires the inversion of a mass matrix. We have two options at this point, we may either *lump* the mass, which reduces the inversion to a pointwise division:

```
if lump_mass:
    p += assemble(dt * inner(nabla_grad(v), nabla_grad(phi))*dx) /␣
↪assemble(v*dx)
```

In the mass lumped case, we must now ensure that the resulting solution for p satisfies the boundary conditions:

```
bc.apply(p)
```

Alternatively, we can invert the mass matrix using a linear solver:

```
else:
    solve(u * v * dx == v * p * dx + dt * inner(grad(v), grad(phi)) * dx,
          p, bcs=bc, solver_parameters={'ksp_type': 'cg',
                                        'pc_type': 'sor',
                                        'pc_sor_symmetric': True})
```

Step forward ϕ by the second half timestep:

```
phi -= dt / 2 * p
```

Advance time and output as appropriate, note how we pass the current timestep value into the *write()* method, so that when visualising the results Paraview will use it:

```
t += dt
if step % 10 == 0:
    outfile.write(phi, time=t)
```

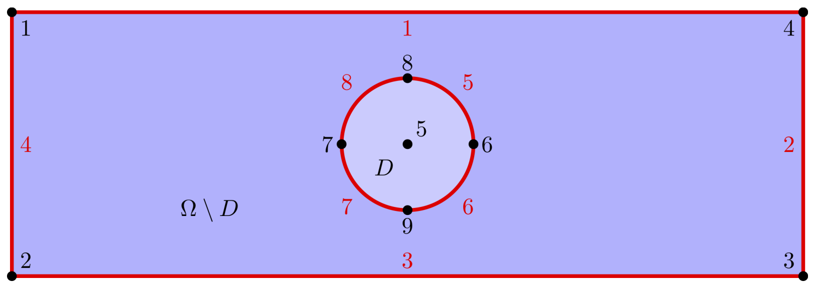
An animation, produced in Paraview, illustrating the output of this simulation can be found on youtube.

A python script version of this demo can be found here. The gmsh input file is here.

2.7 Creating Firedrake-compatible meshes in Gmsh

The purpose of this demo is to summarize the key structure of a `gmsh.geo` file that creates a Firedrake-compatible mesh. For more details about Gmsh, please refer to the Gmsh documentation. The Gmsh syntax used in this document is for Gmsh version 4.4.1 .

As example, we will construct and mesh the following geometry: a rectangle with a disc in the middle. In the picture, numbers in black refer to Gmsh point tags, whereas numbers in read refer to Gmsh curve tags (see below).



The first thing we define are four corners of a rectangle. We specify the x,y, and z(=0) coordinates, as well as the target element size at these corners (which we set to 0.5).

```
Point(1) = {-6,  2, 0, 0.5};
Point(2) = {-6, -2, 0, 0.5};
Point(3) = { 6, -2, 0, 0.5};
Point(4) = { 6,  2, 0, 0.5};
```

Then, we define 5 points to describe a circle.

```
Point(5) = { 0,  0, 0, 0.1};
Point(6) = { 1,  0, 0, 0.1};
Point(7) = {-1,  0, 0, 0.1};
Point(8) = { 0,  1, 0, 0.1};
Point(9) = { 0, -1, 0, 0.1};
```

Then, we create 8 edges: 4 for the rectangle and 4 for the circle. Note that the Gmsh command `Circle` requires the arc to be strictly smaller than π.

```
Line(1) = {1, 4};
Line(2) = {4, 3};
Line(3) = {3, 2};
Line(4) = {2, 1};
Circle(5) = {8, 5, 6};
Circle(6) = {6, 5, 9};
Circle(7) = {9, 5, 7};
Circle(8) = {7, 5, 8};
```

Then, we glue together the rectangle edges and, separately, the circle edges. Note that `Line`, `Circle`, and `Curve Loop` (as well as `Physical Curve` below) are all curves in Gmsh and must possess a unique tag.

```
Curve Loop( 9) = {1, 2, 3, 4};
Curve Loop(10) = {8, 5, 6, 7};
```

Then, we define two plane surfaces: the rectangle without the disc first, and the disc itself then.

```
Plane Surface(1) = {9, 10};
Plane Surface(2) = {10};
```

Finally, we group together some edges and define `Physical` entities. Firedrake uses the tags of these physical identities to distinguish between parts of the mesh (see the concrete example at the end of this page).

```
Physical Curve("HorEdges", 11) = {1, 3};
Physical Curve("VerEdges", 12) = {2, 4};
Physical Curve("Circle", 13) = {8, 7, 6, 5};
Physical Surface("PunchedDom", 3) = {1};
Physical Surface("Disc", 4) = {2};
```

For simplicity, we have gathered all this commands in the file immersed_domain.geo. To generate a mesh using this file, you can type the following command in the terminal

```
gmsh -2 immersed_domain.geo -format msh2
```

Note: Depending on your version of gmsh and DMPlex, the gmsh option `-format msh2` may be omitted.

To illustrate how to access all these features within Firedrake, we consider the following interface problem. Denoting by Ω the filled rectangle and by D the disc, we seek a function $u \in H^1_0(\Omega)$ such that

$$-\nabla \cdot (\sigma \nabla u) + u = 5 \quad \text{in } \Omega$$

where $\sigma = 1$ in $\Omega \setminus D$ and $\sigma = 2$ in D. Since σ attains different values across ∂D, we need to prescribe the behavior of u across this interface. This is implicitly done by imposing $u \in H^1_0(\Omega)$: the function u must be continuous across $\partial\Omega$. This allows us to employ Lagrangian finite elements to approximate u. However, we also need to specify the the jump of $\sigma\nabla u \cdot \vec{n}$ on ∂D. This term arises naturally in the weak formulation of the problem under consideration. In this demo we simply set

$$[\![\sigma \nabla u \cdot \vec{n}]\!] = 3 \quad \text{on } \partial D$$

The resulting weak formulation reads as follows:

$$\int_\Omega \sigma \nabla u \cdot \nabla v + uv \,\mathrm{d}\mathbf{x} - \int_{\partial D} 3v \,\mathrm{d}S = \int_\Omega 5v \,\mathrm{d}\mathbf{x} \quad \text{for every } v \in H^1_0(\Omega)\,.$$

The following Firedrake code shows how to solve this variational problem using linear Lagrangian finite elements.

```
from firedrake import *

# load the mesh generated with Gmsh
mesh = Mesh('immersed_domain.msh')

```

(continues on next page)

(continued from previous page)

```
# define the space of linear Lagrangian finite elements
V = FunctionSpace(mesh, "CG", 1)

# define the trial function u and the test function v
u = TrialFunction(V)
v = TestFunction(V)

# define the bilinear form of the problem under consideration
# to specify the domain of integration, the surface tag is specified in␣
↪brackets after dx
# in this example, 3 is the tag of the rectangle without the disc, and 4 is␣
↪the disc tag
a = 2*dot(grad(v), grad(u))*dx(4) + dot(grad(v), grad(u))*dx(3) + v*u*dx

# define the linear form of the problem under consideration
# to specify the boundary of the boundary integral, the boundary tag is␣
↪specified after dS
# note the use of dS due to 13 not being an external boundary
# Since the dS integral is an interior one, we must restrict the
# test function: since the space is continuous, we arbitrarily pick
# the '+' side.
L = Constant(5.) * v * dx + Constant(3.)*v('+')*dS(13)

# set homogeneous Dirichlet boundary conditions on the rectangle boundaries
# the tag  11 referes to the horizontal edges, the tag 12 refers to the␣
↪vertical edges
DirBC = DirichletBC(V, 0, [11, 12])

# define u to contain the solution to the problem under consideration
u = Function(V)

# solve the variational problem
solve(a == L, u, bcs=DirBC, solver_parameters={'ksp_type': 'cg'})
```

A python script version of this demo can be found here.

CHAPTER

THREE

ADVANCED TUTORIALS

These tutorials demonstrate some more advanced features of Firedrake's PDE solving capabilities, such as block-preconditioning mixed finite element systems.

3.1 Basic printing in parallel

Contributed by Ed Bueler.

This example shows how one may print various quantities in parallel. The Firedrake public interface mostly works as-is in parallel but several of the operations here expose the PETSc and MPI underpinnings in order to print.

Run this example in parallel using P processes by doing `mpiexec -n P python3 parprint.py`.

We start with the usual import but we also import petsc4py so that classes `PETSc.X` are available. Here `X` is one of the PETSc object types, including types like Vec:

```
from firedrake import *
from firedrake.petsc import PETSc
```

In serial the next line could be `print('setting up mesh...')` However, in parallel that would print P times on P processes. In the following form the print happens only once (because it is done only on rank 0):

```
PETSc.Sys.Print('setting up mesh across %d processes' % COMM_WORLD.size)
```

Next we generate a mesh. It has an MPI communicator `mesh.comm`, equal to `COMM_WORLD` by default. By using the `COMM_SELF` communicator each rank reports on the portion of the mesh it owns:

```
mesh = UnitSquareMesh(3, 3)
PETSc.Sys.Print('  rank %d owns %d elements and can access %d vertices' \
                % (mesh.comm.rank, mesh.num_cells(), mesh.num_vertices()),
                comm=COMM_SELF)
```

The *elements* of the mesh are owned uniquely in parallel, while the vertices are shared via "halos" or "ghost vertices". Note there is a nontrivial relationship between vertices and degrees of freedom in a global PETSc Vec (below).

We use a familiar Helmholtz equation problem merely for demonstration. First we set up a weak form just as in the helmholtz.py demo:

```
V = FunctionSpace(mesh, "CG", 1)
u = TrialFunction(V)
v = TestFunction(V)
f = Function(V)
x,y = SpatialCoordinate(mesh)
f.interpolate((1+8*pi*pi)*cos(x*pi*2)*cos(y*pi*2))
a = (dot(grad(v), grad(u)) + v * u) * dx
L = f * v * dx
```

Then solve:

```
PETSc.Sys.Print('solving problem ...')
u = Function(V)
solve(a == L, u, options_prefix='s', solver_parameters={'ksp_type': 'cg'})
```

To print the solution vector in serial one could write `print(u.dat.data)` but then in parallel each processor would show its data separately. So using PETSc we do a "view" of the solution vector:

```
with u.dat.vec_ro as vu:
    vu.view()
```

Here `vu` is an instance of the PETSc.Vec class and `vu.view()` is the equivalent of `VecView(vu, NULL)` using PETSc's C API. This Vec is "global", meaning that each degree of freedom is stored on a unique process. The context manager in the above usage (i.e. `with ...`) allows Firedrake to generate a global Vec by halo exchanges if needed. Here we only need read-only access here so we use `u.dat.vec_ro`; note `u.dat.vec` would allow read-write access.

Finally we compute and print the numerical error, relative to the exact solution, in two norms. The L^2 norm is computed with `assemble` which already includes an MPI reduction across the `mesh.comm` communicator:

```
udiff = Function(V).interpolate(u - cos(x*pi*2)*cos(y*pi*2))
L_2_err = sqrt(assemble(dot(udiff,udiff) * dx))
```

We compute the L^∞ error a different way. Note that `u.dat.data.max()` works in serial but in parallel that only gets the max over the process-owned entries. So again we use the `PETSc.Vec` approach:

```
udiffabs = Function(V).interpolate(abs(udiff))
with udiffabs.dat.vec_ro as v:
    L_inf_err = v.max()[1]
PETSc.Sys.Print('L_2 error norm = %g, L_inf error norm = %g' \
                % (L_2_err,L_inf_err))
```

Note: `max()` on a `PETSc.Vec` returns an `(index,max)` pair, thus the `[1]` to obtain the max value.

3.2 Benney-Luke equations: a reduced water wave model

This tutorial was contributed by Anna Kalogirou and Onno Bokhove.

The work is based on the article "Variational water wave modelling: from continuum to experiment" by Onno Bokhove and Anna Kalogirou [BK16]. The authors gratefully acknowledge funding from EPSRC grant no. EP/L025388/1 with a link to the Dutch Technology Foundation STW for the project "FastFEM: behavior of fast ships in waves".

The Benney-Luke-type equations consist of a reduced potential flow water wave model based on the assumptions of small amplitude parameter ϵ and small dispersion parameter μ (defined by the square of the ratio of the typical depth over a horizontal length scale). They describe the deviation from the still water surface, $\eta(x, y, t)$, and the free surface potential, $\phi(x, y, t)$. A modified version of the Benney-Luke equations can be obtained by the variational principle:

$$\begin{aligned}
0 &= \delta \int_0^T \int_\Omega \eta\phi_t - \frac{\mu}{2}\eta\Delta\phi_t + \frac{1}{2}\eta^2 + \frac{1}{2}(1+\epsilon\eta)|\nabla\phi|^2 + \frac{\mu}{3}(\Delta\phi)^2 \; dx\,dy\,dt \\
&= \delta \int_0^T \int_\Omega \eta\phi_t + \frac{\mu}{2}\nabla\eta\cdot\nabla\phi_t + \frac{1}{2}\eta^2 + \frac{1}{2}(1+\epsilon\eta)|\nabla\phi|^2 + \mu\left(\nabla q\cdot\nabla\phi - \frac{3}{4}q^2\right) \; dx\,dy\,dt \\
&= \int_0^T \int_\Omega \Big(\delta\eta\,\phi_t + \frac{\mu}{2}\nabla\delta\eta\cdot\nabla\phi_t + \eta\,\delta\eta + \frac{\epsilon}{2}\delta\eta\,|\nabla\phi|^2\Big) \\
&\qquad - \Big(\delta\phi\,\eta_t + \frac{\mu}{2}\nabla\eta_t\cdot\nabla\delta\phi - (1+\epsilon\eta)\nabla\phi\cdot\nabla\delta\phi - \mu\nabla q\cdot\nabla\delta\phi\Big) \\
&\qquad + \mu\left(\nabla\delta q\cdot\nabla\phi - \frac{3}{2}q\,\delta q\right) \; dx\,dy\,dt,
\end{aligned}$$

where the spatial domain is assumed to be Ω with natural boundary conditions, namely Neumann conditions on all the boundaries. In addition, suitable end-point conditions at $t = 0$ and $t = T$ are used. Note that the introduction of the auxiliary function q is performed in order to lower the highest derivatives. This is advantageous in a C^0 finite element formulation and motivated the modification of the "standard" Benney-Luke equations. The partial variations in the last line of the variational principle can be integrated by parts in order to get expressions that only depend on $\delta\eta$, $\delta\phi$, δq and not their derivatives:

$$\begin{aligned}
0 = \int_0^T \int_\Omega & \Big(\phi_t - \frac{\mu}{2}\Delta\phi_t + \eta + \frac{\epsilon}{2}|\nabla\phi|^2\Big)\,\delta\eta \\
& - \Big(\eta_t - \frac{\mu}{2}\Delta\eta_t + \nabla\cdot\big((1+\epsilon\eta)\nabla\phi\big) + \mu\Delta q\Big)\,\delta\phi \\
& - \mu\left(\Delta\phi + \frac{3}{2}q\right)\delta q \, dx\,dy\,dt.
\end{aligned}$$

Since the variations $\delta\eta$, $\delta\phi$, δq are arbitrary, the modified Benney-Luke equations then arise for functions $\eta, \phi, q \in V$ from a suitable function space V and are given by:

$$\begin{aligned}
\phi_t - \frac{\mu}{2}\Delta\phi_t + \eta + \frac{\epsilon}{2}|\nabla\phi|^2 &= 0 \\
\eta_t - \frac{\mu}{2}\Delta\eta_t + \nabla\cdot\big((1+\epsilon\eta)\nabla\phi\big) + \mu\Delta q &= 0 \\
q &= -\frac{2}{3}\Delta\phi.
\end{aligned}$$

We can either directly use the partial variations in the variational principle above (last line) as the fundamental weak formulation (with $\delta\phi$, $\delta\eta$, δq playing the role of test functions), or multiply the equations by a test function $v \in V$ and integrate over the domain in order to obtain a weak formulation in a classic manner

$$\int_\Omega \phi_t\, v + \frac{\mu}{2}\nabla\phi_t \cdot \nabla v + \eta\, v + \frac{\epsilon}{2}\nabla\phi\cdot\nabla\phi\, v\, dx\, dy = 0$$
$$\int_\Omega \eta_t\, v + \frac{\mu}{2}\nabla\eta_t\cdot\nabla v - (1+\epsilon\eta)\,\nabla\phi\cdot\nabla v - \mu\nabla q\cdot\nabla v\, dx\, dy = 0$$
$$\int_\Omega q\, v - \frac{2}{3}\nabla\phi\cdot\nabla v\, dx\, dy = 0.$$

Note that the Neumann boundary conditions have been used to remove every surface term that resulted from the integration by parts. Moreover, the variational form of the system requires the use of a symplectic integrator for the time-discretisation. Here we choose the 2nd-order Stormer-Verlet scheme [EHW06], which requires two half-steps to update ϕ in time (one implicit and one explicit in general) and one (implicit) step for η:

$$\begin{aligned}\int_\Omega \frac{\phi^{n+1/2}-\phi^n}{\frac{1}{2}dt}\, v + \frac{\mu}{2}\nabla\left(\frac{\phi^{n+1/2}-\phi^n}{\frac{1}{2}dt}\right)\cdot\nabla v \\ +\eta^n\, v + \frac{\epsilon}{2}\nabla\phi^{n+1/2}\cdot\nabla\phi^{n+1/2}\, v\, dx\, dy = 0\end{aligned}$$

$$\int_\Omega q^{n+1/2}\, v - \frac{2}{3}\nabla\phi^{n+1/2}\cdot\nabla v\, dx\, dy = 0$$

$$\begin{aligned}\int_\Omega \frac{\eta^{n+1}-\eta^n}{dt}\, v + \frac{\mu}{2}\nabla\left(\frac{\eta^{n+1}-\eta^n}{dt}\right)\cdot\nabla v \\ -\frac{1}{2}\Big(\big(1+\epsilon\eta^{n+1}\big) + (1+\epsilon\eta^n)\Big)\nabla\phi^{n+1/2}\cdot\nabla v \\ -\mu\nabla q^{n+1/2}\cdot\nabla v\, dx\, dy = 0\end{aligned}$$

$$\begin{aligned}\int_\Omega \frac{\phi^{n+1}-\phi^{n+1/2}}{\frac{1}{2}dt}\, v + \frac{\mu}{2}\nabla\left(\frac{\phi^{n+1}-\phi^{n+1/2}}{\frac{1}{2}dt}\right)\cdot\nabla v \\ +\eta^{n+1}\, v + \frac{\epsilon}{2}\nabla\phi^{n+1/2}\cdot\nabla\phi^{n+1/2}\, v\, dx\, dy = 0\end{aligned}$$

$$\int_\Omega q^{n+1}\, v - \frac{2}{3}\nabla\phi^{n+1}\cdot\nabla v\, dx\, dy = 0.$$

Furthermore, we note that the Benney-Luke equations admit asymptotic solutions (correct up to order ϵ). The "exact" solutions can be found by assuming one-dimensional travelling waves of the type

$$\eta(x,y,t) = \eta(\xi,\tau), \quad \phi(x,y,t) = \Phi(\xi,\tau), \qquad \text{with} \qquad \xi = \sqrt{\frac{\epsilon}{\mu}}(x-t), \quad \tau = \epsilon\sqrt{\frac{\epsilon}{\mu}}t, \quad \Phi = \sqrt{\frac{\epsilon}{\mu}}\phi.$$

The Benney-Luke equations then become equivalent to a Korteweg-de Vries (KdV) equation for η at leading order in ϵ. The soliton solution of the KdV [DJ89] travels with speed c and is reflected when reaching the solid wall. The initial propagation before reflection matches the asymptotic solution for the surface elevation η well. The asymptotic solution for the surface potential ϕ can be found by using $\eta = \phi_\xi$ (correct at leading order), giving

$$\eta(x, y, t) = \frac{c}{3}\text{sech}^2\left(\frac{1}{2}\sqrt{\frac{c\epsilon}{\mu}}\left(x - x_0 - t - \frac{\epsilon}{6}ct\right)\right),$$
$$\phi(x, y, t) = \frac{2}{3}\sqrt{\frac{c\mu}{\epsilon}}\left(\tanh\left(\frac{1}{2}\sqrt{\frac{c\epsilon}{\mu}}\left(x - x_0 - t - \frac{\epsilon}{6}ct\right)\right) + 1\right).$$

Finally, before implementing the problem in Firedrake, we calculate the total energy defined by the sum of potential and kinetic energy. The system is then stable if the energy is bounded and shows no drift. The expression for total energy is given by:

$$E(t) = \int_\Omega \frac{1}{2}\eta^2 + \frac{1}{2}(1 + \epsilon\eta)\left|\nabla\phi\right|^2 + \mu\left(\nabla q \cdot \nabla\phi - \frac{3}{4}q^2\right)\, dx\, dy.$$

The implementation of this problem in Firedrake requires solving two nonlinear variational problems and one linear problem. The Benney-Luke equations are solved in a rectangular domain $\Omega = [0, 10] \times [0, 1]$, with $\mu = \epsilon = 0.01$, time step $dt = 0.005$ and up to the final time $T = 2.0$. Additionally, the domain is split into 50 cells in the x-direction using a quadrilateral mesh. In the y-direction only 1 cell is enough since there are no variations in y:

```
from firedrake import *
```

Now we move on to defining parameters:

```
T = 2.0
dt = 0.005
Lx = 10
Nx = 50
Ny = 1
c = 1.0
mu = 0.01
epsilon = 0.01

m = UnitIntervalMesh(Nx)
mesh = ExtrudedMesh(m, layers=Ny)
coords = mesh.coordinates
coords.dat.data[:,0] = Lx*coords.dat.data[:,0]
```

The function space chosen consists of degree 2 continuous Lagrange polynomials, and the functions η, ϕ are initialised to take the exact soliton solutions for $t = 0$, centered around the middle of the domain, i.e. with $x_0 = \frac{1}{2}L_x$:

```
V = FunctionSpace(mesh,"CG",2)

eta0 = Function(V, name="eta")
phi0 = Function(V, name="phi")
eta1 = Function(V, name="eta_next")
phi1 = Function(V, name="phi_next")
q1 = Function(V)
phi_h = Function(V)
q_h = Function(V)
ex_eta = Function(V, name="exact_eta")
```

(continues on next page)

(continued from previous page)

```
ex_phi = Function(V, name="exact_phi")

q = TrialFunction(V)
v = TestFunction(V)

x = SpatialCoordinate(mesh)
x0 = 0.5 * Lx
eta0.interpolate(1/3.0*c*pow(cosh(0.5*sqrt(c*epsilon/mu)*(x[0]-x0)),-2))
phi0.interpolate(2/3.0*sqrt(c*mu/epsilon)*(tanh(0.5*sqrt(c*epsilon/mu)*(x[0]-
↪x0))+1))
```

Firstly, ϕ is updated to a half-step value using a nonlinear variational solver to solve the implicit equation:

```
Fphi_h = ( v*(phi_h-phi0)/(0.5*dt) + 0.5*mu*inner(grad(v),grad((phi_h-phi0)/
↪(0.5*dt)))
           + v*eta0 + 0.5*epsilon*inner(grad(phi_h),grad(phi_h))*v )*dx

phi_problem_h = NonlinearVariationalProblem(Fphi_h,phi_h)
phi_solver_h = NonlinearVariationalSolver(phi_problem_h)
```

followed by a calculation of a half-step solution q, performed using a linear solver:

```
aq = v*q*dx
Lq_h = 2.0/3.0*inner(grad(v),grad(phi_h))*dx

q_problem_h = LinearVariationalProblem(aq,Lq_h,q_h)
q_solver_h = LinearVariationalSolver(q_problem_h)
```

Then the nonlinear implicit equation for η is solved:

```
Feta = ( v*(eta1-eta0)/dt + 0.5*mu*inner(grad(v),grad((eta1-eta0)/dt))
         - 0.5*((1+epsilon*eta0)+(1+epsilon*eta1))*inner(grad(v),grad(phi_h))
         - mu*inner(grad(v),grad(q_h)) )*dx

eta_problem = NonlinearVariationalProblem(Feta,eta1)
eta_solver = NonlinearVariationalSolver(eta_problem)
```

and finally the second half-step (explicit this time) for the equation of ϕ is performed and q is computed for the updated solution:

```
Fphi = ( v*(phi1-phi_h)/(0.5*dt) + 0.5*mu*inner(grad(v),grad((phi1-phi_h)/(0.
↪5*dt)))
         + v*eta1 + 0.5*epsilon*inner(grad(phi_h),grad(phi_h))*v )*dx

phi_problem = NonlinearVariationalProblem(Fphi,phi1)
phi_solver = NonlinearVariationalSolver(phi_problem)

Lq = 2.0/3.0*inner(grad(v),grad(phi1))*dx
q_problem = LinearVariationalProblem(aq,Lq,q1)
```

(continues on next page)

(continued from previous page)

```
q_solver = LinearVariationalSolver(q_problem)
```

What is left before iterating over all time steps, is to find the initial energy E_0, used later to evaluate the energy difference $|E - E_0| / E_0$:

```
t = 0
E0 = assemble( (0.5*eta0**2 + 0.5*(1+epsilon*eta0)*abs(grad(phi0))**2
                + mu*(inner(grad(q1),grad(phi0)) - 0.75*q1**2))*dx )
E = E0
```

and define the exact solutions, which need to be updated at every time-step:

```
t_ = Constant(t)
expr_eta = 1/3.0*c*pow(cosh(0.5*sqrt(c*epsilon/mu)*(x[0]-x0-t_-epsilon*c*t_/6.
↪0)),-2)
expr_phi = 2/3.0*sqrt(c*mu/epsilon)*(tanh(0.5*sqrt(c*epsilon/mu)*(x[0]-x0-t_-
↪epsilon*c*t_/6.0))+1)
```

Since we will interpolate these values again and again, we use an *Interpolator* whose *interpolate()* method we can call to perform the interpolation.

```
eta_interpolator = Interpolator(expr_eta, ex_eta)
phi_interpolator = Interpolator(expr_phi, ex_phi)
phi_interpolator.interpolate()
eta_interpolator.interpolate()
```

For visualisation, we save the computed and exact solutions to an output file. Note that the visualised data will be interpolated from piecewise quadratic functions to piecewise linears:

```
output = File('output.pvd')
output.write(phi0, eta0, ex_phi, ex_eta, time=t)
```

We are now ready to enter the main time iteration loop:

```
while t < T:
      print(t, abs((E-E0)/E0))
      t += dt

      t_.assign(t)

      eta_interpolator.interpolate()
      phi_interpolator.interpolate()

      phi_solver_h.solve()
      q_solver_h.solve()
      eta_solver.solve()
      phi_solver.solve()
      q_solver.solve()

      eta0.assign(eta1)
```

(continues on next page)

(continued from previous page)

```
    phi0.assign(phi1)

    output.write(phi0, eta0, ex_phi, ex_eta, time=t)

    E = assemble( (0.5*eta1**2 + 0.5*(1+epsilon*eta1)*abs(grad(phi1))**2
                  + mu*(inner(grad(q1),grad(phi1)) - 0.75*q1**2))*dx )
```

The output can be visualised using paraview.

A python script version of this demo can be found here.

The Benney-Luke system and weak formulations presented in this demo have also been used to model extreme waves that occur due to Mach reflection through the intersection of two obliquely incident solitary waves. More information can be found in [GBK17].

References

3.3 Quasi-Geostrophic Model

This tutorial was contributed by Francis Poulin, based on code from Colin Cotter.

The Quasi-Geostrophic (QG) model is very important in geophysical fluid dynamics as it describes some aspects of large-scale flows in the oceans and atmosphere very well. The interested reader can find derivations in [QG-Ped92] and [QG-Val06].

In these notes we present the nonlinear equations for the one-layer QG model with a free-surface. Then, the weak form will be derived as is needed for Firedrake.

3.3.1 Governing Equations

The Quasi-Geostrophic (QG) model is very similar to the 2D vorticity equation. Since the leading order geostrophic velocity is incompressible in the horizontal, the governing equations can be written as

$$\begin{aligned}\partial_t q + \vec{\nabla} \cdot (\vec{u} q) + \beta v = 0, \\ \vec{u} = \vec{\nabla}^{\perp} \psi, \\ \nabla^2 \psi - \frac{1}{L_d^2} \psi = q.\end{aligned}$$

where the ψ and q are the streamfunction and Potential Vorticity (PV). The Laplacian is 2D since we are only in the horizontal plane and we defined

$$\vec{\nabla}^{\perp} = \hat{e}_z \times \vec{\nabla}.$$

The first equation above states that the PV is conserved following the flow. The second equation forces the leading order velocity to be geostrophic and the third equation is the definition for the QG PV for this barotropic model. To solve this using Finite Elements it is necessary to establish the weak form of the model, which is done in the next subsection.

3.3.2 Weak Form

Evolving the nonlinear equations consists of two steps. First, the elliptic problem must be solved to compute the streamfunction given the PV. Second, the PV equation must be integrated forward in time. This is done using a strong stability preserving Runge Kutta 3 (SSPRK3) method.

Elliptic Equation

First, we focus on the elliptic inversion in the case of a flat bottom. If we compute the inner product of the equation with the test function ϕ we obtain,

$$\begin{aligned}
\langle \nabla^2 \psi, \phi \rangle - \frac{1}{L_d^2}\langle \psi, \phi \rangle &= \langle q, \phi \rangle, \\
\langle \nabla \psi, \nabla \phi \rangle + \frac{1}{L_d^2}\langle \psi, \phi \rangle &= -\langle q, \phi \rangle,
\end{aligned}$$

where in the second equation we used the divergence theorem and the homogeneous Dirichlet boundary conditions on the test function.

Evolution Equation

The SSPRK3 method used as explained in [QG-Got05] can be written as

$$\begin{aligned}
q^{(1)} &= q^n - \Delta t \left[\vec{\nabla} \cdot (\vec{u}^n q^n) + \beta v^n \right], \\
q^{(2)} &= \frac{3}{4} q^n + \frac{1}{4} \left[q^{(1)} - \Delta t \vec{\nabla} \cdot \left(\vec{u}^{(1)} q^{(1)} \right) - \Delta t \beta v^{(1)} \right], \\
q^{n+1} &= \frac{1}{3} q^n + \frac{2}{3} \left[q^{(2)} - \Delta t \vec{\nabla} \cdot \left(\vec{u}^{(2)} q^{(2)} \right) - \Delta t \beta v^{(1)} \right].
\end{aligned}$$

To get the weak form we need to introduce a test function, p, and take the inner product of the first equation with p.

$$\begin{aligned}
\langle q^{(1)}, p \rangle &= \langle q^n, p \rangle - \Delta t \langle \vec{\nabla} \cdot (\vec{u}^n q^n), p \rangle - \Delta t \langle \beta v, q \rangle, \\
\langle q^{(1)}, p \rangle - \Delta t \langle \vec{u}^n q^n, \vec{\nabla} p \rangle + \Delta t \langle \beta v, q \rangle &= \langle q^n, p \rangle - \Delta t \langle \vec{u}^n q^n, p \rangle_{bdry}
\end{aligned}$$

The first and second terms on the left hand side are referred to as a_{mass} and a_{int} in the code. The first term on the right-hand side is referred to as a_{mass} in the code. The second term on the right-hand side is the extra term due to the DG framework, which does not exist in the CG version of the problem and it is referred to as a_{flux}. This above problem must be solved for $q^{(1)}$ and then $q^{(2)}$ and then these are used to compute the numerical approximation to the PV at the new time q^{n+1}.

We now move on to the implementation of the QG model for the case of a freely propagating Rossby wave. As ever, we begin by importing the Firedrake library.

```
from firedrake import *
```

Next we define the domain we will solve the equations on, square domain with 50 cells in each direction that is periodic along the x-axis.

```
Lx = 2.0 * pi  # Zonal length
Ly = 2.0 * pi  # Meridonal length
n0 = 50  # Spatial resolution
mesh = PeriodicRectangleMesh(n0, n0, Lx, Ly, direction="x",␣
↪quadrilateral=True)
```

We define function spaces:

```
Vdg = FunctionSpace(mesh, "DQ", 1)  # DQ elements for Potential Vorticity (PV)
Vcg = FunctionSpace(mesh, "CG", 1)  # CG elements for Streamfunction
Vu = VectorFunctionSpace(mesh, "DQ", 0)  # DQ elements for velocity
```

and initial conditions for the potential vorticity, here we use Firedrake's ability to *interpolate UFL expressions*.

```
x = SpatialCoordinate(mesh)
q0 = Function(Vdg).interpolate(0.1 * sin(x[0]) * sin(x[1]))
```

We define some `Function`s to store the fields:

```
dq1 = Function(Vdg)  # PV fields for different time steps
qh = Function(Vdg)
q1 = Function(Vdg)

psi0 = Function(Vcg)  # Streamfunctions for different time steps
psi1 = Function(Vcg)
```

along with the physical parameters of the model.

```
F = Constant(1.0)  # Rotational Froude number
beta = Constant(0.1)  # beta plane coefficient
Dt = 0.1  # Time step
dt = Constant(Dt)
```

Next, we define the variational problems. First the elliptic problem for the stream function.

```
psi = TrialFunction(Vcg)
phi = TestFunction(Vcg)

# Build the weak form for the inversion
Apsi = (inner(grad(psi), grad(phi)) + F * psi * phi) * dx
Lpsi = -q1 * phi * dx
```

We impose homogeneous dirichlet boundary conditions on the stream function at the top and bottom of the domain.

```
bc1 = DirichletBC(Vcg, 0.0, (1, 2))

psi_problem = LinearVariationalProblem(Apsi, Lpsi, psi0, bcs=bc1, constant_
↪jacobian=True)
```

(continues on next page)

(continued from previous page)

```
psi_solver = LinearVariationalSolver(psi_problem, solver_parameters={"ksp_type
↪": "cg", "pc_type": "hypre"})
```

Next we'll set up the advection equation, for which we need an operator $\vec{\nabla}^{\perp}$, defined as a python anonymouus function:

```
gradperp = lambda u: as_vector((-u.dx(1), u.dx(0)))
```

For upwinding, we'll need a representation of the normal to a facet, and a way of selecting the upwind side:

```
n = FacetNormal(mesh)
un = 0.5 * (dot(gradperp(psi0), n) + abs(dot(gradperp(psi0), n)))
```

Now the variational problem for the advection equation itself.

```
q = TrialFunction(Vdg)
p = TestFunction(Vdg)
a_mass = p * q * dx
a_int = (dot(grad(p), -gradperp(psi0) * q) + beta * p * psi0.dx(0)) * dx
a_flux = (dot(jump(p), un("+") * q("+") - un("-") * q("-"))) * dS
arhs = a_mass - dt * (a_int + a_flux)

q_problem = LinearVariationalProblem(a_mass, action(arhs, q1), dq1)
```

Since the operator is a mass matrix in a discontinuous space, it can be inverted exactly using an incomplete LU factorisation with zero fill.

```
q_solver = LinearVariationalSolver(q_problem,
                                   solver_parameters={"ksp_type": "preonly",
                                                      "pc_type": "bjacobi",
                                                      "sub_pc_type": "ilu"})
```

To visualise the output of the simulation, we create a *File* object. To which we can store multiple *Function*s. So that we can distinguish between them we will give them descriptive names.

```
q0.rename("Potential vorticity")
psi0.rename("Stream function")
v = Function(Vu, name="gradperp(stream function)")
v.project(gradperp(psi0))

output = File("output.pvd")

output.write(q0, psi0, v)
```

Now all that is left is to define the timestepping parameters and execute the time loop.

```
t = 0.0
T = 10.0
dumpfreq = 5
```

(continues on next page)

(continued from previous page)

```
tdump = 0

while t < (T - Dt / 2):
    # Compute the streamfunction for the known value of q0
    q1.assign(q0)
    psi_solver.solve()
    q_solver.solve()

    # Find intermediate solution q^(1)
    q1.assign(dq1)
    psi_solver.solve()
    q_solver.solve()

    # Find intermediate solution q^(2)
    q1.assign(0.75 * q0 + 0.25 * dq1)
    psi_solver.solve()
    q_solver.solve()

    # Find new solution q^(n+1)
    q0.assign(q0 / 3 + 2 * dq1 / 3)

    # Store solutions to xml and pvd
    t += Dt
    print(t)

    tdump += 1
    if tdump == dumpfreq:
        tdump -= dumpfreq
        v.project(gradperp(psi0))
        output.write(q0, psi0, v, time=t)
```

A python script version of this demo can be found here.

References

3.4 Oceanic Basin Modes: Quasi-Geostrophic approach

This tutorial was contributed by Christine Kaufhold and Francis Poulin.

As a continuation of the Quasi-Geostrophic (QG) model described in the other tutorial, we will now see how we can use Firedrake to compute the spatial structure and frequencies of the freely evolving modes in this system, what are referred to as basin modes. Oceanic basin modes are low frequency structures that propagate zonally in the oceans that alter the dynamics of Western Boundary Currents, such as the Gulf Stream. In this particular tutorial we will show how to solve the QG eigenvalue problem with no basic state and no dissipative forces. Unlike the other demo that integrated the equations forward in time, in this problem it is necessary to compute the eigenvalues and eigenfunctions for a particular differential operator. This requires using PETSc matrices and eigenvalue solvers in SLEPc.

This demo requires SLEPc and slepc4py to be installed. This is most easily achieved by providing the optional *–slepc* flag to either *firedrake-install* (for a new installation), or *firedrake-update* (to add SLEPc to an existing installation).

3.4.1 Governing PDE

We first briefly recap the nonlinear, one-layer QG equation that we *considered previously*. The interested reader can find the derivations in [QGeval-Ped92] and [QGeval-Val06]. This model consists of an evolution equation for the Potential Vorticity, q, and an elliptic problem through which we can determine the streamfunction,

$$\partial_t q + \vec{\nabla} \cdot (\vec{u} q) + \beta v = 0$$
$$q = \nabla^2 \psi - F\psi$$

Where ψ is the stream-function, $\vec{u} = (u, v)$ is the velocity field, q is the Potential Vorticity (PV), β is the Coriolis parameter and F is the rotational Froude number. The velocity field is easily obtained using

$$\vec{u} = \vec{\nabla}^{\perp}\psi, \quad \text{with} \quad \vec{\nabla}^{\perp} = \hat{e_z} \times \vec{\nabla}$$

We assume that the amplitude of the wave motion is very small, which allows us to linearize the equations of motion and therefore neglect the nonlinear advection,

$$\frac{\partial}{\partial t}(\nabla^2\psi - F\psi) = -\beta\frac{\partial \psi}{\partial x}$$

We look for wave-like solutions that are periodic in time, with a frequency of ω

$$\psi = \hat{\psi}(x, y)e^{-i\omega t}$$

This has the advantage of removing the time derivative from the equation and replacing it with an eigenvalue, $i\omega$. By substituting the above solution into the QG equation, we can find a complex eigenvalue problem of the form

$$i\omega(\nabla^2\hat{\psi} - F\hat{\psi}) = \hat{\beta}\frac{\partial \hat{\psi}}{\partial x}$$

Weak Formulation

To use a finite element method it is necessary to formulate the weak form and then we can use SLEPc in Firedrake to compute eigenvalue problems easily. To begin, we multiply this equation by a Test Function ϕ and integrate over the domain A.

$$i\omega \iint_A \Big(\phi \cdot \nabla^2\hat{\psi}\, dA - F\phi\hat{\psi}\, dA\Big) = \hat{\beta} \iint_A \phi \cdot \frac{\partial \hat{\psi}}{\partial x}\, dA$$

To remove the Laplacian operator we use integration by parts and the Divergence theorem to obtain

$$\iint_A \phi \cdot \nabla^2\hat{\psi}\, dA = -\iint_A \nabla\phi \cdot \nabla\hat{\psi}\, dA + \oint_{\partial A} \phi \cdot \frac{\partial \hat{\psi}}{\partial n}\, dS$$

No-normal flow boundary conditions are required and mathematically this means that the streamfunction must be a constant on the boundary. Since the test functions inherit these boundary conditions, $\hat{\phi} = 0$ on the boundary, the boundary integral vanishes and the weak form becomes,

$$i\omega \iint_A \Big(\nabla\phi \cdot \nabla\hat{\psi}\, dA + F\phi\hat{\psi}\Big)\, dA = \hat{\beta} \iint_A \phi \cdot \frac{\partial \hat{\psi}}{\partial x}\, dA$$

Firedrake code

Using this form, we can now implement this eigenvalue problem in Firedrake. We import the Firedrake, PETSc, and SLEPc libraries.

```
from firedrake import *
from firedrake.petsc import PETSc
try:
    from slepc4py import SLEPc
except ImportError:
    import sys
    warning("Unable to import SLEPc, eigenvalue computation not possible (try␣
↪firedrake-update --slepc)")
    sys.exit(0)
```

We specify the geometry to be a square geometry with 50 cells with length 1.

```
Lx   = 1.
Ly   = 1.
n0   = 50
mesh = RectangleMesh(n0, n0, Lx, Ly, reorder=None)
```

Next we define the function spaces within which our solution will reside.

```
Vcg  = FunctionSpace(mesh,'CG',3)
```

We impose zero Dirichlet boundary conditions, in a strong sense, which guarantee that we have no-normal flow at the boundary walls.

```
bc = DirichletBC(Vcg, 0.0, "on_boundary")
```

The two non-dimensional parameters are the β parameter, set by the sphericity of the Earth, and the Froude number, the relative importance of rotation to stratification.

```
beta = Constant('1.0')
F    = Constant('1.0')
```

Additionally, we can create some Functions to store the eigenmodes.

```
eigenmodes_real, eigenmodes_imag = Function(Vcg), Function(Vcg)
```

We define the Test Function ϕ and the Trial Function ψ in our function space.

```
phi, psi = TestFunction(Vcg), TrialFunction(Vcg)
```

To build the weak formulation of our equation we need to build two PETSc matrices in the form of a generalized eigenvalue problem, $A\psi = \lambda M\psi$. We impose the boundary conditions on the mass matrix M, since that is where we used integration by parts.

```
a =  beta*phi*psi.dx(0)*dx
m = -inner(grad(psi), grad(phi))*dx - F*psi*phi*dx
```

(continues on next page)

(continued from previous page)

```
petsc_a = assemble(a).M.handle
petsc_m = assemble(m, bcs=bc).M.handle
```

We can declare how many eigenpairs, eigenfunctions and eigenvalues, we want to find

```
num_eigenvalues = 1
```

Next we will impose parameters onto our eigenvalue solver. The first is specifying that we have an generalized eigenvalue problem that is nonhermitian. The second specifies the spectral transform shift factor to be non-zero. The third requires we use a Krylov-Schur method, which is the default so this is not strictly necessary. Then, we ask for the eigenvalues with the largest imaginary part. Finally, we specify the tolerance.

```
opts = PETSc.Options()
opts.setValue("eps_gen_non_hermitian", None)
opts.setValue("st_pc_factor_shift_type", "NONZERO")
opts.setValue("eps_type", "krylovschur")
opts.setValue("eps_largest_imaginary", None)
opts.setValue("eps_tol", 1e-10)
```

Finally, we build our eigenvalue solver using SLEPc. We add our PETSc matrices into the solver as operators and use setFromOptions() to call the PETSc parameters we previously declared.

```
es = SLEPc.EPS().create(comm=COMM_WORLD)
es.setDimensions(num_eigenvalues)
es.setOperators(petsc_a, petsc_m)
es.setFromOptions()
es.solve()
```

Additionally we can find the number of converged eigenvalues.

```
nconv = es.getConverged()
```

We now get the real and imaginary parts of the eigenvalue and eigenvector for the leading eigenpair (that with the largest in magnitude imaginary part). First we check if we actually managed to converge any eigenvalues at all.

```
if nconv == 0:
    import sys
    warning("Did not converge any eigenvalues")
    sys.exit(0)
```

If we did, we go ahead and extract them from the SLEPc eigenvalue solver:

```
vr, vi = petsc_a.getVecs()

lam = es.getEigenpair(0, vr, vi)
```

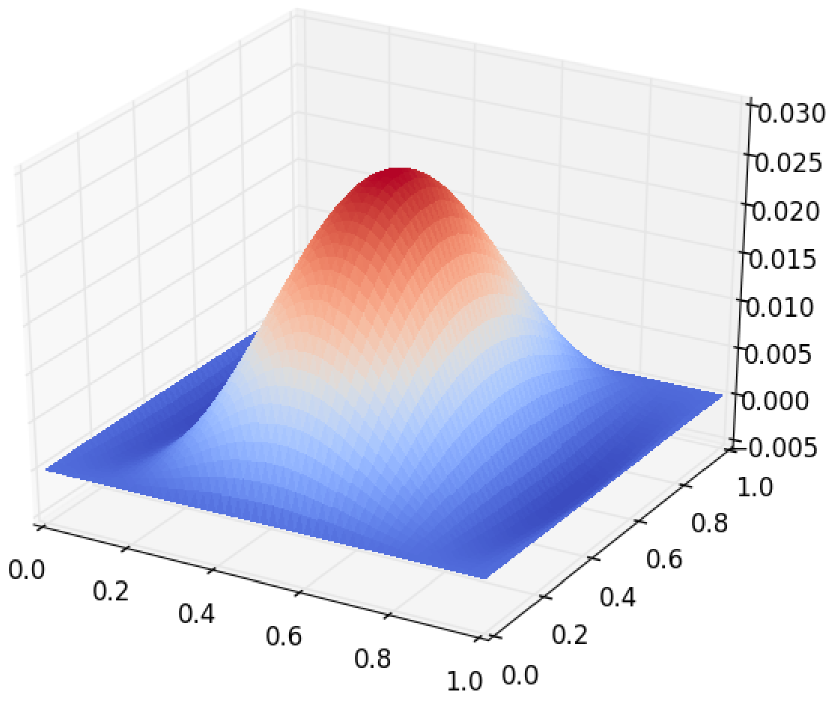
and we gather the final eigenfunctions

```
eigenmodes_real.vector()[:], eigenmodes_imag.vector()[:] = vr, vi
```

We can now list and show plots for the eigenvalues and eigenfunctions that were found.

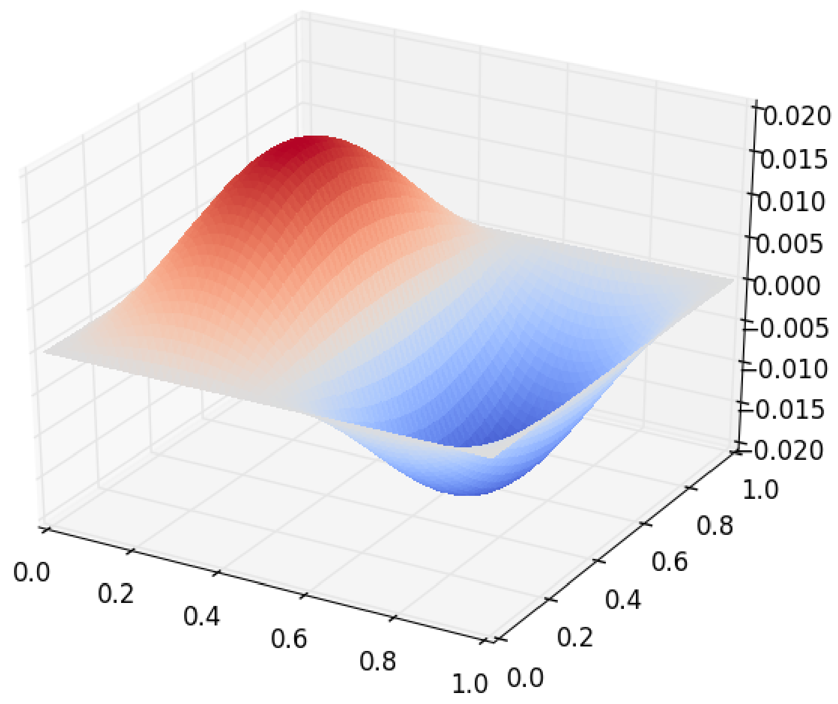
```
print("Leading eigenvalue is:", lam)

try:
    import matplotlib.pyplot as plt
    fig, axes = plt.subplots()
    colors = tripcolor(eigenmodes_real, axes=axes)
    fig.colorbar(colors)

    fig, axes = plt.subplots()
    colors = tripcolor(eigenmodes_imag, axes=axes)
    fig.colorbar(colors)
except ImportError:
    warning("Matplotlib not available, not plotting eigemodes")
```

Below is a plot of the spatial structure of the real part of one of the eigenmodes computed above.



Below is a plot of the spatial structure of the imaginary part of one of the eigenmodes computed above.

This demo can be found as a Python script in qgbasinmodes.py.



References

3.5 Wind-Driven Gyres: Quasi-Geostrophic Limit

Contributed by Christine Kaufhold and Francis Poulin.

Building on the previous two demos that used the Quasi-Geostrophic (QG) model for the *time-stepping* and *eigenvalue problem*, we now consider how to determine a wind-driven gyre solution that includes bottom drag and nonlinear advection. This is referred to as the Nonlinear Stommel Problem.

This is a classical problem going back to [Sto48]. Even though it is far too simple to describe the dynamics of the real oceans quantitatively, it does explain qualitatively why we have western intensification in the world's gyres. The curl of the wind stress adds vorticity into the gyres and the latitudinal variation in the Coriolis parameter causes a weak equatorward flow away from the boundaries (Sverdrup flow). It is because of the dissipation that arises near the boundaries that we must have western intensification. This was first shown by [Sto48] using simple bottom drag but it was only years later after [Mun50] did a similar calculation using lateral viscosity that people took the idea seriously.

After three quarters of a century we are still unable to parametrise the dissipative effects of the small scales so it is very difficult to get a good quantitative predictions as to the mean structure of the gyre that is generated. However, this demo aims to compute the structure of the oceanic gyre given particular parameters. The interested reader can read more about this in [Ped92] and [Val06]. In this tutorial we will consider the nonlinear Stommel problem.

3.5.1 Governing PDE: Stommel Problem

The nonlinear, one-layer, QG model equation that is driven by the winds above (say Q_{winds}), which is the vorticity of the winds that drive the ocean from above) is,

$$\partial_t q + \vec{u} \cdot \vec{\nabla} q + \beta v = -rq + Q_{\text{winds}}$$

with the Potential Vorticity (PV) and geostrophic velocities defined as

$$q = \nabla^2 \psi - F\psi, \quad \text{and} \quad \vec{u} = \hat{z} \times \vec{\nabla}\psi$$

where ψ is the stream-function, $\vec{u} = (u, v)$ is the velocity field, q is the PV, β is the latitudinal gradient of Coriolis parameter, and F is the rotational Froude number.

The non-conservative aspects of this model occur because of r, the strength of the bottom drag, and Q_{winds}, the vorticity of the winds. We pick the wind forcing as to generate a single gyre,

$$Q_{\text{winds}} = \tau \cos\left(\pi \left[\frac{y}{L_y} - \frac{1}{2}\right]\right)$$

where L_y is the length of our domain and τ is the strength of our wind forcing. By putting a 2 in front of the π we get a double gyre [Val06].

If we only look for steady solutions in time, we can ignore the time derivative term, and we get

$$(\vec{u} \cdot \vec{\nabla})\left(\nabla^2\psi - F\psi\right) + \beta \frac{\partial \psi}{\partial x} = -rq + Q_{\text{winds}}$$

We can write this out in one equation, which is the nonlinear Stommel problem:

$$\vec{u} \cdot \vec{\nabla}\left(\nabla^2\psi\right) + r(\nabla^2\psi - F\psi) + \beta \frac{\partial \psi}{\partial x} = Q_{\text{winds}}$$

Note that we dropped the $-F\psi$ term in the nonlinear advection because the streamfunction does not change following the flow, and therefore, we can neglect that term entirely.

3.5.2 Weak Formulation

To build the weak form of the problem in Firedrake we must find the weak form of this equation. We begin by multiplying this equation by a test function, ϕ, which is in the same space as the streamfunction, and then integrate over the domain Ω,

$$\int_\Omega \phi(\vec{u} \cdot \vec{\nabla})\nabla^2\psi \,\mathrm{d}x + r\phi(\nabla^2\psi - F\psi)\,\mathrm{d}x + \beta\phi\frac{\partial\psi}{\partial x}\,\mathrm{d}x = \int_\Omega \phi \cdot Q_{\text{winds}}\,\mathrm{d}x$$

The nonlinear term can be rewritten using the fact that the velocity is divergent free and then integrating by parts,

$$\int_\Omega \phi(\vec{u} \cdot \vec{\nabla})\nabla^2\psi = \int_\Omega \phi\vec{\nabla} \cdot \left(\vec{u}(\nabla^2\psi)\right) = -\int_\Omega (\vec{\nabla}\phi \cdot \vec{u})\nabla^2\psi\,\mathrm{d}x.$$

Note that because we have no normal flow boundary conditions the boundary contribution is zero. For the term with bottom drag we integrate by parts and use the fact that the streamfunction is zero on the walls

$$\int_\Omega r\phi\left(\vec{\nabla}^2\psi - F\psi\right)\,\mathrm{d}x = -r\int_\Omega \left(\vec{\nabla}\phi \cdot \vec{\nabla}\psi + F\phi\psi\right)\mathrm{d}x + r\int_{\partial\Omega} \phi \cdot \frac{\partial\psi}{\partial n}\,\mathrm{d}s$$

The boundary integral above vanishes because we are setting the streamfunction to be zero on the boundary.

Finally we can put the equation back together again to produce the weak form of our problem.

$$\int_{\Omega}\left(-(\vec{\nabla}\phi\cdot\vec{u})\vec{\nabla}^2\psi - r\left(\vec{\nabla}\phi\cdot\vec{\nabla}\psi + F\phi\psi\right) + \beta\phi\frac{\partial\psi}{\partial x}\right)\mathrm{d}x = \int_{\Omega}\phi\cdot Q_{\text{winds}}\,\mathrm{d}x$$

The above problem is the weak form of the nonlinear Stommel problem. The linear term arises from neglecting the nonlinear advection, and can easily be obtained by neglecting the first term on the left hand side.

3.5.3 Defining the Problem

Now that we know the weak form we are now ready to solve this using Firedrake!

First, we import the Firedrake, PETSc, NumPy and UFL packages,

```
from firedrake import *
from firedrake.petsc import PETSc
import numpy as np
import ufl
```

Next, we can define the geometry of our domain. In this example, we will be using a square of length one with 50 cells.

```
n0 = 50             # Spatial resolution
Ly = 1.0            # Meridonal length
Lx = 1.0            # Zonal length
mesh = RectangleMesh(n0, n0, Lx, Ly, reorder = None)
```

We can then define the Function Space within which the solution of the streamfunction will reside.

```
Vcg = FunctionSpace(mesh, 'CG', 3) # CG elements for Streamfunction
```

We will also impose no-normal flow strongly to ensure that the boundary condition $\psi = 0$ will be met,

```
bc = DirichletBC(Vcg, 0.0, 'on_boundary')
```

Now we will define all the parameters we are using in this tutorial.

```
beta = Constant('1.0')       # Beta parameter
F = Constant('1.0')          # Burger number
r = Constant('0.2')          # Bottom drag
tau = Constant('0.001')      # Wind Forcing
x = SpatialCoordinate(mesh)
Qwinds = Function(Vcg).interpolate(-tau * cos(pi * (x[1]/Ly - 0.5)))
```

We can now define the Test Function and the Trial Function of this problem, both must be in the same function space:

```
phi, psi = TestFunction(Vcg), TrialFunction(Vcg)
```

We must define functions that will store our linear and nonlinear solutions. In order to solve the nonlinear problem, we use the linear solution as a guess for the nonlinear problem.

```
psi_lin = Function(Vcg, name='Linear Streamfunction')
psi_non = Function(Vcg, name='Nonlinear Streamfunction')
```

We can finally write down the linear Stommel equation in its weak form. We will use the solution to this as the input for the nonlinear Stommel equation.

```
a = - r * inner(grad(psi), grad(phi)) * dx - F * psi * phi * dx + beta * psi.
↪dx(0) * phi * dx
L = Qwinds * phi * dx
```

We set-up an elliptic solver for this problem, and solve for the linear streamfunction,

```
linear_problem = LinearVariationalProblem(a, L, psi_lin, bcs=bc)
linear_solver = LinearVariationalSolver(linear_problem,
                                        solver_parameters= {'ksp_type':
↪'preonly',
                                                            'pc_type': 'lu'})
linear_solver.solve()
```

We will employ the solution to the linear problem as the initial guess for the nonlinear one:

```
psi_non.assign(psi_lin)
```

And now we can define the weak form of the nonlinear problem. Note that the problem is stated in residual form so there is no trial function.

```
G = - inner(grad(phi), perp(grad(psi_non))) * div(grad(psi_non)) * dx \
    -r * inner(grad(psi_non), grad(phi)) * dx - F * psi_non * phi * dx \
    + beta * psi_non.dx(0) * phi * dx \
    - Qwinds * phi * dx
```

We solve for the nonlinear streamfunction now by setting up another elliptic solver,

```
nonlinear_problem = NonlinearVariationalProblem(G, psi_non, bcs=bc)
nonlinear_solver = NonlinearVariationalSolver(nonlinear_problem,
                                              solver_parameters= {'snes_type
↪': 'newtonls',
                                                                  'ksp_type':
↪'preonly',
                                                                  'pc_type':
↪'lu'})
nonlinear_solver.solve()
```

Now that we have the full solution to the nonlinear Stommel problem, we can plot it using the *tripcolor* function

```
try:
    import matplotlib.pyplot as plt
except:
    warning("Matplotlib not imported")

try:
    fig, axes = plt.subplots()
    colors = tripcolor(psi_non, axes=axes)
    fig.colorbar(colors)
except Exception as e:
    warning("Cannot plot figure. Error msg '%s'" % e)

try:
    plt.show()
except Exception as e:
    warning("Cannot show figure. Error msg '%s'" % e)

file = File('Nonlinear Streamfunction.pvd')
file.write(psi_non)
```

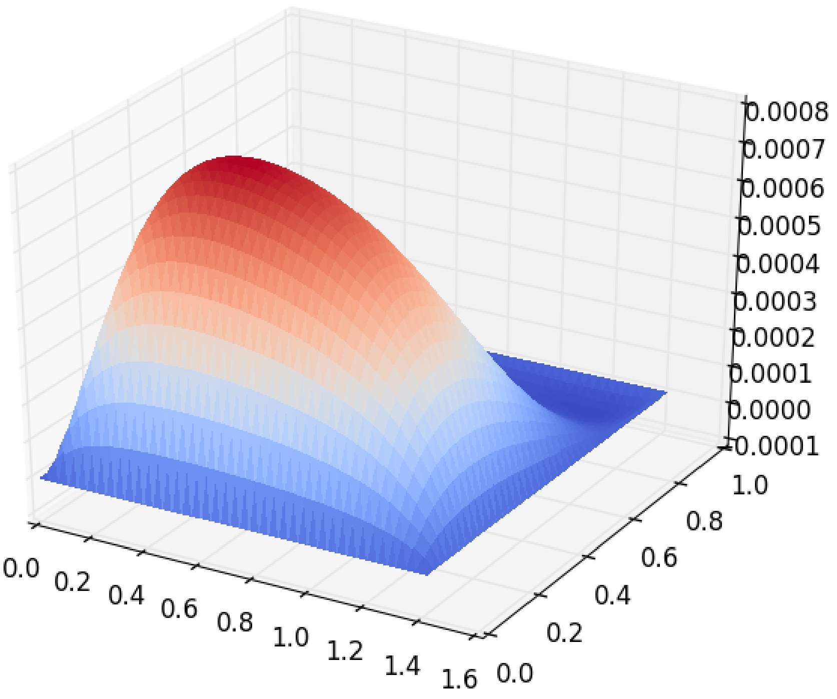
We can also see the difference between the linear solution and the nonlinear solution. We do this by defining a weak form. (Note: other approaches may be possible.)

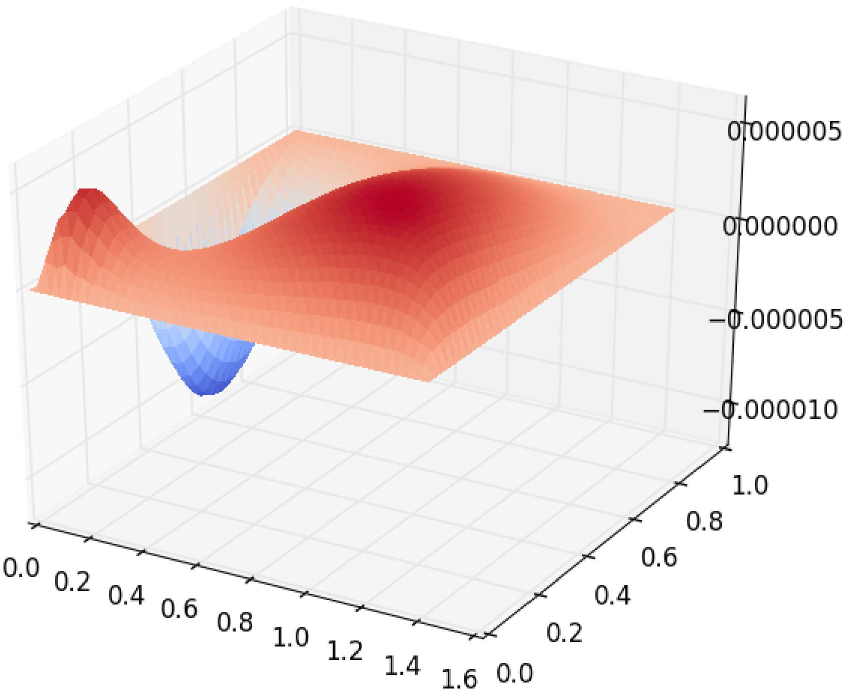
```
tf, difference = TestFunction(Vcg), TrialFunction(Vcg)

difference = assemble(psi_lin - psi_non)

try:
    fig, axes = plt.subplots()
    colors = tripcolor(difference, axes=axes)
    fig.colorbar(colors)
except Exception as e:
    warning("Cannot plot figure. Error msg '%s'" % e)

try:
    plt.show()
except Exception as e:
    warning("Cannot show figure. Error msg '%s'" % e)

file = File('Difference between Linear and Nonlinear Streamfunction.pvd')
file.write(difference)
```

Below is a plot of the linear solution to the QG wind-driven Stommel gyre.

Below is a plot of the difference between the linear and nonlinear solutions to the QG wind-driven Stommel gyre.

This demo can be found as a Python script in qg_winddrivengyre.py.


0.0008
0.0007
0.0006
0.0005
0.0004
0.0003
0.0002
0.0001
0.0000
−0.0001
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0.0
0.2
0.4
0.6
0.8
1.0


0.000005
0.000000
−0.000005
−0.000010
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1.4
1.6
0.0
0.2
0.4
0.6
0.8
1.0

References

3.6 Preconditioning saddle-point systems

3.6.1 Introduction

In this demo, we will discuss strategies for solving saddle-point systems using the mixed formulation of the Poisson equation introduced *previously* as a concrete example. Such systems are somewhat tricky to precondition effectively, modern approaches typically use block-factorisations. We will encounter a number of methods in this tutorial. For many details and background on solution methods for saddle point systems, [BGL05] is a nice review. [ESW14] is an excellent text with a strong focus on applications in fluid dynamics.

We start by repeating the formulation of the problem. Starting from the primal form of the Poisson equation, $\nabla^2 u = -f$, we introduce a vector-valued flux, $\sigma = \nabla u$. The problem then becomes to find u and σ in some domain Ω satisfying

$$\begin{aligned} \sigma - \nabla u &= 0 & &\text{on } \Omega \\ \nabla \cdot \sigma &= -f & &\text{on } \Omega \\ u &= u_0 & &\text{on } \Gamma_D \\ \sigma \cdot n &= g & &\text{on } \Gamma_N \end{aligned}$$

for some specified function f. We now seek $(u, \sigma) \in V \times \Sigma$ such that

$$\begin{aligned} \int_\Omega \sigma \cdot \tau + (\nabla \cdot \tau)\, u \,\mathrm{d}x &= \int_\Gamma (\tau \cdot n)\, u \,\mathrm{d}s & &\forall\, \tau \in \Sigma, \\ \int_\Omega (\nabla \cdot \sigma)\, v \,\mathrm{d}x &= -\int_\Omega f\, v \,\mathrm{d}x & &\forall\, v \in V. \end{aligned}$$

A stable choice of discrete spaces for this problem is to pick $\Sigma_h \subset \Sigma$ to be the lowest order Raviart-Thomas space, and $V_h \subset V$ to be the piecewise constants, although this is *not the only choice*. For ease of exposition we choose the domain to be the unit square, and enforce homogeneous Dirichlet conditions on all walls. The forcing term is chosen to be random.

Globally coupled elliptic problems, such as the Poisson problem, require effective preconditioning to attain *mesh independent* convergence. By this we mean that the number of iterations of the linear solver does not grow when the mesh is refined. In this demo, we will study various ways to achieve this in Firedrake.

As ever, we begin by importing the Firedrake module:

```
from firedrake import *
```

3.6.2 Building the problem

Rather than defining a mesh and function spaces straight away, since we wish to consider the effect that mesh refinement has on the performance of the solver, we instead define a Python function which builds the problem we wish to solve. This takes as arguments the size of the mesh, the solver parameters we wish to apply, an optional parameter specifying a "preconditioning" operator to apply, and a final optional argument specifying whether the block system should be assembled as a single "monolithic" matrix or a 2×2 block of smaller matrices.

```
def build_problem(mesh_size, parameters, aP=None, block_matrix=False):
    mesh = UnitSquareMesh(2 ** mesh_size, 2 ** mesh_size)

    Sigma = FunctionSpace(mesh, "RT", 1)
    V = FunctionSpace(mesh, "DG", 0)
    W = Sigma * V
```

Having built the function spaces, we can now proceed to defining the problem. We will need some trial and test functions for the spaces:

```
#
    sigma, u = TrialFunctions(W)
    tau, v = TestFunctions(W)
```

along with a function to hold the forcing term, living in the discontinuous space.

```
#
    f = Function(V)
```

To initialise this function to a random value we access its *`Vector`* form and use numpy to set the values:

```
#
    import numpy as np
    fvector = f.vector()
    fvector.set_local(np.random.uniform(size=fvector.local_size()))
```

Note that the homogeneous Dirichlet conditions in the primal formulation turn into homogeneous Neumann conditions on the dual variable and we therefore drop the surface integral terms in the variational formulation (they are identically zero). As a result, the specification of the variational problem is particularly simple:

```
#
    a = dot(sigma, tau)*dx + div(tau)*u*dx + div(sigma)*v*dx
    L = -f*v*dx
```

Now we treat the mysterious optional `aP` argument. When solving a linear system, Firedrake allows specifying that the problem should be preconditioned with an operator different to the operator defining the problem to be solved. We will use this functionality in a number of cases later. The `aP` function will take one argument, the *`FunctionSpace`* defining the space, and return a bilinear form suitable for assembling as an operator. Obviously we only do so if `aP` is provided.

```
#
    if aP is not None:
        aP = aP(W)
```

Now we have all the pieces to build our linear system. We will return a *`LinearSolver`* object from this function, so we preassemble the operators to build it. It is here that we must specify whether we want a monolithic matrix or not, by setting the matrix type parameter to *`assemble()`*.

```
#
    if block_matrix:
        mat_type = 'nest'
    else:
        mat_type = 'aij'

    if aP is not None:
        P = assemble(aP, mat_type=mat_type)
    else:
        P = None

    w = Function(W)
    vpb = LinearVariationalProblem(a, L, w, aP=aP)
    solver =  LinearVariationalSolver(vpb, solver_parameters=parameters)
```

Finally, we return solver and solution function as a tuple.

```
#
    return solver, w
```

With these preliminaries out of the way, we can now move on to solution strategies, in particular, preconditioner options.

3.6.3 Preconditioner choices

A naive approach

To illustrate the problem, we first attempt to solve the problem on a sequence of finer and finer meshes preconditioning the problem with zero-fill incomplete LU factorisation. Configuration of the solver is carried out by providing appropriate parameters when constructing the *LinearSolver* object through the `solver_parameters` keyword argument which should be a `dict` of parameters. These parameters are passed directly to PETSc, and their form is described in more detail in *Solving PDEs*. For this problem, we use GMRES with a restart length of 100,

```
parameters = {
    "ksp_type": "gmres",
    "ksp_gmres_restart": 100,
```

solve to a relative tolerance of 1e-8,

```
#
    "ksp_rtol": 1e-8,
```

and precondition with ILU(0).

```
#
    "pc_type": "ilu",
    }
```

We now loop over a range of mesh sizes, assembling the system and solving it

```
print("Naive preconditioning")
for n in range(8):
    solver, w = build_problem(n, parameters, block_matrix=False)
    solver.solve()
```

Finally, at each mesh size, we print out the number of cells in the mesh and the number of iterations the solver took to converge

```
#
    print(w.function_space().mesh().num_cells(), solver.snes.ksp.
↪getIterationNumber())
```

The resulting convergence is unimpressive:

Mesh elements	GMRES iterations
2	2
8	12
32	27
128	54
512	111
2048	255
8192	717
32768	2930

Were this a primal Poisson problem, we would be able to use a standard algebraic multigrid preconditioner, such as hypre. However, this dual formulation is slightly more complicated.

Schur complement approaches

A better approach is to use a Schur complement preconditioner, described in *Preconditioning mixed finite element systems*. The system we are trying to solve is conceptually a 2×2 block matrix.

$$\begin{pmatrix} A & B \\ C & 0 \end{pmatrix}$$

which admits a factorisation

$$\begin{pmatrix} I & 0 \\ CA^{-1} & I \end{pmatrix} \begin{pmatrix} A & 0 \\ 0 & S \end{pmatrix} \begin{pmatrix} I & A^{-1}B \\ 0 & I \end{pmatrix},$$

with the *Schur complement* $S = -CA^{-1}B$. The inverse of the operator can be therefore be written as

$$P = \begin{pmatrix} I & -A^{-1}B \\ 0 & I \end{pmatrix} \begin{pmatrix} A^{-1} & 0 \\ 0 & S^{-1} \end{pmatrix} \begin{pmatrix} I & 0 \\ -CA^{-1} & I \end{pmatrix}.$$

An algorithmically optimal solution

If we can find a good way of approximating P then we can use that to precondition our original problem. This boils down to finding good approximations to A^{-1} and S^{-1}. For our problem, A is just a mass matrix and so we can invert it well with a cheap method: either a few iterations of jacobi or ILU(0) are fine. The troublesome term is S which is spectrally a Laplacian, but dense (since A^{-1} is dense). However, before we worry too much about this, let us just try using a Schur complement preconditioner. This simple setup can be driven using only solver options.

Note that we will exactly invert the inner blocks for A^{-1} and S^{-1} using Krylov methods. We therefore need to use *flexible* GMRES as our outer solver, since the use of inner Krylov methods in our preconditioner makes the application of the preconditioner nonlinear. This time we use the default restart length of 30, but solve to a relative tolerance of 1e-8:

```
parameters = {
    "ksp_type": "fgmres",
    "ksp_rtol": 1e-8,
```

this time we want a `fieldsplit` preconditioner.

```
#
    "pc_type": "fieldsplit",
    "pc_fieldsplit_type": "schur",
    "pc_fieldsplit_schur_fact_type": "full",
```

If we use this preconditioner and invert all the blocks exactly, then the preconditioned operator will have at most three distinct eigenvalues [MGW00] and hence GMRES should converge in at most three iterations. To try this, we start out by exactly inverting A and S to check the convergence.

```
    "fieldsplit_0_ksp_type": "cg",
    "fieldsplit_0_pc_type": "ilu",
    "fieldsplit_0_ksp_rtol": 1e-12,
    "fieldsplit_1_ksp_type": "cg",
    "fieldsplit_1_pc_type": "none",
    "fieldsplit_1_ksp_rtol": 1e-12,
}
```

Let's go ahead and run this. Note that for this problem, we're applying the action of blocks, so we can use a block matrix format.

```
print("Exact full Schur complement")
for n in range(8):
    solver, w = build_problem(n, parameters, block_matrix=True)
    solver.solve()
    print(w.function_space().mesh().num_cells(), solver.snes.ksp.
↪getIterationNumber())
```

The resulting convergence is algorithmically good, however, the larger problems still take a long time.

Mesh elements	fGMRES iterations
2	1
8	1
32	1
128	1
512	1
2048	1
8192	1
32768	1

We can improve things by building a matrix used to precondition the inversion of the Schur complement. Note how we're currently not using any preconditioning, and so the inner solver struggles (this can be observed by additionally running with the parameter `"fieldsplit_1_ksp_converged_reason": True`.

As we increase the number of mesh elements, the solver inverting S takes more and more iterations, which means that we take longer and longer to solve the problem as the mesh is refined.

Mesh elements	CG iterations on S
2	2
8	7
32	32
128	73
512	149
2048	289
8192	553
32768	1143

Approximating the Schur complement

Fortunately, PETSc gives us some options to try here. For our problem a diagonal "mass-lumping" of the velocity mass matrix gives a good approximation to A^{-1}. Under these circumstances $S_p = -C\mathrm{diag}(A)^{-1}B$ is spectrally close to S, but sparse, and can be used to precondition the solver inverting S. To do this, we need some additional parameters. First we repeat those that remain unchanged

```
parameters = {
    "ksp_type": "fgmres",
    "ksp_rtol": 1e-8,
    "pc_type": "fieldsplit",
    "pc_fieldsplit_type": "schur",
    "pc_fieldsplit_schur_fact_type": "full",
    "fieldsplit_0_ksp_type": "cg",
    "fieldsplit_0_pc_type": "ilu",
    "fieldsplit_0_ksp_rtol": 1e-12,
```

(continues on next page)

(continued from previous page)

```
    "fieldsplit_1_ksp_type": "cg",
    "fieldsplit_1_ksp_rtol": 1e-12,
```

Now we tell PETSc to construct S_p using the diagonal of A, and to precondition the resulting linear system using algebraic multigrid from the hypre suite.

```
    "pc_fieldsplit_schur_precondition": "selfp",
    "fieldsplit_1_pc_type": "hypre"
}
```

Note: For this set of options to work, you will have needed to build PETSc with support for hypre (for example, by specifying `--download-hypre` when configuring).

Let's see what happens.

```
print("Schur complement with S_p")
for n in range(8):
    solver, w = build_problem(n, parameters, block_matrix=True)
    solver.solve()
    print(w.function_space().mesh().num_cells(), solver.snes.ksp.
↪getIterationNumber())
```

This is much better, the problem takes much less time to solve and when observing the iteration counts for inverting S we can see why.

Mesh elements	CG iterations on S
2	2
8	8
32	17
128	18
512	19
2048	19
8192	19
32768	19

We can now think about backing off the accuracy of the inner solves. Effectively computing a worse approximation to P that we hope is faster, despite taking more GMRES iterations. Additionally we can try dropping some terms in the factorisation of P, by adjusting `pc_fieldsplit_schur_fact_type` from `full` to one of `upper`, `lower`, or `diag` we make the preconditioner slightly worse, but gain because we require fewer applications of A^{-1}. For our problem where computing A^{-1} is cheap, this is not a great problem, however for many fluids problems A^{-1} is expensive and it pays to experiment.

For example, we might wish to try a full factorisation, but approximate A^{-1} by a single application of ILU(0) and S^{-1} by a single multigrid V-cycle on S_p. To do this, we use the following set of parameters.

```
parameters = {
    "ksp_type": "gmres",
    "ksp_rtol": 1e-8,
    "pc_type": "fieldsplit",
    "pc_fieldsplit_type": "schur",
    "pc_fieldsplit_schur_fact_type": "full",
    "fieldsplit_0_ksp_type": "preonly",
    "fieldsplit_0_pc_type": "ilu",
    "fieldsplit_1_ksp_type": "preonly",
    "pc_fieldsplit_schur_precondition": "selfp",
    "fieldsplit_1_pc_type": "hypre"
}
```

Note how we can switch back to GMRES here, our inner solves are linear and so we no longer need a flexible Krylov method.

```
print("Schur complement with S_p and inexact inner inverses")
for n in range(8):
    solver, w = build_problem(n, parameters, block_matrix=True)
    solver.solve()
    print(w.function_space().mesh().num_cells(), solver.snes.ksp.
↪getIterationNumber())
```

This results in the following GMRES iteration counts

Mesh elements	GMRES iterations
2	2
8	9
32	11
128	13
512	13
2048	12
8192	12
32768	12

and the solves take only a few seconds.

Providing the Schur complement approximation

Instead of asking PETSc to build an approximation to S which we then use to solve the problem, we can provide one ourselves. Recall that S is spectrally a Laplacian only in a discontinuous space. A natural choice is therefore to use an interior penalty DG formulation for the Laplacian term on the block of the scalar variable. We can provide it as an *AuxiliaryOperatorPC* via a python preconditioner.

```
class DGLaplacian(AuxiliaryOperatorPC):
    def form(self, pc, u, v):
        W = u.function_space()
```

(continues on next page)

(continued from previous page)

```
        n = FacetNormal(W.mesh())
        alpha = Constant(4.0)
        gamma = Constant(8.0)
        h = CellSize(W.mesh())
        h_avg = (h('+') + h('-'))/2
        a_dg = -(inner(grad(u), grad(v))*dx \
            - inner(jump(u, n), avg(grad(v)))*dS \
            - inner(avg(grad(u)), jump(v, n), )*dS \
            + alpha/h_avg * inner(jump(u, n), jump(v, n))*dS \
            - inner(u*n, grad(v))*ds \
            - inner(grad(u), v*n)*ds \
            + (gamma/h)*inner(u, v)*ds)
        bcs = None
        return (a_dg, bcs)

parameters = {
    "ksp_type": "gmres",
    "ksp_rtol": 1e-8,
    "pc_type": "fieldsplit",
    "pc_fieldsplit_type": "schur",
    "pc_fieldsplit_schur_fact_type": "full",
    "fieldsplit_0_ksp_type": "preonly",
    "fieldsplit_0_pc_type": "ilu",
    "fieldsplit_1_ksp_type": "preonly",
    "fieldsplit_1_pc_type": "python",
    "fieldsplit_1_pc_python_type": __name__+ ".DGLaplacian",
    "fieldsplit_1_aux_pc_type": "hypre"
}

print("DG approximation for S_p")
for n in range(8):
    solver, w = build_problem(n, parameters, aP=None, block_matrix=False)
    solver.solve()
    print(w.function_space().mesh().num_cells(), solver.snes.ksp.
↪getIterationNumber())
```

This actually results in slightly worse convergence than the diagonal approximation we used above.

Mesh elements	GMRES iterations
2	2
8	9
32	12
128	13
512	14
2048	13
8192	13
32768	13

Block diagonal preconditioners

An alternate approach to using a Schur complement is to use a block-diagonal preconditioner. To do this, we note that the mesh-dependent ill conditioning of linear operators comes from working in the wrong norm. To convert into working in the correct norm, we can precondition our problem using the *Riesz map* for the spaces. For details on the mathematics behind this approach see for example [Kir10].

We are working in a space $W \subset H(\text{div}) \times L^2$, and as such, the appropriate Riesz map is just $H(\text{div})$ inner product in Σ and the L^2 inner product in V. As was the case for the DG Laplacian, we do this by providing a function that constructs this operator to our `build_problem` function.

```
def riesz(W):
    sigma, u = TrialFunctions(W)
    tau, v = TestFunctions(W)

    return (dot(sigma, tau) + div(sigma)*div(tau) + u*v)*dx
```

Now we set up the solver parameters. We will still use a `fieldsplit` preconditioner, but this time it will be additive, rather than a Schur complement.

```
parameters = {
    "ksp_type": "gmres",
    "ksp_rtol": 1e-8,
    "pc_type": "fieldsplit",
    "pc_fieldsplit_type": "additive",
```

Now we choose how to invert the two blocks. The second block is easy, it is just a mass matrix in a discontinuous space and is therefore inverted exactly using a single application of zero-fill ILU.

```
#
    "fieldsplit_1_ksp_type": "preonly",
    "fieldsplit_1_pc_type": "ilu",
```

The $H(\text{div})$ inner product is the tricky part. For a first attempt, we will invert it with a direct solver. This is a reasonable option up to a few tens of thousands of degrees of freedom.

```
#
    "fieldsplit_0_ksp_type": "preonly",
    "fieldsplit_0_pc_type": "lu",
}
```

Note: For larger problems, you will probably need to use a sparse direct solver such as MUMPS, which may be selected by additionally specifying `"fieldsplit_0_pc_factor_mat_solver_type": "mumps"`.

To use MUMPS you will need to have configured PETSc appropriately (using at the very least `--download-mumps`).

Let's see what the iteration count looks like now.

```
print("Riesz-map preconditioner")
for n in range(8):
    solver, w = build_problem(n, parameters, aP=riesz, block_matrix=True)
    solver.solve()
    print(w.function_space().mesh().num_cells(), solver.snes.ksp.
↪getIterationNumber())
```

Mesh elements	GMRES iterations
2	3
8	5
32	5
128	5
512	5
2048	5
8192	5
32768	5

Firedrake provides some facility to solve the $H(\mathrm{div})$ Riesz map in a scalable way. In particular either by employing a geometric multigrid method with overlapping Schwarz smoothers (using *PatchPC*), or using the algebraic approach of [HX07] provided by Hypre's "auxiliary space" preconditioners `AMS` and `ADS`. See the separate manual page on *Preconditioning infrastructure*.

A runnable python script version of this demo is available here.

References

3.7 Camassa-Holm equation

This tutorial was contributed by Colin Cotter.

The Camassa Holm equation [CH93] is an integrable 1+1 PDE which may be written in the form

$$m_t + mu_x + (mu)_x = 0, \quad u - \alpha^2 u_{xx} = m,$$

solved in the interval $[a, b]$ either with periodic boundary conditions or with boundary conditions *u(a)=u(b)=0*; $\alpha > 0$ is a constant that sets a lengthscale for the solution. The solution is entirely composed of peaked solitons corresponding to Dirac delta functions in m. Further, the solution has a conserved energy, given by

$$\int_a^b \frac{1}{2}u^2 + \frac{\alpha^2}{2}u_x^2 \,\mathrm{d}x.$$

In this example we will concentrate on the periodic boundary conditions case.

A weak form of these equations is given by

$$\int pm_t + pmu_x - p_x mu \,\mathrm{d}x = 0, \quad \forall p,$$
$$\int qu + \alpha^2 q_x u_x - qm \,\mathrm{d}x = 0, \quad \forall q.$$

Energy conservation then follows from substituting the second equation into the first, and then setting $p = u$,

$$\begin{aligned}
\dot{E} &= \frac{\mathrm{d}}{\mathrm{d}t}\int_a^b \frac{1}{2}u^2 + \frac{\alpha^2}{2}u_x^2 \,\mathrm{d}x \\
&= \int_a^b uu_t + \alpha^2 u_x u_{xt}\,\mathrm{d}x, \\
&= \int_a^b um_t\,\mathrm{d}x, \\
&= \int_a^b -umu_x + u_x mu\,\mathrm{d}x = 0.
\end{aligned}$$

If we choose the same continuous finite element spaces for m and u then this proof immediately extends to the spatial discretisation, as noted by [Mat10]. Further, it is a property of the implicit midpoint rule time discretisation that any quadratic conserved quantities of an ODE are also conserved by the time discretisation (see [Ise09], for example). Hence, the fully discrete scheme,

$$\begin{aligned}
\int p(m^{n+1} - m^n) + \Delta t(pm^{n+1/2}u_x^{n+1/2} - p_x m^{n+1/2}u^{n+1/2})\,\mathrm{d}x = 0, &\quad \forall p \in V, \\
\int qu^{n+1/2} + \alpha^2 q_x u_x^{n+1/2} - qm^{n+1/2}\,\mathrm{d}x = 0, &\quad \forall q \in V,
\end{aligned}$$

where $u^{n+1/2} = (u^{n+1} + u^n)/2$, $m^{n+1/2} = (m^{n+1} + m^n)/2$, conserves the energy exactly. This is a useful property since the energy is the square of the H^1 norm, which guarantees regularity of the numerical solution.

As usual, to implement this problem, we start by importing the Firedrake namespace.

```
from firedrake import *
```

To visualise the output, we also need to import matplotlib.pyplot to display the visual output

```
try:
  import matplotlib.pyplot as plt
except:
  warning("Matplotlib not imported")
```

We then set the parameters for the scheme.

```
alpha = 1.0
alphasq = Constant(alpha**2)
dt = 0.1
Dt = Constant(dt)
```

These are set with type `Constant` so that the values can be changed without needing to regenerate code.

We use a `periodic mesh` of width 40 with 100 cells,

```
n = 100
mesh = PeriodicIntervalMesh(n, 40.0)
```

and build a `mixed function space` for the two variables.

```
V = FunctionSpace(mesh, "CG", 1)
W = MixedFunctionSpace((V, V))
```

We construct a *Function* to store the two variables at time level `n`, and *split()* it so that we can interpolate the initial condition into the two components.

```
w0 = Function(W)
m0, u0 = w0.split()
```

Then we interpolate the initial condition,

$$u^0 = 0.2\text{sech}(x - 403/15) + 0.5\text{sech}(x - 203/15),$$

into u,

```
x, = SpatialCoordinate(mesh)
u0.interpolate(0.2*2/(exp(x-403./15.) + exp(-x+403./15.))
               + 0.5*2/(exp(x-203./15.)+exp(-x+203./15.)))
```

before solving for the initial condition for `m`. This is done by setting up the linear problem and solving it (here we use a direct solver since the problem is one dimensional).

```
p = TestFunction(V)
m = TrialFunction(V)

am = p*m*dx
Lm = (p*u0 + alphasq*p.dx(0)*u0.dx(0))*dx

solve(am == Lm, m0, solver_parameters={
      'ksp_type': 'preonly',
      'pc_type': 'lu'
      }
   )
```

Next we build the weak form of the timestepping algorithm. This is expressed as a mixed nonlinear problem, which must be written as a bilinear form that is a function of the output *Function* `w1`.

```
p, q = TestFunctions(W)

w1 = Function(W)
w1.assign(w0)
m1, u1 = split(w1)
m0, u0 = split(w0)
```

Note the use of `split(w1)` here, which splits up a *Function* so that it may be inserted into a UFL expression.

```
mh = 0.5*(m1 + m0)
uh = 0.5*(u1 + u0)
```

(continues on next page)

(continued from previous page)

```
L = (
(q*u1 + alphasq*q.dx(0)*u1.dx(0) - q*m1)*dx +
(p*(m1-m0) + Dt*(p*uh.dx(0)*mh -p.dx(0)*uh*mh))*dx
)
```

Since we are in one dimension, we use a direct solver for the linear system within the Newton algorithm. To do this, we assemble a monolithic rather than blocked system.

```
uprob = NonlinearVariationalProblem(L, w1)
usolver = NonlinearVariationalSolver(uprob, solver_parameters=
   {'mat_type': 'aij',
    'ksp_type': 'preonly',
    'pc_type': 'lu'})
```

Next we use the other form of *split()*, `w0.split()`, which is the way to split up a Function in order to access its data e.g. for output.

```
m0, u0 = w0.split()
m1, u1 = w1.split()
```

We choose a final time, and initialise a *File* object for storing `u`. as well as an array for storing the function to be visualised:

```
T = 100.0
ufile = File('u.pvd')
t = 0.0
ufile.write(u1, time=t)
all_us = []
```

We also initialise a dump counter so we only dump every 10 timesteps.

```
ndump = 10
dumpn = 0
```

Now we enter the timeloop.

```
while (t < T - 0.5*dt):
   t += dt
```

The energy can be computed and checked.

```
#
   E = assemble((u0*u0 + alphasq*u0.dx(0)*u0.dx(0))*dx)
   print("t = ", t, "E = ", E)
```

To implement the timestepping algorithm, we just call the solver, and assign `w1` to `w0`.

```
#
   usolver.solve()
   w0.assign(w1)
```

Finally, we check if it is time to dump the data. The function will be appended to the array of functions to be plotted later:

```
#
   dumpn += 1
   if dumpn == ndump:
      dumpn -= ndump
      ufile.write(u1, time=t)
      all_us.append(Function(u1))
```

This solution leads to emergent peakons (peaked solitons); the left peakon is travelling faster than the right peakon, so they collide and momentum is transferred to the right peakon.

At last, we call the function `plot` on the final value to visualize it:

```
try:
  fig, axes = plt.subplots()
  plot(all_us[-1], axes=axes)
except Exception as e:
  warning("Cannot plot figure. Error msg: '%s'" % e)
```

And finally show the figure:

```
try:
  plt.show()
except Exception as e:
  warning("Cannot show figure. Error msg: '%s'" % e)
```

Images of the solution at shown below.

A python script version of this demo can be found here.

References

3.8 The Monge-Ampère equation

This tutorial was contributed by Colin Cotter, based on code from Andrew McRae and Lawrence Mitchell.

The Monge-Ampère equation provides the solution to the optimal transportation problem between two measures. Here, we consider the case where the target measure is the usual Lebesgue measure, and the template measure is $f(x)d^nx$, both defined on the same domain. Then, in two dimensions, the optimal transportation plan is given by

$$(x, y) \mapsto (x, y) + \nabla u,$$

where u satisfies the the Monge-Ampère equation

$$\det\left(I + D^2 u\right) = f,$$

where I is the identity matrix, and D^2 is the Hessian matrix of second derivatives, subject to the boundary conditions $\frac{\partial u}{\partial n} = 0$.

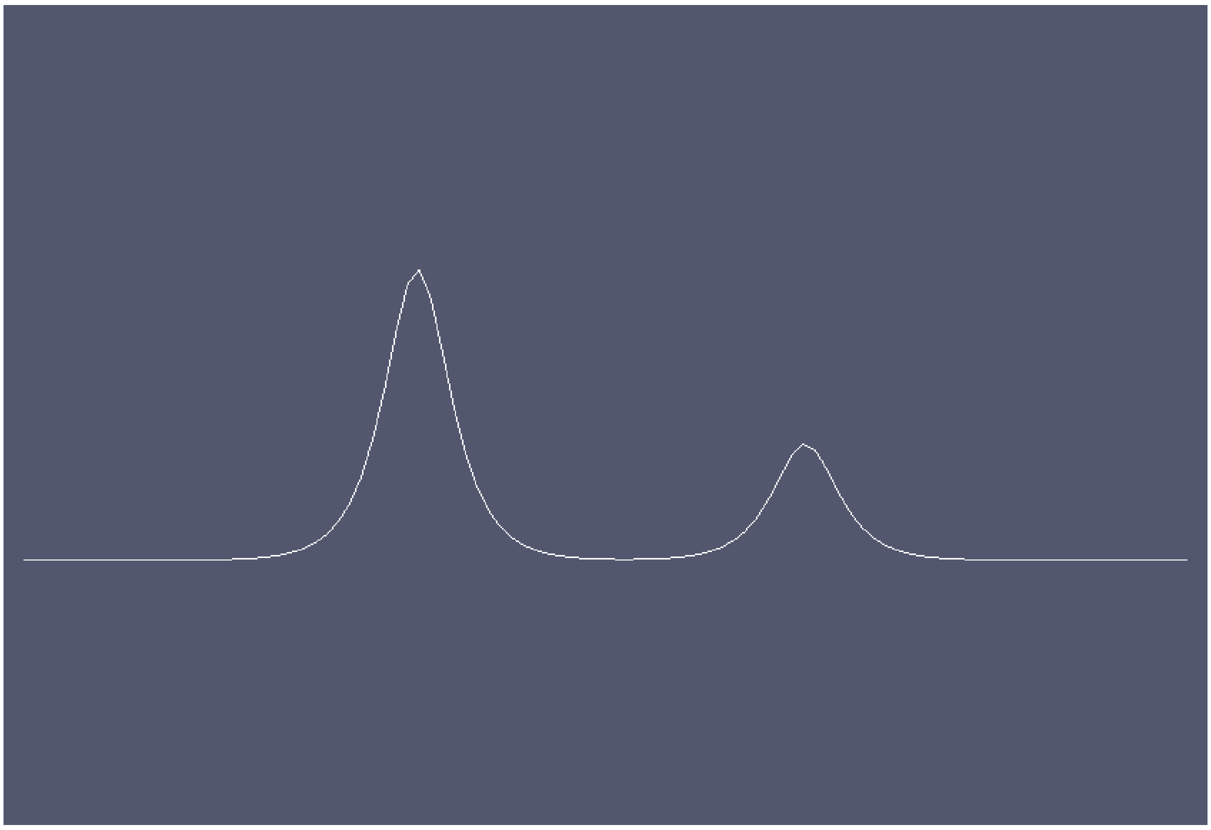


Figure1: Solution at $t = 0$.

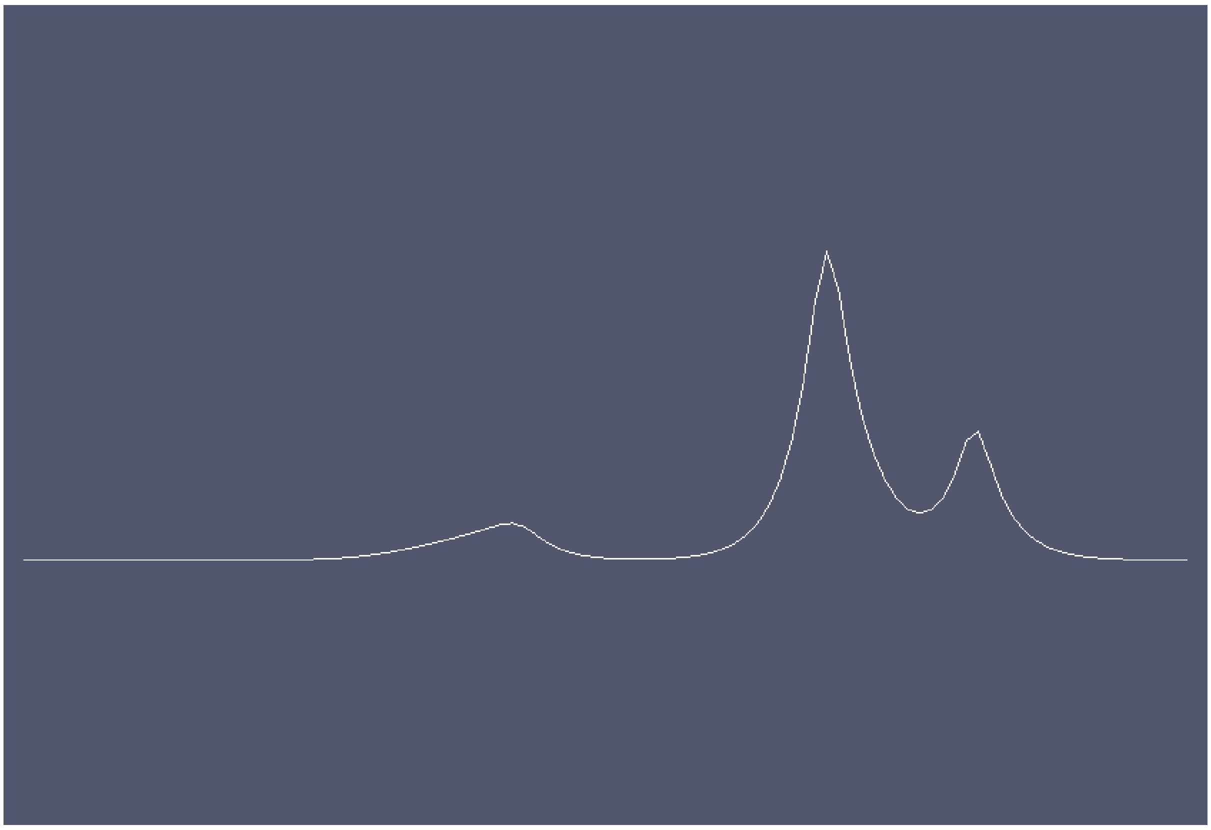


Figure2: Solution at $t = 2.5$.

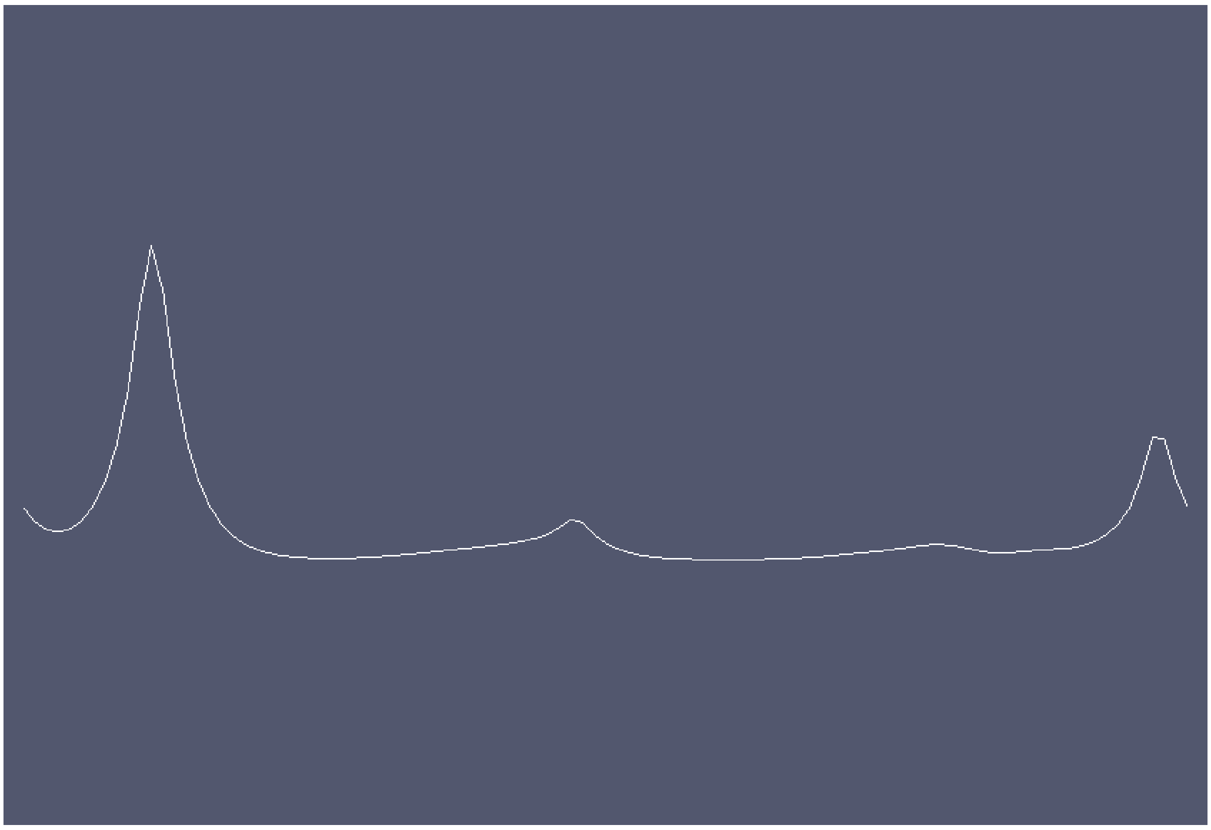


Figure3: Solution at $t = 5.3$.

Here we follow the approach of [LP13], namely to use the mixed formulation

$$\begin{aligned}\sigma &= D^2 u,\\ \det(I+\sigma) &= f,\end{aligned}$$

where σ is a 2×2 tensor.

Written in weak form, our problem is to find $(u, \sigma) \in V \times \Sigma = W$ such that

$$\begin{gathered}\int_\Omega \tau : (\sigma + D^2 u)\,\mathrm{d}x - \int_{\partial\Omega} \tau_{12} n_2 u_x + \tau_{21} n_1 u_y \,\mathrm{d}s = 0, \quad \forall \tau \in \Sigma\\ \int_\Omega v \det(I+\sigma)\,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x \quad \forall v \in V.\end{gathered}$$

This is called a nonvariational discretisation since the PDE is not in a divergence form. Note that we have dropped the boundary terms that vanish due to the boundary condition. To proceed in the discretisation, we simply choose V to be a continuous degree-k finite element space, and Σ to be the 2×2 tensor continuous finite element space of the same degree. Since we have Neumann boundary conditions, this variational problem has a null space consisting of the constant functions in V.

For Dirichlet boundary conditions, [Awa14] proved that this algorithm converges when $k > 1$. Note that the Jacobian system arising from Newton is only elliptic when $I+\sigma$ is positive-definite; it is observed that positive-definiteness is preserved by Newton iteration and hence we must be careful to choose an appropriate initial guess. This is one of the reasons why we have set things up with $I + \sigma$ here instead of σ as is more conventional for these equations, since then $u = 0$ is an appropriate initial guess. This setup also makes the application of the weak boundary conditions easier.

We now proceed to set up the problem in Firedrake using a square mesh of quadrilaterals.

```
from firedrake import *
n = 100
mesh = UnitSquareMesh(n, n, quadrilateral=True)
```

We construct the quadratic function space for u,

```
V = FunctionSpace(mesh, "CG", 2)
```

and the function space for σ.

```
Sigma = TensorFunctionSpace(mesh, "CG", 2)
```

We then combine them together in a mixed function space.

```
W = V*Sigma
```

Next, we set up the source function, which must integrate to the area of the domain. Note how in the integration of the `Constant` one, we must explicitly specify the domain we wish to integrate over.

```
x, y = SpatialCoordinate(mesh)
fexpr = exp(-(cos(x)**2 + cos(y)**2))
f = Function(V).interpolate(fexpr)
scaling = assemble(Constant(1, domain=mesh)*dx)/assemble(f*dx)
f *= scaling
assert abs(assemble(f*dx)-assemble(Constant(1, domain=mesh)*dx)) < 1.0e-8
```

Now we build the UFL expression for the variational form. We will use the nonlinear solve, so the form needs to be a 1-form that depends on a Function, w.

```
v, tau = TestFunctions(W)
w = Function(W)
u, sigma = split(w)

n = FacetNormal(mesh)

I = Identity(mesh.geometric_dimension())

L = inner(sigma, tau)*dx
L += (inner(div(tau), grad(u))*dx
      - (tau[0, 1]*n[1]*u.dx(0) + tau[1, 0]*n[0]*u.dx(1))*ds)
L -= (det(I + sigma) - f)*v*dx
```

We must specify the nullspace for the operator. First we define a constant nullspace,

```
V_basis = VectorSpaceBasis(constant=True)
```

then we use it to build a nullspace of the mixed function space W.

```
nullspace = MixedVectorSpaceBasis(W, [V_basis, W.sub(1)])
```

Then we set up the variational problem.

```
u_prob = NonlinearVariationalProblem(L, w)
```

We need to set quite a few solver options, so we'll put them into a dictionary.

```
sp_it = {
```

We'll only use stationary preconditioners in the Schur complement, so we can get away with GMRES applied to the whole mixed system

```
#
   "ksp_type": "gmres",
```

We set up a Schur preconditioner, which is of type "fieldsplit". We also need to tell the preconditioner that we want to eliminate σ, which is field "1", to get an equation for u, which is field "0".

```
#
   "pc_type": "fieldsplit",
   "pc_fieldsplit_type": "schur",
   "pc_fieldsplit_0_fields": "1",
   "pc_fieldsplit_1_fields": "0",
```

The "selfp" option selects a diagonal approximation of the A00 block.

```
#
   "pc_fieldsplit_schur_precondition": "selfp",
```

We just use ILU to approximate the inverse of A00, without a KSP solver,

```
#
   "fieldsplit_0_pc_type": "ilu",
   "fieldsplit_0_ksp_type": "preonly",
```

and use GAMG to approximate the inverse of the Schur complement matrix.

```
#
   "fieldsplit_1_ksp_type": "preonly",
   "fieldsplit_1_pc_type": "gamg",
   "fieldsplit_1_mg_levels_pc_type": "sor",
```

Finally, we'd like to see some output to check things are working, and to limit the KSP solver to 20 iterations.

```
#
   "ksp_monitor": None,
   "ksp_max_it": 20,
   "snes_monitor": None
   }
```

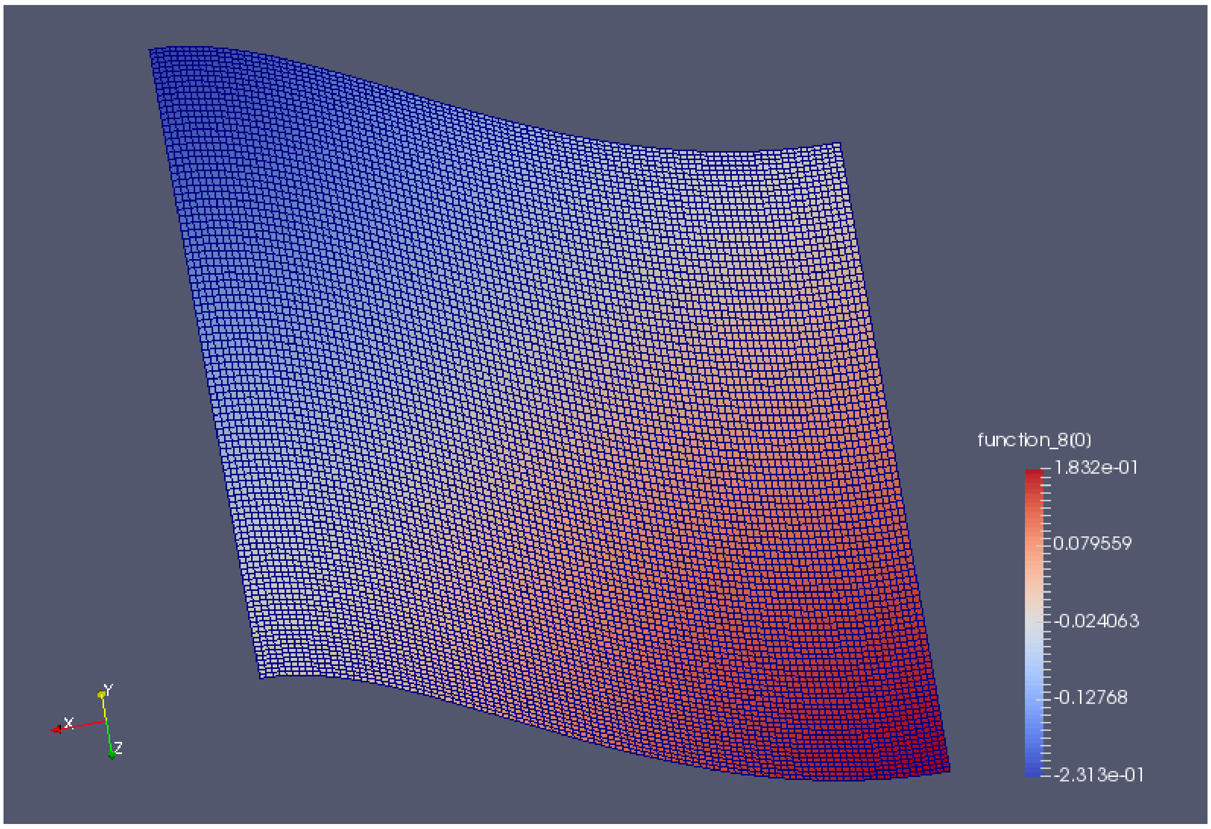
We then put all of these options into the iterative solver,

```
u_solv = NonlinearVariationalSolver(u_prob, nullspace=nullspace,
                                    solver_parameters=sp_it)
```

and output the solution to a file.

```
u, sigma = w.split()
u_solv.solve()
File("u.pvd").write(u)
```

An image of the solution is shown below.



A python script version of this demo can be found here.

References

3.9 Using geometric multigrid solvers in Firedrake

In addition to the full gamut of algebraic solvers offered by PETSc, Firedrake also provides access to multilevel solvers with geometric hierarchies. In this demo, we will see how to use this functionality. We first solve the prototypical elliptic problem, the Poisson equation. We move on to a multi-field example, the Stokes equations, demonstrating how the multigrid functionality composes with fieldsplit preconditioning.

3.9.1 Creating a geometric hierarchy

Geometric multigrid requires a geometric hierarchy of meshes on which the equations will be discretised. To create a hierarchy, we use *MeshHierarchy()* to create a hierarchy of meshes, the resulting object remembers the relationships between them. Currently, these hierarchies are constructed using regular bisection refinement, so we must create a coarse mesh.

```
from firedrake import *

mesh = UnitSquareMesh(8, 8)
```

Now we will create the mesh hierarchy, providing the coarse mesh and the number of refinements we would like. Here, we request four refinements, going from 128 cells on the coarse mesh to 32768 cells on the finest.

```
hierarchy = MeshHierarchy(mesh, 4)
```

3.9.2 Defining the problem: the Poisson equation

Having defined the hierarchy we now need to set up our problem. The most transparent way to do this is to set up the problem on the finest mesh, Firedrake then manages the rediscretised operators by providing appropriate callbacks to PETSc. In this way, we can control the behaviour of the solver entirely through runtime options. So our next step is just to grab the finest mesh and define the problem.

```
mesh = hierarchy[-1]

V = FunctionSpace(mesh, "CG", 1)

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

a = dot(grad(u), grad(v))*dx

bcs = DirichletBC(V, zero(), (1, 2, 3, 4))
```

For a forcing function, we will use a product of sines such that we know the exact solution and can compute an error.

```
x, y = SpatialCoordinate(mesh)

f = -0.5*pi*pi*(4*cos(pi*x) - 5*cos(pi*x*0.5) + 2)*sin(pi*y)

L = f*v*dx
```

The exact solution is:

```
exact = sin(pi*x)*tan(pi*x*0.25)*sin(pi*y)
```

We'll demonstrate a few different sets of solver parameters, so let's define a function that takes in set of parameters and returns the solution

```
def run_solve(parameters):
    u = Function(V)
    solve(a == L, u, bcs=bcs, solver_parameters=parameters)
    return u
```

and another to compute the error.

```
def error(u):
    expect = Function(V).interpolate(exact)
    return norm(assemble(u - expect))
```

Specifying the solver

Let's start with our first test. We'll confirm a working solve by using a direct method.

```
u = run_solve({"ksp_type": "preonly", "pc_type": "lu"})
print('LU solve error', error(u))
```

Next we'll use the conjugate gradient method preconditioned by a geometric multigrid V-cycle. Firedrake automatically takes care of rediscretising the operator on coarse grids, and providing the number of levels to PETSc.

```
u = run_solve({"ksp_type": "cg", "pc_type": "mg"})
print('MG V-cycle + CG error', error(u))
```

For such a simple problem, an appropriately configured multigrid solve can achieve algebraic error equal to discretisation error in one cycle, without the application of a Krylov accelerator. In particular, for the Poisson equation with constant coefficients, a single full multigrid cycle with appropriately chosen smoothers achieves discretisation error. As ever, PETSc allows us to configure the appropriate settings using solver parameters.

```
parameters = {
   "ksp_type": "preonly",
   "pc_type": "mg",
   "pc_mg_type": "full",
   "mg_levels_ksp_type": "chebyshev",
   "mg_levels_ksp_max_it": 2,
   "mg_levels_pc_type": "jacobi"
}

u = run_solve(parameters)
print('MG F-cycle error', error(u))
```

3.9.3 A saddle-point system: The Stokes equations

Having demonstrated basic usage, we'll now move on to an example where the configuration of the multigrid solver is somewhat more complex. This demonstrates how the multigrid functionality composes with the other aspects of solver configuration, like fieldsplit preconditioning. We'll use Taylor-Hood elements and solve a problem with specified velocity inflow and outflow conditions.

```
mesh = RectangleMesh(15, 10, 1.5, 1)

hierarchy = MeshHierarchy(mesh, 3)

mesh = hierarchy[-1]

V = VectorFunctionSpace(mesh, "CG", 2)
W = FunctionSpace(mesh, "CG", 1)
Z = V * W

u, p = TrialFunctions(Z)
v, q = TestFunctions(Z)
nu = Constant(1)

a = (nu*inner(grad(u), grad(v)) - p * div(v) + div(u) * q)*dx

L = inner(Constant((0, 0)), v) * dx

x, y = SpatialCoordinate(mesh)

t = conditional(y < 0.5, y - 0.25, y - 0.75)
l = 1.0/6.0
gbar = conditional(Or(And(0.25 - l/2 < y,
y < 0.25 + l/2),
And(0.75 - l/2 < y,
y < 0.75 + l/2)),
Constant(1.0), Constant(0.0))

value = gbar*(1 - (2*t/l)**2)
inflowoutflow = Function(V).interpolate(as_vector([value, 0]))
bcs = [DirichletBC(Z.sub(0), inflowoutflow, (1, 2)),
DirichletBC(Z.sub(0), zero(2), (3, 4))]
```

First up, we'll use an algebraic preconditioner, with a direct solve, remembering to tell PETSc to use pivoting in the factorisation.

```
u = Function(Z)
solve(a == L, u, bcs=bcs, solver_parameters={"ksp_type": "preonly",
                                             "pc_type": "lu",
                                             "pc_factor_shift_type": "inblocks
↪",
                                             "ksp_monitor": None,
                                             "pmat_type": "aij"})
```

Next we'll use a Schur complement solver, using geometric multigrid to invert the velocity block. The Schur complement is spectrally equivalent to the viscosity-weighted pressure mass matrix. Since the pressure mass matrix does not appear in the original form, we need to supply its bilinear form to the solver ourselves:

```
class Mass(AuxiliaryOperatorPC):

    def form(self, pc, test, trial):
        a = 1/nu * inner(test, trial)*dx
        bcs = None
        return (a, bcs)

parameters = {
    "ksp_type": "gmres",
    "ksp_monitor": None,
    "pc_type": "fieldsplit",
    "pc_fieldsplit_type": "schur",
    "pc_fieldsplit_schur_fact_type": "lower",
    "fieldsplit_0_ksp_type": "preonly",
    "fieldsplit_0_pc_type": "mg",
    "fieldsplit_1_ksp_type": "preonly",
    "fieldsplit_1_pc_type": "python",
    "fieldsplit_1_pc_python_type": "__main__.Mass",
    "fieldsplit_1_aux_pc_type": "bjacobi",
    "fieldsplit_1_aux_sub_pc_type": "icc",
}

u = Function(Z)
solve(a == L, u, bcs=bcs, solver_parameters=parameters)
```

Finally, we'll use coupled geometric multigrid on the full problem, using Schur complement "smoothers" on each level. On the coarse grid we use a full factorisation for the velocity and Schur complement approximations, whereas on the finer levels we use incomplete factorisations for the velocity block and Schur complement approximations.

Note: If we wanted to just use LU for the velocity-pressure system on the coarse grid we would have to say `"mat_type": "aij"`, rather than `"mat_type": "nest"`.

```
parameters = {
      "ksp_type": "gcr",
      "ksp_monitor": None,
      "mat_type": "nest",
      "pc_type": "mg",
      "mg_coarse_ksp_type": "preonly",
      "mg_coarse_pc_type": "fieldsplit",
      "mg_coarse_pc_fieldsplit_type": "schur",
      "mg_coarse_pc_fieldsplit_schur_fact_type": "full",
      "mg_coarse_fieldsplit_0_ksp_type": "preonly",
      "mg_coarse_fieldsplit_0_pc_type": "lu",
```

(continues on next page)

(continued from previous page)

```
        "mg_coarse_fieldsplit_1_ksp_type": "preonly",
        "mg_coarse_fieldsplit_1_pc_type": "python",
        "mg_coarse_fieldsplit_1_pc_python_type": "__main__.Mass",
        "mg_coarse_fieldsplit_1_aux_pc_type": "cholesky",
        "mg_levels_ksp_type": "richardson",
        "mg_levels_ksp_max_it": 1,
        "mg_levels_pc_type": "fieldsplit",
        "mg_levels_pc_fieldsplit_type": "schur",
        "mg_levels_pc_fieldsplit_schur_fact_type": "upper",
        "mg_levels_fieldsplit_0_ksp_type": "richardson",
        "mg_levels_fieldsplit_0_ksp_convergence_test": "skip",
        "mg_levels_fieldsplit_0_ksp_max_it": 2,
        "mg_levels_fieldsplit_0_ksp_richardson_self_scale": None,
        "mg_levels_fieldsplit_0_pc_type": "bjacobi",
        "mg_levels_fieldsplit_0_sub_pc_type": "ilu",
        "mg_levels_fieldsplit_1_ksp_type": "richardson",
        "mg_levels_fieldsplit_1_ksp_convergence_test": "skip",
        "mg_levels_fieldsplit_1_ksp_richardson_self_scale": None,
        "mg_levels_fieldsplit_1_ksp_max_it": 3,
        "mg_levels_fieldsplit_1_pc_type": "python",
        "mg_levels_fieldsplit_1_pc_python_type": "__main__.Mass",
        "mg_levels_fieldsplit_1_aux_pc_type": "bjacobi",
        "mg_levels_fieldsplit_1_aux_sub_pc_type": "icc",
}

u = Function(Z)
solve(a == L, u, bcs=bcs, solver_parameters=parameters)
```

Finally, we'll write the solution for visualisation with Paraview.

```
u, p = u.split()
u.rename("Velocity")
p.rename("Pressure")

File("stokes.pvd").write(u, p)
```

A runnable python version of this demo can be found here.

3.10 Linear mixed fluid-structure interaction system

This tutorial demonstrates the use of subdomain functionality and show how to describe a system consisting of multiple materials in Firedrake.

The tutorial was contributed by Tomasz Salwa and Onno Bokhove.

The model considered consists of fluid with a free surface and an elastic solid. We will be using interchangeably notions of fluid/water and structure/solid/beam. For simplicity (and speed of computation) we consider a model in 2D, however it can be easily generalised to 3D. The starting point is the linearised version (domain is fixed) of the fully nonlinear variational principle.

In non-dimensional units:

$$
\begin{aligned}
0 =&\delta \int_0^{t_{\text{end}}} \int \left(\partial_t\eta\phi - \frac{1}{2}\eta^2\right) \mathrm{d}S_f - \int \frac{1}{2}|\nabla\phi|^2 \mathrm{d}x_F \\
&+ \int \mathbf{n}\cdot\partial_t\mathbf{X}\phi \,\mathrm{d}s_s \\
&+ \int \rho_0\partial_t\mathbf{X}\cdot\mathbf{U} - \frac{1}{2}\rho_0|\mathbf{U}|^2 - \frac{1}{2}\lambda e_{ii}e_{jj} - \mu e_{ij}e_{ij} \,\mathrm{d}x_S \,\mathrm{d}t\,,
\end{aligned}
$$

in which the first line contains integration over fluid domain, second, fluid-structure interface, and third, structure domain. The following notions are used:

- η - free surface deviation
- ϕ - fluid flow potential
- ρ_0 - structure density (in fluid density units)
- λ - first Lame constant (material parameter)
- μ - second Lame constant (material parameter)
- $\mathbf{X}$ - structure displacement
- $\mathbf{U}$ - structure velocity
- $e_{ij} = \frac{1}{2}\left(\frac{\partial X_j}{\partial x_i} + \frac{\partial X_i}{\partial x_j}\right)$ - linear strain tensor; i, j denote vector components
- $\mathrm{d}S_f$ - integration element over fluid free surface
- $\mathrm{d}s_s$ - integration element over structure-fluid interface
- $\mathrm{d}x_F$ - integration element over fluid domain
- $\mathrm{d}x_S$ - integration element over structure domain

After numerous manipulations (described in detail in [SBK17]) and evaluation of individual variations, the time-discrete equations, with symplectic Euler scheme, that we would like to implement in Firedrake, are:

$$\int v\phi^{n+1}\,\mathrm{d}S_f = \int v(\phi^n - \Delta t\eta^n)\,\mathrm{d}S_f$$

$$\begin{aligned}\int \rho_0 \mathbf{v}\cdot\mathbf{U}^{n+1}\,\mathrm{d}x_S + \underline{\int \mathbf{n}\cdot\mathbf{v}\,\phi^{n+1}\,\mathrm{d}s_s} = \rho_0 \int \mathbf{v}\cdot\mathbf{U}^n\,\mathrm{d}x_S \\ - \Delta t \int \left(\lambda\nabla\cdot\mathbf{v}\nabla\cdot\mathbf{X}^n + \mu\frac{\partial X_j^n}{\partial x_i}\left(\frac{\partial v_i}{\partial x_j} + \frac{\partial v_j}{\partial x_i}\right)\right)\mathrm{d}x_S \\ \underline{+\int \mathbf{n}\cdot\mathbf{v}\,\phi^n\,\mathrm{d}s_s}\end{aligned}$$

$$\int \nabla v\cdot\nabla\phi^{n+1}\,\mathrm{d}x_F \underline{- \int v\mathbf{n}\cdot\mathbf{U}^{n+1}\,\mathrm{d}s_s} = 0$$

$$\begin{aligned}\int v\eta^{n+1}\,\mathrm{d}S_f = \int v\eta^n\,\mathrm{d}S_f + \Delta t \int \nabla v\cdot\nabla\phi^{n+1}\,\mathrm{d}x_F \\ \underline{-\Delta t \int v\mathbf{n}\cdot\mathbf{U}^{n+1}\,\mathrm{d}s_s}\end{aligned}$$

$$\int \mathbf{v}\cdot\mathbf{X}^{n+1}\,\mathrm{d}x_S = \int \mathbf{v}\cdot(\mathbf{X}^n + \Delta t\mathbf{U}^{n+1})\,\mathrm{d}x_S\,.$$

The underlined terms are the coupling terms. Note that the first equation for ϕ at the free surface is solved on the free surface only, the last equation for $\mathbf{X}$ in the structure domain, while the others are solved in both domains. Moreover, the second and third equations for ϕ and $\mathbf{U}$ need to be solved simultaneously. The geometry of the system with initial condition is shown below.

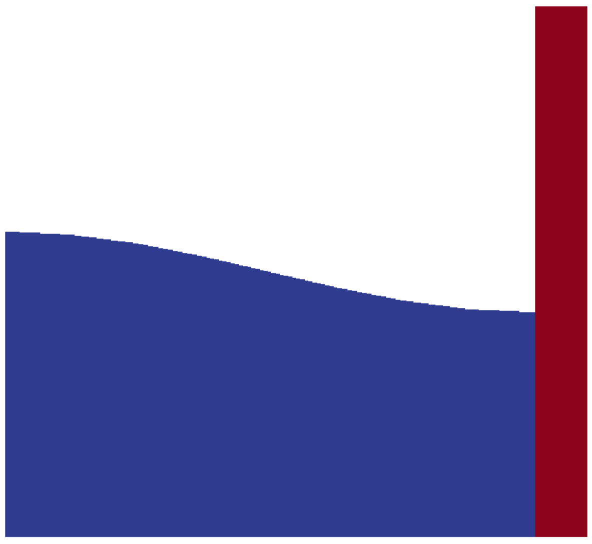


Figure4: Geometry and initial condition in the system. Fluid (blue) with deflected free surface and the structure (red).

Now we present the code used to solve the system of equations above. We start with appropriate imports:

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

Then, we set parameters of the simulation:

```
# parameters in SI units
t_end = 5.0  # time of simulation [s]
dt = 0.005  # time step [s]
g = 9.8  # gravitational acceleration
# water
Lx = 20.0  # length of the tank [m] in x-direction; needed for computing␣
↪initial condition
Lz = 10.0  # height of the tank [m]; needed for computing initial condition
rho = 1000.0  # fluid density in kg/m^2 in 2D [water]
# solid parameters
#  - we use a sufficiently soft material to be able to see noticeable␣
↪structural displacement
rho_B = 7700.0  # structure density in kg/m^2 in 2D
lam = 1e7  # N/m in 2D - first Lame constant
mu = 1e7  # N/m in 2D - second Lame constant
# mesh
mesh = Mesh("L_domain.msh")
# these numbers must match the ones defined in the mesh file
fluid_id = 1  # fluid subdomain
structure_id = 2  # structure subdomain
bottom_id = 1  # structure bottom
top_id = 6  # fluid surface
interface_id = 9  # fluid-structure interface
# control parameters
output_data_every_x_time_steps = 20  # to avoid saving data every time step
coupling = True  # turn on coupling terms
```

The equations are in nondimensional units, hence we transform:

```
L = Lz
T = L / math.sqrt(g * L)
t_end /= T
dt /= T
Lx /= L
Lz /= L
rho_B /= rho
lam /= g * rho * L
mu /= g * rho * L
rho = 1.0  # or equivalently rho /= rho
```

Let us define function spaces, including the mixed one:

```
V_W = FunctionSpace(mesh, "CG", 1)
V_B = VectorFunctionSpace(mesh, "CG", 1)
mixed_V = V_W * V_B
```

Then, we define functions. First, in the fluid domain:

```
phi = Function(V_W, name="phi")
phi_f = Function(V_W, name="phi_f")  # at the free surface
eta = Function(V_W, name="eta")
trial_W = TrialFunction(V_W)
v_W = TestFunction(V_W)
```

Second, in the beam domain:

```
X = Function(V_B, name="X")
U = Function(V_B, name="U")
trial_B = TrialFunction(V_B)
v_B = TestFunction(V_B)
```

And last, mixed functions in the mixed domain:

```
trial_f, trial_s = TrialFunctions(mixed_V)
v_f, v_s = TestFunctions(mixed_V)
tmp_f = Function(V_W)
tmp_s = Function(V_B)
result_mixed = Function(mixed_V)
```

We need auxiliary indicator functions, that are 0 in one subdomain and 1 in the other. They are needed both in “CG” and “DG” space. We use the fact that the fluid and structure subdomains are defined in the mesh file with an appropriate ID number that Firedrake is able to recognise. That can be used in constructing indicator functions:

```
V_DG0_W = FunctionSpace(mesh, "DG", 0)
V_DG0_B = FunctionSpace(mesh, "DG", 0)

# Heaviside step function in fluid
I_W = Function(V_DG0_W)
par_loop(("{[i] : 0 <= i < f.dofs}", "f[i, 0] = 1.0"),
         dx(fluid_id),
         {"f": (I_W, WRITE)},
         is_loopy_kernel=True)
I_cg_W = Function(V_W)
par_loop(("{[i] : 0 <= i < A.dofs}", "A[i, 0] = fmax(A[i, 0], B[0, 0])"),
         dx,
         {"A": (I_cg_W, RW), "B": (I_W, READ)},
         is_loopy_kernel=True)

# Heaviside step function in solid
I_B = Function(V_DG0_B)
par_loop(("{[i] : 0 <= i < f.dofs}", "f[i, 0] = 1.0"),
         dx(structure_id),
```

(continues on next page)

(continued from previous page)

```
         {"f": (I_B, WRITE)},
         is_loopy_kernel=True)
I_cg_B = Function(V_B)
par_loop(("{[i, j] : 0 <= i < A.dofs and 0 <= j < 2}", "A[i, j] = fmax(A[i,␣
↪j], B[0, 0])"),
         dx,
         {"A": (I_cg_B, RW), "B": (I_B, READ)},
         is_loopy_kernel=True)
```

We use indicator functions to construct normal unit vector outward to the fluid domain at the fluid-structure interface:

```
n_vec = FacetNormal(mesh)
n_int = I_B("+") * n_vec("+") + I_B("-") * n_vec("-")
```

Now we can construct special boundary conditions that limit the solvers only to the appropriate subdomains of our interest:

```
class MyBC(DirichletBC):
    def __init__(self, V, value, markers):
        # Call superclass init
        # We provide a dummy subdomain id.
        super(MyBC, self).__init__(V, value, 0)
        # Override the "nodes" property which says where the boundary
        # condition is to be applied.
        self.nodes = np.unique(np.where(markers.dat.data_ro_with_halos ==␣
↪0)[0])


def surface_BC():
    # This will set nodes on the top boundary to 1.
    bc = DirichletBC(V_W, 1, top_id)
    # We will use this function to determine the new BC nodes (all those
    # that aren't on the boundary)
    f = Function(V_W, dtype=np.int32)
    # f is now 0 everywhere, except on the boundary
    bc.apply(f)
    # Now I can use MyBC to create a "boundary condition" to zero out all
    # the nodes that are *not* on the top boundary:
    return MyBC(V_W, 0, f)


# same as above, but in the mixed space
def surface_BC_mixed():
    bc_mixed = DirichletBC(mixed_V.sub(0), 1, top_id)
    f_mixed = Function(mixed_V.sub(0), dtype=np.int32)
    bc_mixed.apply(f_mixed)
    return MyBC(mixed_V.sub(0), 0, f_mixed)
```

(continues on next page)

(continued from previous page)

```
BC_exclude_beyond_surface = surface_BC()
BC_exclude_beyond_surface_mixed = surface_BC_mixed()
BC_exclude_beyond_solid = MyBC(V_B, 0, I_cg_B)
BC_exclude_beyond_water_mixed = MyBC(mixed_V.sub(0), 0, I_cg_W)
BC_exclude_beyond_solid_mixed = MyBC(mixed_V.sub(1), 0, I_cg_B)
```

Finally, we are ready to define the solvers of our equations. First, equation for ϕ at the free surface:

```
a_phi_f = trial_W * v_W * ds(top_id)
L_phi_f = (phi_f - dt * eta) * v_W * ds(top_id)
LVP_phi_f = LinearVariationalProblem(a_phi_f, L_phi_f, phi_f, bcs=BC_exclude_
↪beyond_surface)
LVS_phi_f = LinearVariationalSolver(LVP_phi_f)
```

Second, equation for the beam displacement $\mathbf{X}$, where we also fix it to the bottom by applying zero Dirichlet boundary condition:

```
a_X = dot(trial_B, v_B) * dx(structure_id)
L_X = dot((X + dt * U), v_B) * dx(structure_id)
# no-motion beam bottom boundary condition
BC_bottom = DirichletBC(V_B, as_vector([0.0, 0.0]), bottom_id)
LVP_X = LinearVariationalProblem(a_X, L_X, X, bcs=[BC_bottom, BC_exclude_
↪beyond_solid])
LVS_X = LinearVariationalSolver(LVP_X)
```

Finally, we define solvers for ϕ, $\mathbf{U}$ and η in the mixed domain. In particular, value of ϕ at the free surface is used as a boundary condition. Note that avg(...) is necessary for terms in expressions containing n_int, which is built in "DG" space:

```
# phi-U
# no-motion beam bottom boundary condition in the mixed space
BC_bottom_mixed = DirichletBC(mixed_V.sub(1), as_vector([0.0, 0.0]), bottom_
↪id)
# boundary condition to set phi_f at the free surface
BC_phi_f = DirichletBC(mixed_V.sub(0), phi_f, top_id)
delX = nabla_grad(X)
delv_B = nabla_grad(v_s)
T_x_dv = lam * div(X) * div(v_s) + mu * (inner(delX, delv_B + transpose(delv_
↪B)))
a_U = rho_B * dot(trial_s, v_s) * dx(structure_id)
L_U = (rho_B * dot(U, v_s) - dt * T_x_dv) * dx(structure_id)
a_phi = dot(grad(trial_f), grad(v_f)) * dx(fluid_id)
if coupling:
    a_U += dot(avg(v_s), n_int) * avg(trial_f) * dS  # avg(...) necessary␣
↪here and below
    L_U += dot(avg(v_s), n_int) * avg(phi) * dS
    a_phi += -dot(n_int, avg(trial_s)) * avg(v_f) * dS
LVP_U_phi = LinearVariationalProblem(a_U + a_phi, L_U, result_mixed,
```

(continues on next page)

(continued from previous page)

```
                                                  bcs=[BC_phi_f,
                                                       BC_bottom_mixed,
                                                       BC_exclude_beyond_solid_mixed,
                                                       BC_exclude_beyond_water_mixed])
LVS_U_phi = LinearVariationalSolver(LVP_U_phi)

# eta
a_eta = trial_W * v_W * ds(top_id)
L_eta = eta * v_W * ds(top_id) + dt * dot(grad(v_W), grad(phi)) * dx(fluid_id)
if coupling:
    L_eta += -dt * dot(n_int, avg(U)) * avg(v_W) * dS
LVP_eta = LinearVariationalProblem(a_eta, L_eta, eta, bcs=BC_exclude_beyond_
↪surface)
LVS_eta = LinearVariationalSolver(LVP_eta)
```

Let us set the initial condition. We choose no motion at the beginning in both fluid and structure, zero displacement in the structure and deflected free surface in the fluid. The shape of the deflection is computed from the analytical solution:

```
# initial condition in fluid based on analytical solution
# compute analytical initial phi and eta
n_mode = 1
a = 0.0 * T / L ** 2  # in nondim units
b = 5.0 * T / L ** 2  # in nondim units
lambda_x = np.pi * n_mode / Lx
omega = np.sqrt(lambda_x * np.tanh(lambda_x * Lz))
x = mesh.coordinates
phi_exact_expr = a * cos(lambda_x * x[0]) * cosh(lambda_x * x[1])
eta_exact_expr = -omega * b * cos(lambda_x * x[0]) * cosh(lambda_x * Lz)

bc_top = DirichletBC(V_W, 0, top_id)
eta.assign(0.0)
phi.assign(0.0)
eta_exact = Function(V_W)
eta_exact.interpolate(eta_exact_expr)
eta.assign(eta_exact, bc_top.node_set)
phi.interpolate(phi_exact_expr)
phi_f.assign(phi, bc_top.node_set)
```

A file to store data for visualization:

```
outfile_phi = File("results_pvd/phi.pvd")
```

To save data for visualization, we change the position of the nodes in the mesh, so that they represent the computed dynamic position of the free surface and the structure:

```
def output_data():
    output_data.counter += 1
    if output_data.counter % output_data_every_x_time_steps != 0:
        return
```

(continues on next page)

(continued from previous page)

```
    mesh_static = mesh.coordinates.vector().get_local()
    mesh.coordinates.vector().set_local(mesh_static + X.vector().get_local())
    mesh.coordinates.dat.data[:, 1] += eta.dat.data_ro
    outfile_phi.write(phi)
    mesh.coordinates.vector().set_local(mesh_static)


output_data.counter = -1  # -1 to exclude counting print of initial state
```

In the end, we proceed with the actual computation loop:

```
t = 0.0
output_data()

while t <= t_end + dt:
    t += dt
    print("time = ", t * T)
    # symplectic Euler scheme
    LVS_phi_f.solve()
    LVS_U_phi.solve()
    tmp_f, tmp_s = result_mixed.split()
    phi.assign(tmp_f)
    U.assign(tmp_s)
    LVS_eta.solve()
    LVS_X.solve()

    output_data()
```

The result of the computation, visualised with paraview, is shown below.

The mesh is deflected for visualization only. As the model is linear, the actual mesh used for computation is fixed. Colours indicate values of the flow potential ϕ.

A python script version of this demo can be found here.

The mesh file is here. It can be generated with gmsh from this file with a command: gmsh -2 L_domain.geo.

An extended 3D version of this code is published here.

The work is based on the articles [SBK17] and [SBK16]. The authors gratefully acknowledge funding from European Commission, Marie Curie Actions - Initial Training Networks (ITN), project number 607596.

References

3.11 Scalar wave equation with higher-order mass lumping

3.11.1 Introduction

In this demo, we solve the scalar wave equation with a fully explicit, higher-order (up to degree 5) mass lumping technique for triangular and tetrahedral meshes. This scalar wave equation is widely used in seismology to model seismic waves and is especially popular in algorithms for geophysical exploration such as Full Waveform Inversion and Reverse Time Migration. This tutorial demonstrates how to use the mass-lumped triangular elements originally discovered in [CJKMVV99] and later improved upon in [GMvdV18] in the Firedrake computing environment.**

The short tutorial was prepared by `Keith J. Roberts <mailto:krober@usp.br>`__

The scalar wave equation is:

$$\begin{aligned} \rho \partial_t^2 u = \nabla \cdot c^2 \nabla u + f \\ u = 0 \\ u|_{t=0} = u_0 \\ \partial_t u|_{t=0} = v_0 \end{aligned}$$

where c is the scalar wave speed and rho is the density (assumed to be 1 for simplicity).

The weak formulation is finding $u \in V$ such that:

$$< \partial_t(\rho \partial_t u), v > + a(u, v) = (f, w)$$

where $< \cdot, \cdot >$ denotes the pairing between $H^{-1}(\Omega)$ and $H_0^1(\Omega)$, $(\cdot, \cdot)$ denotes the $L^2(\Omega)$ inner product, and $a(\cdot, \cdot) : H_0^1(\Omega) \times H_0^1(\Omega) \rightarrow \mathbb{R}$ is the elliptic operator given by:

$$a(u, v) := \int_\Omega c^2 \nabla u \cdot \nabla v \mathrm{d}x$$

We solve the above weak formulation using the finite element method.

In the work of [CJKMVV99] and later [GMvdV18], several triangular and tetrahedral elements were discovered that could produce convergent and stable mass lumping for $p \geq 2$. These elements have enriched function spaces in the interior of the element that lead to more degree-of-freedom per element than the standard Lagrange element. However, this additional computational cost is offset by the fact that these elements produce diagonal matrices that are comparatively quick to solve, which improve simulation throughput especially at scale. Firedrake supports (through FInAT) these elements up to degree 5 on triangular, and degree 3 on tetrahedral meshes. They can be selected by choosing the “KMV” finite element.

In addition to importing firedrake as usual, we will need to construct the correct quadrature rules for the mass-lumping by hand. FInAT is responsible for providing these quadrature rules, so we import it here too.:

```
from firedrake import *
import finat

import math
```

A simple uniform triangular mesh is created:

```
mesh = UnitSquareMesh(50, 50)
```

We choose a degree 2 *KMV* continuous function space, set it up and then create some functions used in time-stepping:

```
V = FunctionSpace(mesh, "KMV", 2)

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

u_np1 = Function(V)  # timestep n+1
u_n = Function(V)    # timestep n
u_nm1 = Function(V)  # timestep n-1
```

Note: The user can select orders up to p=5 for triangles and up to p=3 for tetrahedra.

We create an output file to hold the simulation results:

```
outfile = File("out.pvd")
```

Now we set the time-stepping variables performing a simulation for 1 second with a timestep of 0.001 seconds:

```
T = 1.0
dt = 0.001
t = 0
step = 0
```

Ricker wavelets are often used to excite the domain in seismology. They have one free parameter: a peak frequency peak.

Here we inject a Ricker wavelet into the domain with a frequency of 6 Hz. For simplicity, we set the seismic velocity in the domain to be a constant:

```
freq = 6
c = Constant(1.5)
```

The following two functions are used to inject the Ricker wavelet source into the domain. We create a time-varying function to model the time evolution of the Ricker wavelet:

```
def RickerWavelet(t, freq, amp=1.0):
    # Shift in time so the entire wavelet is injected
    t = t - (math.sqrt(6.0) / (math.pi * freq))
    return amp * (
        1.0 - (1.0 / 2.0) * (2.0 * math.pi * freq) * (2.0 * math.pi * freq) *␣
↪t * t
    )
```

The spatial distribution of the source function is a Guassian kernel with a standard deviation of 2,000 so that it's sufficiently localized to emulate a Dirac delta function:

```
def delta_expr(x0, x, y, sigma_x=2000.0):
    sigma_x = Constant(sigma_x)
    return exp(-sigma_x * ((x - x0[0]) ** 2 + (y - x0[1]) ** 2))
```

To assemble the diagonal mass matrix, we need to create the matching colocated quadrature rule. FInAT implements custom "KMV" quadrature rules to do this. We obtain the appropriate cell from the function space, along with the degree of the element and construct the quadrature rule:

```
quad_rule = finat.quadrature.make_quadrature(V.finat_element.cell, V.ufl_
↪element().degree(), "KMV")
```

Then we make a new Measure object that uses this rule:

```
dxlump=dx(rule=quad_rule)
```

To discretize $\partial_t^2 u$ we use a central scheme

$$\partial_t^2 u = \frac{u^{n+1} - 2 * u^n + u^{n-1}}{\Delta t^2}$$

Substituting the above into the time derivative term in the variational form leads to

$$\frac{u^{n+1} - 2 * u^n + u^{n-1}}{\Delta t^2}), v > +a(u, v) = (f, w)$$

Using Firedrake, we specify the mass matrix using the special quadrature rule with the Measure object we created above like so:

```
m = (u - 2.0 * u_n + u_nm1) / Constant(dt * dt) * v * dxlump
```

Note: Mass lumping is a common technique in finite elements to produce a diagonal mass matrix that can be trivially inverted resulting in a in very efficient explicit time integration scheme. It's usually done with nodal basis functions and an inexact quadrature rule for the mass matrix. A diagonal matrix is obtained when the integration points coincide with the nodes of the basis function. However, when using elements of $p \geq 2$, this technique does not result in a stable and accurate finite element scheme and new elements must be found such as those detailed in :cite:Chin:1999 .

The stiffness matrix $a(u, v)$ is formed using a standard quadrature rule and is treated explicitly:

```
a = c*c*dot(grad(u_n), grad(v)) * dx
```

The source is injected at the center of the unit square:

```
x, y = SpatialCoordinate(mesh)
source = Constant([0.5, 0.5])
ricker = Constant(0.0)
ricker.assign(RickerWavelet(t, freq))
```

We also create a function *R* to save the assembled RHS vector:

```
R = Function(V)
```

Finally, we define the whole variational form F, assemble it, and then create a cached PETSc *LinearSolver* object to efficiently timestep with:

```
F = m + a -  delta_expr(source, x, y)*ricker * v * dx
a, r = lhs(F), rhs(F)
A = assemble(a)
solver = LinearSolver(A, solver_parameters={"ksp_type": "preonly", "pc_type":
↪"jacobi"})
```

Note: Since we have arranged that the matrix A is diagonal, we can invert it with a single application of Jacobi iteration. We select this here using appropriate solver parameters, which tell PETSc to construct a solver which just applies a single step of Jacobi preconditioning.

Now we are ready to start the time-stepping loop:

```
step = 0
while t < T:
    step += 1

    # Update the RHS vector according to the current simulation time `t`

    ricker.assign(RickerWavelet(t, freq))

    R = assemble(r, tensor=R)

    # Call the solver object to do point-wise division to solve the system.

    solver.solve(u_np1, R)

    # Exchange the solution at the two time-stepping levels.

    u_nm1.assign(u_n)
    u_n.assign(u_np1)

    # Increment the time and write the solution to the file for visualization␣
↪in ParaView.

    t += dt
    if step % 10 == 0:
        print("Elapsed time is: "+str(t))
        outfile.write(u_n, time=t)
```

References

3.12 Stokes Equations

A simple example of a saddle-point system, we will use the Stokes equations to demonstrate some of the ways we can do field-splitting with matrix-free operators. We set up the problem as a lid-driven cavity.

As ever, we import firedrake and define a mesh.:

```
from firedrake import *

N = 64

M = UnitSquareMesh(N, N)

V = VectorFunctionSpace(M, "CG", 2)
W = FunctionSpace(M, "CG", 1)
Z = V * W

u, p = TrialFunctions(Z)
v, q = TestFunctions(Z)

a = (inner(grad(u), grad(v)) - inner(p, div(v)) + inner(div(u), q))*dx

L = inner(Constant((0, 0)), v) * dx
```

The boundary conditions are defined on the velocity space. Zero Dirichlet conditions on the bottom and side walls, a constant $u = (1, 0)$ condition on the lid.:

```
bcs = [DirichletBC(Z.sub(0), Constant((1, 0)), (4,)),
       DirichletBC(Z.sub(0), Constant((0, 0)), (1, 2, 3))]

up = Function(Z)
```

Since we do not specify boundary conditions on the pressure space, it is only defined up to a constant. We will remove this component of the solution in the solver by providing the appropriate nullspace.:

```
nullspace = MixedVectorSpaceBasis(
    Z, [Z.sub(0), VectorSpaceBasis(constant=True)])
```

First up, we will solve the problem directly. For this to work, the sparse direct solver MUMPS must be installed. Hence this solve is wrapped in a `try/except` block so that an error is not raised in the case that it is not, to do this we must import `PETSc`:

```
from firedrake.petsc import PETSc
```

To factor the matrix from this mixed system, we must specify a `mat_type` of `aij` to the solve call.:

```
try:
    solve(a == L, up, bcs=bcs, nullspace=nullspace,
          solver_parameters={"ksp_type": "gmres",
                             "mat_type": "aij",
                             "pc_type": "lu",
                             "pc_factor_mat_solver_type": "mumps"})
except PETSc.Error as e:
    if e.ierr == 92:
        warning("MUMPS not installed, skipping direct solve")
    else:
        raise e
```

Now we'll use a Schur complement preconditioner using unassembled matrices. We can do all of this purely by changing the solver options. We'll define the parameters separately to run through the options.:

```
parameters = {
```

First up we select the unassembled matrix type:

```
"mat_type": "matfree",
```

Now we configure the solver, using GMRES using the diagonal part of the Schur complement factorisation to approximate the inverse. We'll also monitor the convergence of the residual, and ask PETSc to view the configured Krylov solver object.:

```
"ksp_type": "gmres",
"ksp_monitor_true_residual": None,
"ksp_view": None,
"pc_type": "fieldsplit",
"pc_fieldsplit_type": "schur",
"pc_fieldsplit_schur_fact_type": "diag",
```

Next we configure the solvers for the blocks. For the velocity block, we use an *AssembledPC* and approximate the inverse of the vector laplacian using a single multigrid V-cycle.:

```
"fieldsplit_0_ksp_type": "preonly",
"fieldsplit_0_pc_type": "python",
"fieldsplit_0_pc_python_type": "firedrake.AssembledPC",
"fieldsplit_0_assembled_pc_type": "hypre",
```

For the Schur complement block, we approximate the inverse of the schur complement with a pressure mass inverse. For constant viscosity this works well. For variable, but low-contrast viscosity, one should use a viscosity-weighted mass-matrix. This is achievable by passing a dictionary with "mu" associated with the viscosity into solve. The MassInvPC will choose a default value of 1.0 if not set. For high viscosity contrasts, this preconditioner is mesh-dependent and should be replaced by some form of approximate commutator.:

```
"fieldsplit_1_ksp_type": "preonly",
"fieldsplit_1_pc_type": "python",
"fieldsplit_1_pc_python_type": "firedrake.MassInvPC",
```

The mass inverse is dense, and therefore approximated with a Krylov iteration, which we configure now:

```
    "fieldsplit_1_Mp_ksp_type": "preonly",
    "fieldsplit_1_Mp_pc_type": "ilu"
}
```

Having set up the parameters, we can now go ahead and solve the problem.:

```
up.assign(0)
solve(a == L, up, bcs=bcs, nullspace=nullspace, solver_parameters=parameters)
```

Last, but not least, we'll write the solution to a file for later visualisation. We split the function into its velocity and pressure parts and give them reasonable names, then write them to a paraview file.:

```
u, p = up.split()
u.rename("Velocity")
p.rename("Pressure")

File("stokes.pvd").write(u, p)
```

By default, the mass matrix is assembled in the *MassInvPC* preconditioner, however, this can be controlled using a `mat_type` argument. To do this, we must specify the `mat_type` inside the preconditioner. We can use the previous set of parameters and just modify them slightly.

```
parameters["fieldsplit_1_Mp_mat_type"] = "matfree"
```

With an unassembled matrix, of course, we are not able to use standard preconditioners, so for this example, we will just invert the mass matrix using unpreconditioned conjugate gradients.

```
parameters["fieldsplit_1_Mp_ksp_type"] = "cg"
parameters["fieldsplit_1_Mp_pc_type"] = "none"

up.assign(0)
solve(a == L, up, bcs=bcs, nullspace=nullspace, solver_parameters=parameters)
```

A runnable python script implementing this demo file is available here.

3.13 Navier-Stokes equations

We solve the Navier-Stokes equations using Taylor-Hood elements. The example is that of a lid-driven cavity.

```
from firedrake import *

N = 64

M = UnitSquareMesh(N, N)

```

(continues on next page)

(continued from previous page)

```
V = VectorFunctionSpace(M, "CG", 2)
W = FunctionSpace(M, "CG", 1)
Z = V * W

up = Function(Z)
u, p = split(up)
v, q = TestFunctions(Z)

Re = Constant(100.0)

F = (
    1.0 / Re * inner(grad(u), grad(v)) * dx +
    inner(dot(grad(u), u), v) * dx -
    p * div(v) * dx +
    div(u) * q * dx
)

bcs = [DirichletBC(Z.sub(0), Constant((1, 0)), (4,)),
       DirichletBC(Z.sub(0), Constant((0, 0)), (1, 2, 3))]

nullspace = MixedVectorSpaceBasis(
    Z, [Z.sub(0), VectorSpaceBasis(constant=True)])
```

Having set up the problem, we now move on to solving it. Some preconditioners, for example pressure convection-diffusion (PCD), require information about the the problem that is not easily accessible from the bilinear form. In the case of PCD, we need the Reynolds number and additionally which part of the mixed velocity-pressure space the velocity corresponds to. We provide this information to preconditioners by passing in a dictionary context to the solver. This is propagated down through the matrix-free operators and is therefore accessible to custom preconditioners.

```
appctx = {"Re": Re, "velocity_space": 0}
```

Now we'll solve the problem. First, using a direct solver. Again, if MUMPS is not installed, this solve will not work, so we wrap the solve in a `try/except` block.

```
from firedrake.petsc import PETSc

try:
    solve(F == 0, up, bcs=bcs, nullspace=nullspace,
          solver_parameters={"snes_monitor": None,
                             "ksp_type": "gmres",
                             "mat_type": "aij",
                             "pc_type": "lu",
                             "pc_factor_mat_solver_type": "mumps"})
except PETSc.Error as e:
    if e.ierr == 92:
        warning("MUMPS not installed, skipping direct solve")
    else:
        raise e
```

Now we'll show an example using the *PCDPC* preconditioner that implements the pressure convection-diffusion approximation to the pressure Schur complement. We'll need more solver parameters this time, so again we'll set those up in a dictionary.

```
parameters = {"mat_type": "matfree",
              "snes_monitor": None,
```

We'll use a non-stationary Krylov solve for the Schur complement, so we need to use a flexible Krylov method on the outside.

```
"ksp_type": "fgmres",
"ksp_gmres_modifiedgramschmidt": None,
"ksp_monitor_true_residual": None,
```

Now to configure the preconditioner:

```
"pc_type": "fieldsplit",
"pc_fieldsplit_type": "schur",
"pc_fieldsplit_schur_fact_type": "lower",
```

we invert the velocity block with LU:

```
"fieldsplit_0_ksp_type": "preonly",
"fieldsplit_0_pc_type": "python",
"fieldsplit_0_pc_python_type": "firedrake.AssembledPC",
"fieldsplit_0_assembled_pc_type": "lu",
```

and invert the schur complement inexactly using GMRES, preconditioned with PCD.

```
"fieldsplit_1_ksp_type": "gmres",
"fieldsplit_1_ksp_rtol": 1e-4,
"fieldsplit_1_pc_type": "python",
"fieldsplit_1_pc_python_type": "firedrake.PCDPC",
```

We now need to configure the mass and stiffness solvers in the PCD preconditioner. For this example, we will just invert them with LU, although of course we can use a scalable method if we wish. First the mass solve:

```
"fieldsplit_1_pcd_Mp_ksp_type": "preonly",
"fieldsplit_1_pcd_Mp_pc_type": "lu",
```

and the stiffness solve.:

```
"fieldsplit_1_pcd_Kp_ksp_type": "preonly",
"fieldsplit_1_pcd_Kp_pc_type": "lu",
```

Finally, we just need to decide whether to apply the action of the pressure-space convection-diffusion operator with an assembled matrix or matrix free. Here we will use matrix-free:

```
"fieldsplit_1_pcd_Fp_mat_type": "matfree"}
```

With the parameters set up, we can solve the problem, remembering to pass in the application context so that the PCD preconditioner can find the Reynolds number.

```
up.assign(0)

solve(F == 0, up, bcs=bcs, nullspace=nullspace, solver_parameters=parameters,
      appctx=appctx)
```

And finally we write the results to a file for visualisation.

```
u, p = up.split()
u.rename("Velocity")
p.rename("Pressure")

File("cavity.pvd").write(u, p)
```

A runnable python script implementing this demo file is available here.

3.14 Rayleigh-Benard Convection

This problem involves a variable-temperature incompressible fluid. Variations in the fluid temperature are assumed to affect the momentum balance through a buoyant term (the Boussinesq approximation), leading to a Navier-Stokes equation with a nonlinear coupling to a convection-diffusion equation for temperature.

We will set up the problem using Taylor-Hood elements for the Navier-Stokes part, and piecewise linear elements for the temperature.

```
from firedrake import *

N = 128

M = UnitSquareMesh(N, N)

V = VectorFunctionSpace(M, "CG", 2)
W = FunctionSpace(M, "CG", 1)
Q = FunctionSpace(M, "CG", 1)
Z = V * W * Q

upT = Function(Z)
u, p, T = split(upT)
v, q, S = TestFunctions(Z)
```

Two key physical parameters are the Rayleigh number (Ra), which measures the ratio of energy from buoyant forces to viscous dissipation and heat conduction and the Prandtl number (Pr), which measures the ratio of viscosity to heat conduction.

```
Ra = Constant(200.0)
Pr = Constant(6.8)
```

Along with gravity, which points down.

```
g = Constant((0, -1))

F = (
    inner(grad(u), grad(v))*dx
    + inner(dot(grad(u), u), v)*dx
    - inner(p, div(v))*dx
    - (Ra/Pr)*inner(T*g, v)*dx
    + inner(div(u), q)*dx
    + inner(dot(grad(T), u), S)*dx
    + 1/Pr * inner(grad(T), grad(S))*dx
)
```

There are two common versions of this problem. In one case, heat is applied from bottom to top so that the temperature gradient is enforced parallel to the gravitation. In this case, the temperature difference is applied horizontally, perpendicular to gravity. It tends to make prettier pictures for low Rayleigh numbers, but also tends to take more Newton iterations since the coupling terms in the Jacobian are a bit stronger. Switching to the first case would be a simple change of bits of the boundary associated with the second and third boundary conditions below:

```
bcs = [
    DirichletBC(Z.sub(0), Constant((0, 0)), (1, 2, 3, 4)),
    DirichletBC(Z.sub(2), Constant(1.0), (1,)),
    DirichletBC(Z.sub(2), Constant(0.0), (2,))
]
```

Like Navier-Stokes, the pressure is only defined up to a constant.:

```
nullspace = MixedVectorSpaceBasis(
    Z, [Z.sub(0), VectorSpaceBasis(constant=True), Z.sub(2)])
```

First off, we'll solve the full system using a direct solver. As previously, we use MUMPS, so wrap the solve in `try/except` to avoid errors if it is not available.

```
from firedrake.petsc import PETSc

try:
   solve(F == 0, upT, bcs=bcs, nullspace=nullspace,
         solver_parameters={"mat_type": "aij",
                            "snes_monitor": None,
                            "ksp_type": "gmres",
                            "pc_type": "lu",
                            "pc_factor_mat_solver_type": "mumps"})
except PETSc.Error as e:
    if e.ierr == 92:
        warning("MUMPS not installed, skipping direct solve")
    else:
        raise e
```

For our next trick, we will use a fieldsplit preconditioner. This time, rather than using a Schur complement, we will use a multiplicative type (effectively block Gauss-Seidel). As ever, this has more options, so we'll use a parameters dictionary. We use matrix-free actions for the coupled operator, and solve the linearised system with GMRES preconditioned with a multiplicative fieldsplit.

```
parameters = {"mat_type": "matfree",
              "snes_monitor": None,
              "ksp_type": "gmres",
              "pc_type": "fieldsplit",
              "pc_fieldsplit_type": "multiplicative",
```

We want to split the Navier-Stokes part off from the temperature variable.

```
"pc_fieldsplit_0_fields": "0,1",
"pc_fieldsplit_1_fields": "2",
```

We'll invert the Navier-Stokes block with MUMPS:

```
"fieldsplit_0_ksp_type": "preonly",
"fieldsplit_0_pc_type": "python",
"fieldsplit_0_pc_python_type": "firedrake.AssembledPC",
"fieldsplit_0_assembled_pc_type": "lu",
"fieldsplit_0_assembled_pc_factor_mat_solver_type": "mumps",
```

the temperature block will also be inverted directly, but with plain LU.:

```
"fieldsplit_1_ksp_type": "preonly",
"fieldsplit_1_pc_type": "python",
"fieldsplit_1_pc_python_type": "firedrake.AssembledPC",
"fieldsplit_1_assembled_pc_type": "lu"}
```

Now for the solve.

```
upT.assign(0)
try:
    solve(F == 0, upT, bcs=bcs, nullspace=nullspace,
          solver_parameters=parameters)
except PETSc.Error as e:
    if e.ierr == 92:
        warning("MUMPS not installed, skipping assembled fieldsplit solve")
    else:
        raise e
```

Finally, we'll demonstrate recursive fieldsplitting. We'll use the same multiplicative fieldsplit preconditioner for the velocity-pressure and temperature blocks, but we'll precondition the Navier-Stokes part with `PCDPC` using a lower Schur complement factorisation, and approximately invert the temperature block using algebraic multigrid. There are lots of parameters here, so let's run through them. Since there are many options here, in particular for the nested subsolves, we *specify options using nested*, rather than flat, dictionaries. The solver parameters dictionary can either be a flat dictionary of key-value pairs, where both the keys and the values are strings, or it can be nested. In the latter case, the value should be a dictionary, of options and the key is *prepended* to all keys in the dictionary before passing to the solver.

```
parameters = {"mat_type": "matfree",
              "snes_monitor": None,
```

We'll use inexact GMRES solves to invert the Navier-Stokes block, so the preconditioner as a whole is not stationary, hence we need flexible GMRES.

```
"ksp_type": "fgmres",
"ksp_gmres_modifiedgramschmidt": True,
"pc_type": "fieldsplit",
"pc_fieldsplit_type": "multiplicative",
```

Again we split off Navier-Stokes from the temperature block

```
"pc_fieldsplit_0_fields": "0,1",
"pc_fieldsplit_1_fields": "2",
```

which we solve inexactly using preconditioned GMRES.

```
"fieldsplit_0": {
    "ksp_type": "gmres",
    "ksp_gmres_modifiedgramschmidt": True,
    "ksp_rtol": 1e-2,
    "pc_type": "fieldsplit",
    "pc_fieldsplit_type": "schur",
    "pc_fieldsplit_schur_fact_type": "lower",
```

Invert the velocity block with a single V-cycle of algebraic multigrid:

```
"fieldsplit_0": {
    "ksp_type": "preonly",
    "pc_type": "python",
    "pc_python_type": "firedrake.AssembledPC",
    "assembled_pc_type": "hypre"
},
```

and approximate the Schur complement inverse with PCD.

```
"fieldsplit_1": {
     "ksp_type": "preonly",
     "pc_type": "python",
     "pc_python_type": "firedrake.PCDPC",
```

We need to configure the pressure mass and Poisson solves, along with how to apply the convection-diffusion operator. For the latter, we will use an assembled operator this time round.

```
         "pcd_Mp_ksp_type": "preonly",
         "pcd_Mp_pc_type": "ilu",
         "pcd_Kp_ksp_type": "preonly",
         "pcd_Kp_pc_type": "hypre",
         "pcd_Fp_mat_type": "aij"
```

(continues on next page)

(continued from previous page)

```
    }
},
```

Now for the temperature block, we use a moderately coarse tolerance for algebraic multigrid preconditioned GMRES.

```
     "fieldsplit_1": {
          "ksp_type": "gmres",
          "ksp_rtol": "1e-4",
          "pc_type": "python",
          "pc_python_type": "firedrake.AssembledPC",
          "assembled_pc_type": "hypre"
     }
}
```

And we're done with all the options. All that's left is to solve the problem. Recall that the PCD preconditioner needs to know where the velocity space lives in the velocity-pressure block, which we provide through the application context argument. It also needs to know the Reynolds number, which defaults to 1.0, which happens to work for our problem setup. We haven't added the Rayleigh or Prandtl numbers to the dictionary since our known preconditioners don't actually require them, although doing so would be quite easy.:

```
appctx = {"velocity_space": 0}
upT.assign(0)

solve(F == 0, upT, bcs=bcs, nullspace=nullspace,
      solver_parameters=parameters, appctx=appctx)
```

Finally, we'll output the results for visualisation.

```
u, p, T = upT.split()
u.rename("Velocity")
p.rename("Pressure")
T.rename("Temperature")

File("benard.pvd").write(u, p, T)
```

A runnable python script implementing this demo file is available here.

CHAPTER

FOUR

FIREDRAKE PACKAGE

4.1 Subpackages

4.1.1 firedrake.adjoint package

Submodules

firedrake.adjoint.assembly module

`firedrake.adjoint.assembly.`**`annotate_assemble`**`(`*assemble*`)`

firedrake.adjoint.blocks module

`class` `firedrake.adjoint.blocks.`**`AssembleBlock`**`(`*form*, *ad_block_tag=None*`)`

Bases: `AssembleBlock`, *`Backend`*

`block_helper`

`recompute_component``(`*inputs*, *block_variable*, *idx*, *prepared*`)`

This method must be overridden.

The method should implement a routine for recomputing one output of the block in the forward computations. The output to recompute is determined by the *idx* argument, which corresponds to the index of the output in the outputs list. If the block only has a single output, then *idx* will always be 0.

Args:

inputs (list): A list of the saved input values, determined by the dependencies list. block_variable (BlockVariable): The block variable of the output corresponding to index *idx*. idx (int): The index of the output to compute. prepared (object): Anything returned by the prepare_recompute_component method. Default is None.

Returns:

An object of the same type as *block_variable.checkpoint* which is determined by *OverloadedType._ad_create_checkpoint* (often the same as *block_variable.saved_output*): The new output.

`class` `firedrake.adjoint.blocks.`**`Backend`**

Bases: `object`

backend

compat

class firedrake.adjoint.blocks.**ConstantAssignBlock**(*other*, *ad_block_tag=None*)

Bases: `ConstantAssignBlock`, *Backend*

block_helper

class firedrake.adjoint.blocks.**DirichletBCBlock**(**args*, ***kwargs*)

Bases: `DirichletBCBlock`, *Backend*

block_helper

class firedrake.adjoint.blocks.**FunctionAssignBlock**(*func*, *other*, *ad_block_tag=None*)

Bases: `FunctionAssignBlock`, *Backend*

block_helper

recompute_component(*inputs*, *block_variable*, *idx*, *prepared*)

This method must be overridden.

The method should implement a routine for recomputing one output of the block in the forward computations. The output to recompute is determined by the *idx* argument, which corresponds to the index of the output in the outputs list. If the block only has a single output, then *idx* will always be 0.

Args:

inputs (list): A list of the saved input values, determined by the dependencies list. block_variable (BlockVariable): The block variable of the output corresponding to index *idx*. idx (int): The index of the output to compute. prepared (object): Anything returned by the prepare_recompute_component method. Default is None.

Returns:

An object of the same type as *block_variable.checkpoint* which is determined by *OverloadedType._ad_create_checkpoint* (often the same as *block_variable.saved_output*): The new output.

class firedrake.adjoint.blocks.**FunctionMergeBlock**(*func*, *idx*, *ad_block_tag=None*)

Bases: `Block`, *Backend*

block_helper

evaluate_adj_component(*inputs*, *adj_inputs*, *block_variable*, *idx*, *prepared=None*)

This method should be overridden.

The method should implement a routine for evaluating the adjoint of the block that corresponds to one dependency. If one considers the adjoint action a vector right multiplied with the Jacobian matrix, then this method should return one entry in the resulting product, where the entry returned is decided by the argument *idx*.

Args:

inputs (list): A list of the saved input values, determined by the dependencies list. adj_inputs (list): A list of the adjoint input values, determined by the outputs list. block_variable (BlockVariable): The block variable of the dependency corresponding to index *idx*. idx (int): The index of the component to compute. prepared (object): Anything returned by the prepare_evaluate_adj method. Default is None.

Returns:
An object of a type consistent with the adj_value type of *block_variable*: The resulting product.

evaluate_hessian_component(*inputs*, *hessian_inputs*, *adj_inputs*, *block_variable*, *idx*, *relevant_dependencies*, *prepared=None*)

This method must be overridden.

The method should implement a routine for evaluating the hessian of the block. It is preferable that a "Forward-over-Reverse" scheme is used. Thus the hessians are evaluated in reverse (starting with the last block on the tape).

evaluate_tlm()

Computes the tangent linear action and stores the result in the *tlm_value* attribute of the outputs.

This method will by default call the *evaluate_tlm_component* method for each output.

Args:

markings (bool): If True, then each block_variable will have set *marked_in_path* attribute indicating
whether their tlm components are relevant for computing the final target tlm values. Default is False.

recompute_component(*inputs*, *block_variable*, *idx*, *prepared*)

This method must be overridden.

The method should implement a routine for recomputing one output of the block in the forward computations. The output to recompute is determined by the *idx* argument, which corresponds to the index of the output in the outputs list. If the block only has a single output, then *idx* will always be 0.

Args:
inputs (list): A list of the saved input values, determined by the dependencies list. block_variable (BlockVariable): The block variable of the output corresponding to index *idx*. idx (int): The index of the output to compute. prepared (object): Anything returned by the prepare_recompute_component method. Default is None.

Returns:
An object of the same type as *block_variable.checkpoint* which is determined by *OverloadedType._ad_create_checkpoint* (often the same as *block_variable.saved_output*): The new output.

class firedrake.adjoint.blocks.**FunctionSplitBlock**(*func*, *idx*, *ad_block_tag=None*)

Bases: `Block`, *Backend*

block_helper

evaluate_adj_component(*inputs*, *adj_inputs*, *block_variable*, *idx*, *prepared=None*)

This method should be overridden.

The method should implement a routine for evaluating the adjoint of the block that corresponds to one dependency. If one considers the adjoint action a vector right multiplied with the Jacobian matrix, then this method should return one entry in the resulting product, where the entry returned is decided by the argument *idx*.

Args:
inputs (list): A list of the saved input values, determined by the dependencies list. adj_inputs (list): A list of the adjoint input values, determined by the outputs list. block_variable (BlockVariable): The block variable of the dependency corresponding to index *idx*. idx (int): The index of the component to compute. prepared (object): Anything returned by the prepare_evaluate_adj method. Default is None.

Returns:
An object of a type consistent with the adj_value type of *block_variable*: The resulting product.

`evaluate_hessian_component`(*inputs*, *hessian_inputs*, *adj_inputs*, *block_variable*, *idx*, *relevant_dependencies*, *prepared=None*)

This method must be overridden.

The method should implement a routine for evaluating the hessian of the block. It is preferable that a "Forward-over-Reverse" scheme is used. Thus the hessians are evaluated in reverse (starting with the last block on the tape).

`evaluate_tlm_component`(*inputs*, *tlm_inputs*, *block_variable*, *idx*, *prepared=None*)

This method should be overridden.

The method should implement a routine for computing the tangent linear model of the block that corresponds to one output. If one considers the tangent linear action as a Jacobian matrix multiplied with a vector, then this method should return one entry in the resulting product, where the entry returned is decided by the argument *idx*.

Args:
inputs (list): A list of the saved input values, determined by the dependencies list. tlm_inputs (list): A list of the tlm input values, determined by the dependencies list. block_variable (BlockVariable): The block variable of the output corresponding to index *idx*. idx (int): The index of the component to compute. prepared (object): Anything returned by the prepare_evaluate_tlm method. Default is None.

Returns:
An object of the same type as *block_variable.saved_output*: The resulting product.

`recompute_component`(*inputs*, *block_variable*, *idx*, *prepared*)

This method must be overridden.

The method should implement a routine for recomputing one output of the block in the forward computations. The output to recompute is determined by the *idx* argument, which corresponds to the index of the output in the outputs list. If the block only has a single output, then *idx* will always be 0.

Args:
inputs (list): A list of the saved input values, determined by the dependencies list.

block_variable (BlockVariable): The block variable of the output corresponding to index *idx*. idx (int): The index of the output to compute. prepared (object): Anything returned by the prepare_recompute_component method. Default is None.

Returns:
An object of the same type as *block_variable.checkpoint* which is determined by *OverloadedType._ad_create_checkpoint* (often the same as *block_variable.saved_output*): The new output.

class `firedrake.adjoint.blocks.`**`GenericSolveBlock`**(*lhs*, *rhs*, *func*, *bcs*, **args*, ***kwargs*)

Bases: `GenericSolveBlock`, *`Backend`*

`block_helper`

`recompute_component`(*inputs*, *block_variable*, *idx*, *prepared*)

This method must be overridden.

The method should implement a routine for recomputing one output of the block in the forward computations. The output to recompute is determined by the *idx* argument, which corresponds to the index of the output in the outputs list. If the block only has a single output, then *idx* will always be 0.

Args:
inputs (list): A list of the saved input values, determined by the dependencies list. block_variable (BlockVariable): The block variable of the output corresponding to index *idx*. idx (int): The index of the output to compute. prepared (object): Anything returned by the prepare_recompute_component method. Default is None.

Returns:
An object of the same type as *block_variable.checkpoint* which is determined by *OverloadedType._ad_create_checkpoint* (often the same as *block_variable.saved_output*): The new output.

class `firedrake.adjoint.blocks.`**`InterpolateBlock`**(*interpolator*, **functions*, ***kwargs*)

Bases: `Block`, *`Backend`*

Annotates an interpolator.

Consider the block as f with 1 forward model output `v`, and inputs `u` and `g` (there can, in principle, be any number of outputs). The adjoint input is `vhat` (`uhat` and `ghat` are adjoints to `u` and `v` respectively and are shown for completeness). The downstream block is `J` which has input `v`.

```
 _            _
|J|--<--v--<--|f|--<--u--<--...
 -      |     -      |
       vhat   |     uhat
              |
               ---<--g--<--...
                     |
                    ghat
```

(Arrows indicate forward model direction)

```
J : V → R i.e. J(v) ∈ R ∀ v ∈ V
```

Interpolation can operate on an expression which may not be linear in its arguments.

```
f : W × G → V i.e. f(u, g) ∈ V ∀ u ∈ W and g ∈ G.
f = I ∘ expr
I :   X → V i.e. I(;x) ∈ V ∀ x ∈ X.
                  X is infinite dimensional.
expr: W × G → X i.e. expr(u, g) ∈ X ∀ u ∈ W and g ∈ G.
```

Arguments after a semicolon are linear (i.e. operation I is linear)

block_helper

evaluate_adj_component(*inputs*, *adj_inputs*, *block_variable*, *idx*, *prepared=None*)

Denote `d_u[A]` as the gateaux derivative in the `u` direction. Arguments after a semicolon are linear.

This calculates

```
uhat = vhat · d_u[f](u, g; ·) (for inputs[idx] ∈ W)
or
ghat = vhat · d_g[f](u, g; ·) (for inputs[idx] ∈ G)
```

where `inputs[idx]` specifies the derivative direction, `vhat` is `adj_inputs[0]` (since we assume only one block output) and · denotes an unspecified operand of `u'` (for `inputs[idx]` ∈ `W`) or `g'` (for `inputs[idx]` ∈ `G`) size (`vhat` left multiplies the derivative).

```
f = I ∘ expr : W × G → V
               i.e. I(expr|_{u, g}) ∈ V ∀ u ∈ W, g ∈ G.
```

Since `I` is linear we get that

```
d_u[I ∘ expr](u, g; u') = I ∘ d_u[expr](u|_u, g|_g; u')
d_g[I ∘ expr](u, g; g') = I ∘ d_u[expr](u|_u, g|_g; g').
```

In tensor notation

```
uhat_q^T = vhat_p^T I([dexpr/du|_u]_q)_p
or
ghat_q^T = vhat_p^T I([dexpr/dg|_u]_q)_p
```

the output is then

```
uhat_q = I^T([dexpr/du|_u]_q)_p vhat_p
or
ghat_q = I^T([dexpr/dg|_u]_q)_p vhat_p.
```

evaluate_hessian_component(*inputs*, *hessian_inputs*, *adj_inputs*, *block_variable*, *idx*, *relevant_dependencies*, *prepared=None*)

Denote `d_u[A]` as the gateaux derivative in the `u` direction. Arguments after a semicolon are linear.

hessian_input is `d_v[d_v[J]](v; v', ·)` where the direction · is left unspecified so it can be operated upon.

> **Warning:** NOTE: This comment describes the implementation of 1 block input `u`. (e.g. interpolating from an expression with 1 coefficient). Explaining how this works for multiple block inputs (e.g. `u` and `g`) is currently too complicated for the author to do succinctly!

This function needs to output `d_u[d_u[J ∘ f]](u; u', ·)` where the direction · will be specified in another function and multiplied on the right with the output of this function. We will calculate this using the chain rule.

```
J : V ⟶ R i.e. J(v) ∈ R ∀ v ∈ V
f = I ∘ expr : W ⟶ V
J ∘ f : W ⟶ R i.e. J(f|u) ∈ R ∀ u ∈ V.
d_u[J ∘ f] : W × W ⟶ R i.e. d_u[J ∘ f](u; u')
d_u[d_u[J ∘ f]] : W × W × W ⟶ R i.e. d_u[d_u[J ∘ f]](u; u', u'')
d_v[J] : V × V ⟶ R i.e. d_v[J](v; v')
d_v[d_v[J]] : V × V × V ⟶ R i.e. d_v[d_v[J]](v; v', v'')
```

Chain rule:

```
d_u[J ∘ f](u; u') = d_v[J](v = f|u; v' = d_u[f](u; u'))
```

Multivariable chain rule:

```
d_u[d_u[J ∘ f]](u; u', u'') =
d_v[d_v[J]](v = f|u; v' = d_u[f](u; u'), v'' = d_u[f](u; u''))
+ d_v'[d_v[J]](v = f|u; v' = d_u[f](u; u'), v'' = d_u[d_u[f]](u; u',␣
↪u''))
= d_v[d_v[J]](v = f|u; v' = d_u[f](u; u'), v''=d_u[f](u; u''))
+ d_v[J](v = f|u; v' = v'' = d_u[d_u[f]](u; u', u''))
```

since `d_v[d_v[J]]` is linear in `v'` so differentiating wrt to it leaves its coefficient, the bare d_v[J] operator which acts on the `v''` term that remains.

The `d_u[d_u[f]](u; u', u'')` term can be simplified further:

```
f = I ∘ expr : W ⟶ V i.e. I(expr|u) ∈ V ∀ u ∈ W
d_u[I ∘ expr] : W × W ⟶ V i.e. d_u[I ∘ expr](u; u')
d_u[d_u[I ∘ expr]] : W × W × W ⟶ V i.e. d_u[I ∘ expr](u; u', u'')
d_x[I] : X × X ⟶ V i.e. d_x[I](x; x')
d_x[d_x[I]] : X × X × X ⟶ V i.e. d_x[d_x[I]](x; x', x'')
d_u[expr] : W × W ⟶ X i.e. d_u[expr](u; u')
d_u[d_u[expr]] : W × W × W ⟶ X i.e. d_u[d_u[expr]](u; u', u'')
```

Since `I` is linear we get that

```
d_u[d_u[I ∘ expr]](u; u', u'') = I ∘ d_u[d_u[expr]](u; u', u'').
```

So our full hessian is:

```
d_u[d_u[J ∘ f]](u; u', u'')
= d_v[d_v[J]](v = f|u; v' = d_u[f](u; u'), v''=d_u[f](u; u''))
+ d_v[J](v = f|u; v' = v'' = d_u[d_u[f]](u; u', u''))
```

In tensor notation

```
[d^2[J ∘ f]/du^2|_u]_{lk} u'_k u''_k =
[d^2J/dv^2|_{v=f|_u}]_{ij} [df/du|_u]_{jk} u'_k [df/du|_u]_{il} u''_l
+ [dJ/dv|_{v=f_u}]_i I([d^2expr/du^2|_u]_{lk} u'_k)_i u''_l
```

In the first term:

```
[df/du|_u]_{jk} u'_k = v'_j
=> [d^2J/dv^2|_{v=f|_u}]_{ij} [df/du|_u]_{jk} u'_k
= [d^2J/dv^2|_{v=f|_u}]_{ij} v'_j
= hessian_input_i
=> [d^2J/dv^2|_{v=f|_u}]_{ij} [df/du|_u]_{jk} u'_k [df/du|_u]_{il}
= hessian_input_i [df/du|_u]_{il}
= self.evaluate_adj_component(inputs, hessian_inputs, ...)_l
```

In the second term we calculate everything explicitly though note `[dJ/dv|_{v=f_u}]_i = adj_inputs[0]_i`

Also, the second term is 0 if `expr` is linear.

evaluate_tlm_component(*inputs*, *tlm_inputs*, *block_variable*, *idx*, *prepared=None*)

Denote `d_u[A]` as the gateaux derivative in the `u` direction. Arguments after a semicolon are linear.

For a block with two inputs this calculates

```
v' = d_u[f](u, g; u') + d_g[f](u, g; g')
```

where `u' = tlm_inputs[0]` and `g = tlm_inputs[1]`.

```
f = I ∘ expr : W × G → V
                i.e. I(expr|_{u, g}) ∈ V ∀ u ∈ W, g ∈ G.
```

Since `I` is linear we get that

```
d_u[I ∘ expr](u, g; u') = I ∘ d_u[expr](u|_u, g|_g; u')
d_g[I ∘ expr](u, g; g') = I ∘ d_u[expr](u|_u, g|_g; g').
```

In tensor notation the output is then

```
v'_l = I([dexpr/du|_{u,g}]_k u'_k)_l + I([dexpr/du|_{u,g}]_k g'_k)_l
     = I([dexpr/du|_{u,g}]_k u'_k + [dexpr/du|_{u,g}]_k g'_k)_l
```

since `I` is linear.

prepare_evaluate_adj(*inputs*, *adj_inputs*, *relevant_outputs*)

Runs preparations before *evalute_adj_component* is ran.

The return value is supplied to each of the subsequent *evaluate_adj_component* calls. This method is intended to be overridden for blocks that require such preparations, by default there is none.

Args:
inputs: The values of the inputs adj_inputs: The adjoint inputs relevant_dependencies: A list of the relevant block variables for *evaluate_adj_component*.

Returns:
Anything. The returned value is supplied to *evaluate_adj_component*

prepare_evaluate_hessian(*inputs*, *hessian_inputs*, *adj_inputs*, *relevant_dependencies*)

Runs preparations before *evalute_hessian_component* is ran for each relevant dependency.

The return value is supplied to each of the subsequent *evaluate_hessian_component* calls. This method is intended to be overridden for blocks that require such preparations, by default there is none.

Args:
inputs: The values of the inputs hessian_inputs: The hessian inputs adj_inputs: The adjoint inputs relevant_dependencies: A list of the relevant block variables for *evaluate_hessian_component*.

Returns:
Anything. The returned value is supplied to *evaluate_hessian_component*

prepare_evaluate_tlm(*inputs*, *tlm_inputs*, *relevant_outputs*)

Runs preparations before *evalute_tlm_component* is ran.

The return value is supplied to each of the subsequent *evaluate_tlm_component* calls. This method is intended to be overridden for blocks that require such preparations, by default there is none.

Args:
inputs: The values of the inputs tlm_inputs: The tlm inputs relevant_outputs: A list of the relevant block variables for *evaluate_tlm_component*.

Returns:
Anything. The returned value is supplied to *evaluate_tlm_component*

prepare_recompute_component(*inputs*, *relevant_outputs*)

Runs preparations before *recompute_component* is ran.

The return value is supplied to each of the subsequent *recompute_component* calls. This method is intended to be overridden for blocks that require such preparations, by default there is none.

Args:
inputs: The values of the inputs relevant_outputs: A list of the relevant block variables for *recompute_component*.

Returns:
Anything. The returned value is supplied to *recompute_component*

recompute_component(*inputs*, *block_variable*, *idx*, *prepared*)

This method must be overridden.

The method should implement a routine for recomputing one output of the block in the forward computations. The output to recompute is determined by the *idx* argument, which corresponds to the index of the output in the outputs list. If the block only has a single output, then *idx* will always be 0.

Args:
inputs (list): A list of the saved input values, determined by the dependencies list. block_variable (BlockVariable): The block variable of the output corresponding to index *idx*. idx (int): The index of the output to compute. prepared (object): Anything returned by the prepare_recompute_component method. Default is None.

Returns:
An object of the same type as *block_variable.checkpoint* which is determined by *OverloadedType._ad_create_checkpoint* (often the same as *block_variable.saved_output*): The new output.

class firedrake.adjoint.blocks.**MeshInputBlock**(*mesh*, *ad_block_tag=None*)

Bases: `Block`

Block which links a MeshGeometry to its coordinates, which is a firedrake function.

block_helper

evaluate_adj_component(*inputs*, *adj_inputs*, *block_variable*, *idx*, *prepared=None*)

This method should be overridden.

The method should implement a routine for evaluating the adjoint of the block that corresponds to one dependency. If one considers the adjoint action a vector right multiplied with the Jacobian matrix, then this method should return one entry in the resulting product, where the entry returned is decided by the argument *idx*.

Args:
inputs (list): A list of the saved input values, determined by the dependencies list. adj_inputs (list): A list of the adjoint input values, determined by the outputs list. block_variable (BlockVariable): The block variable of the dependency corresponding to index *idx*. idx (int): The index of the component to compute. prepared (object): Anything returned by the prepare_evaluate_adj method. Default is None.

Returns:
An object of a type consistent with the adj_value type of *block_variable*: The resulting product.

evaluate_hessian_component(*inputs*, *hessian_inputs*, *adj_inputs*, *idx*, *block_variable*, *relevant_dependencies*, *prepared=None*)

This method must be overridden.

The method should implement a routine for evaluating the hessian of the block. It is preferable that a “Forward-over-Reverse” scheme is used. Thus the hessians are evaluated in reverse (starting with the last block on the tape).

evaluate_tlm_component(*inputs*, *tlm_inputs*, *block_variable*, *idx*, *prepared=None*)

This method should be overridden.

The method should implement a routine for computing the tangent linear model of the block that corresponds to one output. If one considers the tangent linear action as a Jacobian matrix multiplied with a vector, then this method should return one entry in the resulting product, where the entry returned is decided by the argument *idx*.

Args:

inputs (list): A list of the saved input values, determined by the dependencies list. tlm_inputs (list): A list of the tlm input values, determined by the dependencies list. block_variable (BlockVariable): The block variable of the output corresponding to index *idx*. idx (int): The index of the component to compute. prepared (object): Anything returned by the prepare_evaluate_tlm method. Default is None.

Returns:

An object of the same type as *block_variable.saved_output*: The resulting product.

recompute_component(*inputs*, *block_variable*, *idx*, *prepared*)

This method must be overridden.

The method should implement a routine for recomputing one output of the block in the forward computations. The output to recompute is determined by the *idx* argument, which corresponds to the index of the output in the outputs list. If the block only has a single output, then *idx* will always be 0.

Args:

inputs (list): A list of the saved input values, determined by the dependencies list. block_variable (BlockVariable): The block variable of the output corresponding to index *idx*. idx (int): The index of the output to compute. prepared (object): Anything returned by the prepare_recompute_component method. Default is None.

Returns:

An object of the same type as *block_variable.checkpoint* which is determined by *OverloadedType._ad_create_checkpoint* (often the same as *block_variable.saved_output*): The new output.

class firedrake.adjoint.blocks.**MeshOutputBlock**(*func*, *mesh*, *ad_block_tag=None*)

Bases: `Block`

Block which is called when the coordinates of a mesh are changed.

block_helper

evaluate_adj_component(*inputs*, *adj_inputs*, *block_variable*, *idx*, *prepared=None*)

This method should be overridden.

The method should implement a routine for evaluating the adjoint of the block that corresponds to one dependency. If one considers the adjoint action a vector right multiplied with the Jacobian matrix, then this method should return one entry in the resulting product, where the entry returned is decided by the argument *idx*.

Args:

inputs (list): A list of the saved input values, determined by the dependencies list. adj_inputs (list): A list of the adjoint input values, determined by the outputs list. block_variable (BlockVariable): The block variable of the dependency corresponding to index *idx*. idx (int): The index of the component to compute. prepared (object): Anything returned by the prepare_evaluate_adj method. Default is None.

Returns:

An object of a type consistent with the adj_value type of *block_variable*: The resulting product.

`evaluate_hessian_component`(*inputs*, *hessian_inputs*, *adj_inputs*, *idx*, *block_variable*, *relevant_dependencies*, *prepared=None*)

This method must be overridden.

The method should implement a routine for evaluating the hessian of the block. It is preferable that a "Forward-over-Reverse" scheme is used. Thus the hessians are evaluated in reverse (starting with the last block on the tape).

`evaluate_tlm_component`(*inputs*, *tlm_inputs*, *block_variable*, *idx*, *prepared=None*)

This method should be overridden.

The method should implement a routine for computing the tangent linear model of the block that corresponds to one output. If one considers the tangent linear action as a Jacobian matrix multiplied with a vector, then this method should return one entry in the resulting product, where the entry returned is decided by the argument *idx*.

Args:

inputs (list): A list of the saved input values, determined by the dependencies list. tlm_inputs (list): A list of the tlm input values, determined by the dependencies list. block_variable (BlockVariable): The block variable of the output corresponding to index *idx*. idx (int): The index of the component to compute. prepared (object): Anything returned by the prepare_evaluate_tlm method. Default is None.

Returns:

An object of the same type as *block_variable.saved_output*: The resulting product.

`recompute_component`(*inputs*, *block_variable*, *idx*, *prepared*)

This method must be overridden.

The method should implement a routine for recomputing one output of the block in the forward computations. The output to recompute is determined by the *idx* argument, which corresponds to the index of the output in the outputs list. If the block only has a single output, then *idx* will always be 0.

Args:

inputs (list): A list of the saved input values, determined by the dependencies list. block_variable (BlockVariable): The block variable of the output corresponding to index *idx*. idx (int): The index of the output to compute. prepared (object): Anything returned by the prepare_recompute_component method. Default is None.

Returns:
An object of the same type as *block_variable.checkpoint* which is determined by *OverloadedType._ad_create_checkpoint* (often the same as *block_variable.saved_output*): The new output.

`class firedrake.adjoint.blocks.`**`NonlinearVariationalSolveBlock`**`(`*equation*, *func*, *bcs*, *adj_F*, *dFdm_cache*, *problem_J*, *solver_params*, *solver_kwargs*, ***kwargs*`)`

Bases: *GenericSolveBlock*

`block_helper`

`evaluate_adj_component``(`*inputs*, *adj_inputs*, *block_variable*, *idx*, *prepared=None*`)`

This method should be overridden.

The method should implement a routine for evaluating the adjoint of the block that corresponds to one dependency. If one considers the adjoint action a vector right multiplied with the Jacobian matrix, then this method should return one entry in the resulting product, where the entry returned is decided by the argument *idx*.

Args:
inputs (list): A list of the saved input values, determined by the dependencies list. adj_inputs (list): A list of the adjoint input values, determined by the outputs list. block_variable (BlockVariable): The block variable of the dependency corresponding to index *idx*. idx (int): The index of the component to compute. prepared (object): Anything returned by the prepare_evaluate_adj method. Default is None.

Returns:
An object of a type consistent with the adj_value type of *block_variable*: The resulting product.

`prepare_evaluate_adj``(`*inputs*, *adj_inputs*, *relevant_dependencies*`)`

Runs preparations before *evalute_adj_component* is ran.

The return value is supplied to each of the subsequent *evaluate_adj_component* calls. This method is intended to be overridden for blocks that require such preparations, by default there is none.

Args:
inputs: The values of the inputs adj_inputs: The adjoint inputs relevant_dependencies: A list of the relevant block variables for *evaluate_adj_component*.

Returns:
Anything. The returned value is supplied to *evaluate_adj_component*

`class firedrake.adjoint.blocks.`**`ProjectBlock`**`(`*v*, *V*, *output*, *bcs=[]*, **args*, ***kwargs*`)`

Bases: *SolveVarFormBlock*

`block_helper`

class firedrake.adjoint.blocks.**SolveLinearSystemBlock**(*A*, *u*, *b*, **args*, ***kwargs*)

Bases: *GenericSolveBlock*

block_helper

class firedrake.adjoint.blocks.**SolveVarFormBlock**(*equation*, *func*, *bcs=[]*, **args*, ***kwargs*)

Bases: *GenericSolveBlock*

block_helper

class firedrake.adjoint.blocks.**SupermeshProjectBlock**(*source*, *target_space*, *target*, *bcs=[]*, ***kwargs*)

Bases: Block, *Backend*

Annotates supermesh projection.

Suppose we have a source space, V_A, and a target space, V_B. Projecting a source from V_A to V_B amounts to solving the linear system

$$M_B * v_B = M_{AB} * v_A,$$

where

- M_B is the mass matrix on V_B,
- M_{AB} is the mixed mass matrix for V_A and V_B,
- v_A and v_B are vector representations of the source and target *Function* s.

This can be broken into two steps:

Step 1. form RHS, multiplying the source with the mixed mass matrix;

Step 2. solve linear system.

apply_mixedmass(*a*)

block_helper

evaluate_adj_component(*inputs*, *adj_inputs*, *block_variable*, *idx*, *prepared=None*)

Recall that the forward propagation can be broken down as

Step 1. multiply $w := M_{AB} * v_A$; Step 2. solve $M_B * v_B = w$.

For a seed vector v_B^{seed} from the target space, the adjoint is given by

Adjoint of step 2. solve $M_B^T * w = v_B^{seed}$ for *w*; Adjoint of step 1. multiply $v_A^{adj} := M_{AB}^T * w$.

evaluate_hessian_component(*inputs*, *hessian_inputs*, *adj_inputs*, *block_variable*, *idx*, *relevant_dependencies*, *prepared=None*)

This method must be overridden.

The method should implement a routine for evaluating the hessian of the block. It is preferable that a “Forward-over-Reverse” scheme is used. Thus the hessians are evaluated in reverse (starting with the last block on the tape).

evaluate_tlm_component(*inputs*, *tlm_inputs*, *block_variable*, *idx*, *prepared=None*)

Given that the input is a *Function*, we just have a linear operation. As such, the tlm is just the sum of each tlm input projected into the target space.

recompute_component(*inputs*, *block_variable*, *idx*, *prepared*)

This method must be overridden.

The method should implement a routine for recomputing one output of the block in the forward computations. The output to recompute is determined by the *idx* argument, which corresponds to the index of the output in the outputs list. If the block only has a single output, then *idx* will always be 0.

Args:

inputs (list): A list of the saved input values, determined by the dependencies list. block_variable (BlockVariable): The block variable of the output corresponding to index *idx*. idx (int): The index of the output to compute. prepared (object): Anything returned by the prepare_recompute_component method. Default is None.

Returns:

An object of the same type as *block_variable.checkpoint* which is determined by *OverloadedType._ad_create_checkpoint* (often the same as *block_variable.saved_output*): The new output.

firedrake.adjoint.blocks.**solve_init_params**(*self*, *args*, *kwargs*, *varform*)

firedrake.adjoint.checkpointing module

A module providing support for disk checkpointing of the adjoint tape.

firedrake.adjoint.checkpointing.**checkpointable_mesh**(*mesh*)

Write a mesh to disk and read it back.

Since a mesh will be repartitioned by being written to disk and reread, only meshes read from a checkpoint file are safe to use with disk checkpointing.

The workflow for disk checkpointing is therefore to create the mesh(es) required, and then call this function on them. Only the mesh(es) returned by this function can be used in disk checkpointing.

Parameters

mesh
[firedrake.Mesh] The mesh to be checkpointed.

Returns

firedrake.Mesh
The checkpointed mesh to be used in the rest of the computation.

firedrake.adjoint.checkpointing.**continue_disk_checkpointing**()

Resume disk checkpointing.

firedrake.adjoint.checkpointing.**disk_checkpointing**()

Return true if disk checkpointing of the adjoint tape is active.

firedrake.adjoint.checkpointing.**enable_disk_checkpointing**(*dirname=None*, *comm=<mpi4py.MPI.Intracomm object>*, *cleanup=True*)

Add a DiskCheckpointer to the current tape and switch on disk checkpointing.

Parameters

dirname
[str] The directory in which the disk checkpoints should be stored. If not specified then the current working directory is used. Checkpoints are stored in a temporary subdirectory of this directory.

comm
[mpi4py.MPI.Intracomm] The MPI communicator over which the computation to be disk checkpointed is defined. This will usually match the communicator on which the mesh(es) are defined.

cleanup
[bool] If set to False, checkpoint files will not be deleted when no longer required. This is usually only useful for debugging.

`firedrake.adjoint.checkpointing.`**`pause_disk_checkpointing`**`()`

Pause disk checkpointing and instead checkpoint to memory.

class `firedrake.adjoint.checkpointing.`**`stop_disk_checkpointing`**

Bases: `object`

A context manager inside which disk checkpointing is paused.

firedrake.adjoint.constant module

class `firedrake.adjoint.constant.`**`ConstantMixin`**`(`**args*, ***kwargs*`)`

Bases: `OverloadedType`

`get_derivative``(`*options={}*`)`

firedrake.adjoint.dirichletbc module

class `firedrake.adjoint.dirichletbc.`**`DirichletBCMixin`**`(`**args*, ***kwargs*`)`

Bases: `FloatingType`

firedrake.adjoint.function module

class `firedrake.adjoint.function.`**`DelegatedFunctionCheckpoint`**`(`*other*`)`

Bases: `CheckpointBase`

A wrapper which delegates the checkpoint of this Function to another Function.

This enables us to avoid checkpointing a Function twice when it is copied.

Parameters

other: BlockVariable
The block variable to which we delegate checkpointing.

`restore``()`

Recover and return the checkpointed object.

class `firedrake.adjoint.function.`**`FunctionMixin`**`(`**args*, ***kwargs*`)`

Bases: `FloatingType`

firedrake.adjoint.interpolate module

`firedrake.adjoint.interpolate.`**`annotate_interpolate`**`(`*`interpolate`*`)`

firedrake.adjoint.mesh module

`class` `firedrake.adjoint.mesh.`**`MeshGeometryMixin`**`(`*`*args`*`, `*`**kwargs`*`)`

Bases: `OverloadedType`

firedrake.adjoint.projection module

`firedrake.adjoint.projection.`**`annotate_project`**`(`*`project`*`)`

firedrake.adjoint.solving module

`firedrake.adjoint.solving.`**`annotate_solve`**`(`*`solve`*`)`

This solve routine wraps the Firedrake *`solve()`* call. Its purpose is to annotate the model, recording what solves occur and what forms are involved, so that the adjoint and tangent linear models may be constructed automatically by pyadjoint.

To disable the annotation, just pass `annotate=False` to this routine, and it acts exactly like the Firedrake solve call. This is useful in cases where the solve is known to be irrelevant or diagnostic for the purposes of the adjoint computation (such as projecting fields to other function spaces for the purposes of visualisation).

The overloaded solve takes optional callback functions to extract adjoint solutions. All of the callback functions follow the same signature, taking a single argument of type Function.

Keyword Args:

adj_cb (function, optional): callback function supplying the adjoint solution in the interior.
The boundary values are zero.

adj_bdy_cb (function, optional): callback function supplying the adjoint solution on the boundary.
The interior values are not guaranteed to be zero.

adj2_cb (function, optional): callback function supplying the second-order adjoint solution in the interior.
The boundary values are zero.

adj2_bdy_cb (function, optional): callback function supplying the second-order adjoint solution on
the boundary. The interior values are not guaranteed to be zero.

ad_block_tag (string, options): tag used to label the resulting block on the Pyadjoint tape. This
is useful for identifying which block is associated with which equation in the forward model.

`firedrake.adjoint.solving.`**`get_solve_blocks`**`()`

Extract all blocks of the tape which correspond to PDE solves, except for those which correspond to calls of the `project` operator.

firedrake.adjoint.variational_solver module

class `firedrake.adjoint.variational_solver.`**`NonlinearVariationalProblemMixin`**

Bases: `object`

class `firedrake.adjoint.variational_solver.`**`NonlinearVariationalSolverMixin`**

Bases: `object`

Module contents

4.1.2 firedrake.cython package

Submodules

firedrake.cython.dmcommon module

`firedrake.cython.dmcommon.`**`cell_facet_labeling`**`()`

Computes a labeling for the facet numbers on a particular cell (interior and exterior facet labels with subdomain markers). The i-th local facet is represented as:

cell_facets[c, i]

If *cell_facets[c, i, 0]* is `0`, then the local facet `i` is an exterior facet, otherwise if the result is `1` it is interior. *cell_facets[c, i, 1]* returns the subdomain marker for the local facet.

Parameters

- **`plex`** – The DMPlex object representing the mesh topology.
- **`cell_numbering`** – PETSc.Section describing the global cell numbering
- **`cell_closures`** – 2D array of ordered cell closures.

`firedrake.cython.dmcommon.`**`clear_adjacency_callback`**`()`

Clear the callback for DMPlexGetAdjacency.

Parameters

`dm` – The DMPlex object

`firedrake.cython.dmcommon.`**`closure_ordering`**`()`

Apply Fenics local numbering to a cell closure.

Parameters

- **`dm`** – The DM object encapsulating the mesh topology
- **`vertex_numbering`** – Section describing the universal vertex numbering
- **`cell_numbering`** – Section describing the global cell numbering

- **entity_per_cell** – List of the number of entity points in each dimension

Vertices := Ordered according to global/universal
vertex numbering

Edges/faces := Ordered according to lexicographical
ordering of non-incident vertices

firedrake.cython.dmcommon.**complete_facet_labels**()

Transfer label values from the facet labels to everything in the closure of the facets.

firedrake.cython.dmcommon.**compute_point_cone_global_sizes**()

Compute the total number of DMPlex points and the global sum of cone sizes for dm.

Parameters
dm – The dm.

Returns
A list of global number of points and global sum of cone sizes.

firedrake.cython.dmcommon.**count_labelled_points**()

Return the number of points in the chart [start, end) that are marked by the given label.

Note: This destroys any index on the label.

Parameters

- **dm** – The DM containing the label
- **name** – The label name.
- **start** – The smallest point to consider.
- **end** – One past the largest point to consider.

firedrake.cython.dmcommon.**create_cell_closure**()

Create a map from FIAT local entity numbers to DMPlex point numbers for each cell.

Parameters

- **dm** – The DM object encapsulating the mesh topology
- **cell_numbering** – Section describing the global cell numbering
- **_closureSize** – Number of entities in the cell

firedrake.cython.dmcommon.**create_section**()

Create the section describing a global numbering.

Parameters

- **mesh** – The mesh.
- **nodes_per_entity** – Number of nodes on each type of topological entity of the mesh. Or, if the mesh is extruded, the number of nodes on, and on top of, each topological entity in the base mesh.

- **on_base** – If True, assume extruded space is actually Foo x Real.
- **block_size** – The integer by which nodes_per_entity is uniformly multiplied to get the true data layout.

Returns
A PETSc Section providing the number of dofs, and offset of each dof, on each mesh point.

`firedrake.cython.dmcommon.`**`entity_orientations`**`()`

Compute entity orientations.

Parameters

- **mesh** – The *MeshTopology* object encapsulating the mesh topology
- **cell_closure** – The two-dimensional array, each row of which contains the closure of the associated cell

Returns
A 2D array of the same shape as cell_closure, each row of which contains orientations of the entities in the closure of the associated cell

See `entity_orientations()` for details on the returned array.

See *get_cell_nodes* for the usage of the returned array.

`firedrake.cython.dmcommon.`**`exchange_cell_orientations`**`()`

Halo exchange of cell orientations.

Parameters

- **plex** – The DMPlex object encapsulating the mesh topology
- **section** – Section describing the cell numbering
- **orientations** – Cell orientations to exchange, values in the halo will be overwritten.

`firedrake.cython.dmcommon.`**`facet_closure_nodes`**`()`

Extract nodes in the closure of facets with a given marker.

This works fine for interior as well as exterior facets.

Parameters

- **V** – the function space
- **sub_domain** – a tuple of mesh markers selecting the facets, or the magic string “on_boundary” indicating the entire boundary.

Returns
a numpy array of unique nodes in the closure of facets with the given marker.

`firedrake.cython.dmcommon.`**`facet_numbering`**`()`

Compute the parent cell(s) and the local facet number within each parent cell for each given facet.

Parameters

- **plex** – The DMPlex object encapsulating the mesh topology

- **kind** – String indicating the facet kind (interior or exterior)
- **facets** – Array of input facets
- **cell_numbering** – Section describing the global cell numbering
- **cell_closures** – 2D array of ordered cell closures

firedrake.cython.dmcommon.**fill_reference_coordinates_function**()

Fill the PyOP2 dat of an input vector valued function on a VertexOnlyMesh *reference_coordinates_f* with the reference coordinates of each vertex in their relevant reference cells.

Parameters

reference_coordinates_f – A vector valued function on a VertexOnlyMesh (with vector dimension the topological dimension of the parent mesh) which will have its dat modified.

Returns

The updated *reference_coordinates_f*.

firedrake.cython.dmcommon.**get_cell_markers**()

Get the cells marked by a given subdomain_id.

Parameters

- **dm** – The DM for the mesh topology
- **cell_numbering** – Section mapping dm cell points to firedrake cell indices.
- **subdomain_id** – The subdomain_id to look for.

Raises

ValueError – if the subdomain_id is not valid.

Returns

A numpy array (possibly empty) of the cell ids.

firedrake.cython.dmcommon.**get_cell_nodes**()

Builds the DoF mapping.

Parameters

- **mesh** – The mesh
- **global_numbering** – Section describing the global DoF numbering
- **entity_dofs** – FInAT element entity dofs for the cell
- **entity_permutations** – FInAT element entity permutations for the cell
- **offset** – offsets for each entity dof walking up a column.

Preconditions: This function assumes that cell_closures contains mesh entities ordered by dimension, i.e. vertices first, then edges, faces, and finally the cell. For quadrilateral meshes, edges corresponding to dimension (0, 1) in the FInAT element must precede edges corresponding to dimension (1, 0) in the FInAT element.

`firedrake.cython.dmcommon.`**`get_cell_remote_ranks()`**

Returns an array assigning the rank of the owner to each locally visible cell. Locally owned cells have -1 assigned to them.

Parameters
`plex` – The DMPlex object encapsulating the mesh topology

`firedrake.cython.dmcommon.`**`get_entity_classes()`**

Builds PyOP2 entity class offsets for all entity levels.

Parameters
`dm` – The DM object encapsulating the mesh topology

`firedrake.cython.dmcommon.`**`get_facet_markers()`**

Get an array of facet labels in the mesh.

Parameters

- **`dm`** – The DM that contains labels.
- **`facets`** – The array of facet points.

Returns
a numpy array of facet ids (or None if all facets had the default marker).

`firedrake.cython.dmcommon.`**`get_facet_nodes()`**

Build to DoF mapping from facets.

Parameters

- **`mesh`** – The mesh.
- **`cell_nodes`** – numpy array mapping from cells to function space nodes.
- **`label`** – which set of facets to ask for (interior_facets or exterior_facets).
- **`offset`** – optional offset (extruded only).

Returns
numpy array mapping from facets to nodes in the closure of the support of that facet.

`firedrake.cython.dmcommon.`**`get_facet_ordering()`**

Builds a list of all facets ordered according to the given numbering.

Parameters

- **`plex`** – The DMPlex object encapsulating the mesh topology
- **`facet_numbering`** – A Section describing the global facet numbering

`firedrake.cython.dmcommon.`**`get_facets_by_class()`**

Builds a list of all facets ordered according to PyOP2 entity classes and computes the respective class offsets.

Parameters

- **`plex`** – The DMPlex object encapsulating the mesh topology
- **`ordering`** – An array giving the global traversal order of facets
- **`label`** – Label string that marks the facets to order

`firedrake.cython.dmcommon.`**`get_topological_dimension`**`()`

Get the topological dimension of a DMPlex or DMSwarm. Assumes that a DMSwarm represents a mesh of vertices (tdim 0).

Parameters
`dm` – A DMPlex or DMSwarm.

Returns
For a DMPlex `dm.getDimension()`, for a DMSwarm ``0.`

`firedrake.cython.dmcommon.`**`label_facets`**`()`

Add labels to facets in the the plex

Facets on the boundary are marked with "exterior_facets" while all others are marked with "interior_facets".

Parameters
`label_boundary` – if False, don't label the boundary faces (they must have already been labelled).

`firedrake.cython.dmcommon.`**`make_global_numbering`**`()`

Build an array of global numbers for local dofs

Parameters

- **`lsec`** – Section describing local dof layout and numbers.
- **`gsec`** – Section describing global dof layout and numbers.

`firedrake.cython.dmcommon.`**`mark_entity_classes`**`()`

Mark all points in a given DM according to the PyOP2 entity classes:

core : owned and not in send halo owned : owned and in send halo ghost : in halo

by inspecting the *pointSF* graph.

Parameters
`dm` – The DM object encapsulating the mesh topology

`firedrake.cython.dmcommon.`**`orientations_facet2cell`**`()`

Converts local quadrilateral facet orientations into global quadrilateral cell orientations.

Parameters

- **`plex`** – The DMPlex object encapsulating the mesh topology
- **`vertex_numbering`** – Section describing the universal vertex numbering
- **`facet_orientations`** – Facet orientations (edge directions) relative to the local DMPlex ordering.
- **`cell_numbering`** – Section describing the cell numbering

`firedrake.cython.dmcommon.`**`plex_renumbering`**`()`

Build a global node renumbering as a permutation of Plex points.

Parameters

- **`plex`** – The DMPlex object encapsulating the mesh topology
- **`entity_classes`** – Array of entity class offsets for each dimension.

- **reordering** – A reordering from reordered to original plex points used to provide the traversal order of the cells (i.e. the inverse of the ordering obtained from DMPlexGetOrdering). Optional, if not provided (or `None`), no reordering is applied and the plex is traversed in original order.

The node permutation is derived from a depth-first traversal of the Plex graph over each entity class in turn. The returned IS is the Plex -> PyOP2 permutation.

`firedrake.cython.dmcommon.`**`prune_sf`**`()`

Prune an SF of roots referencing the local rank

Parameters
sf – The PETSc SF to prune.

`firedrake.cython.dmcommon.`**`quadrilateral_closure_ordering`**`()`

Cellwise orders mesh entities according to the given cell orientations.

Parameters

- **plex** – The DMPlex object encapsulating the mesh topology
- **vertex_numbering** – Section describing the universal vertex numbering
- **cell_numbering** – Section describing the cell numbering
- **cell_orientations** – Specifies the starting vertex for each cell, and the order of traversal (CCW or CW).

`firedrake.cython.dmcommon.`**`quadrilateral_facet_orientations`**`()`

Returns globally synchronised facet orientations (edge directions) incident to locally owned quadrilateral cells.

Parameters

- **plex** – The DMPlex object encapsulating the mesh topology
- **vertex_numbering** – Section describing the universal vertex numbering
- **cell_ranks** – MPI rank of the owner of each (visible) non-owned cell, or -1 for (locally) owned cell.

`firedrake.cython.dmcommon.`**`reordered_coords`**`()`

Return coordinates for the dm, reordered according to the global numbering permutation for the coordinate function space.

Shape is a tuple of (mesh.num_vertices(), geometric_dim).

`firedrake.cython.dmcommon.`**`set_adjacency_callback`**`()`

Set the callback for DMPlexGetAdjacency.

Parameters
dm – The DMPlex object.

This is used during DMPlexDistributeOverlap to determine where to grow the halos.

`firedrake.cython.dmcommon.`**`validate_mesh`**`()`

Perform some validation of the input mesh.

Parameters
dm – The DM object encapsulating the mesh topology.

firedrake.cython.extrusion_numbering module

Computation dof numberings for extruded meshes

On meshes with a constant number of cell layers (i.e. each column contains the same number of cells), it is possible to compute all the correct numberings by just lying to DMPlex about how many degrees of freedom there are on the base topological entities.

This ceases to be true as soon as we permit variable numbers of cells in each column, since now, although the number of degrees of freedom on a cell does not change from column to column, the number that are stacked up on each topological entity does change.

This module implements the necessary chicanery to deal with it.

Computation of topological layer extents

First, a picture.

Consider a one-dimensional mesh:

```
x---0---x---1---x---2---x
```

Extruded to form the following two-dimensional mesh:

```
                      x--------x
                      |        |
                      |        |
2                     |        |
                      |        |
    x--------x--------x--------x
    |        |        |
    |        |        |
1   |        |        |
    |        |        |
    x--------x--------x
    |        |
    |        |
0   |        |
    |        |
    x--------x
```

This is constructed by providing the number of cells in each column as well as the starting cell layer:

```
[[0, 2],
 [1, 1],
 [2, 1]]
```

We need to promote this cell layering to layering for all topological entities. Our solution to "interior" facets that only have one side is to require that they are geometrically zero sized, and then guarantee that we never iterate over them. We therefore need to keep track of two bits of information, the layer extent for allocation purposes and also the layer extent for iteration purposes.

We compute both by iterating over the cells and transferring cell layers to points in the closure of each cell. Allocation bounds use min-max on the cell bounds, iteration bounds use max-min.

To simplify some things, we require that the resulting mesh is not topologically disconnected anywhere. Offset cells must, at least, share a vertex with some other cell.

Computation of function space allocation size

With the layer extents computed, we need to compute the dof allocation. For this, we need the number of degrees of freedom *on* the base topological entity, and *above* it in each cell:

```
x-------x
|   o   |
o   o   o
o   o   o
|   o   |
o---o---o
```

This element has one degree of freedom on each base vertex and cell, two degrees of freedom "above" each vertex, and four above each cell. To compute the number of degrees of freedom on the column of topological entities we sum the number on the entity, multiplied by the number of layers with the number above, multiplied by the number of layers minus one (due to the fencepost error difference). This number of layers naturally changes from entity to entity, and so we can't compute this up front, but must do it point by point, constructing the PETSc Section as we go.

Computation of function space maps

Now we need the maps from topological entities (cells and facets) to the function space nodes they can see. The allocation offsets that the numbering section gives us are wrong, because when we have a step in the column height, the offset will be wrong if we're looking from the higher cell. Consider a vertex discretisation on the previous mesh, with a numbering:

```
                  8--------10
                  |         |
                  |         |
                  |         |
                  |         |
2--------5--------7--------9
|        |        |
|        |        |
|        |        |
|        |        |
1--------4--------6
```

(continues on next page)

(continued from previous page)

```
|        |
|        |
|        |
|        |
0--------3
```

The cell node map we get by just looking at allocation offsets is:

```
[[0, 1, 3, 4],
 [3, 4, 6, 7],
 [6, 7, 9, 10]]
```

note how the second and third cells have the wrong value for their “left” vertices. Instead, we need to shift the numbering we obtain from the allocation offset by the number of entities we’re skipping over, to result in:

```
[[0, 1, 3, 4],
 [4, 5, 6, 7],
 [7, 8, 9, 10]]
```

Now, when we iterate over cells, we ensure that we access the correct dofs. The same trick needs to be applied to facet maps too.

Computation of boundary nodes

For the top and bottom boundary nodes, we walk over the cells at, respectively, the top and bottom of the column and pull out those nodes whose entity height matches the appropriate cell height. As an example:

```
                  8--------10
                  |        |
                  |        |
                  |        |
                  |        |
2--------5--------7--------9
|        |        |
|        |        |
|        |        |
|        |        |
1--------4--------6
|        |
|        |
|        |
|        |
0--------3
```

The bottom boundary nodes are:

```
[0, 3, 4, 6, 7, 9]
```

whereas the top are:

```
[2, 5, 7, 8, 10]
```

For these strange "interior" facets, we first walk over the cells, picking up the dofs in the closure of the base (ceiling) of the cell, then we walk over facets, picking up all the dofs in the closure of facets that are exposed (there may be more than one of these in the cell column). We don't have to worry about any lower-dimensional entities, because if a co-dim 2 or greater entity is exposed in a column, then the co-dim 1 entity in its star is also exposed.

For the side boundary nodes, we can make a simplification: we know that the facet heights are always the same as the cell column heights (because there is only one cell in the support of the facet). Hence we just walk over the boundary facets of the base mesh, extract out the nodes on that facet on the bottom cell and walk up the column. This is guaranteed to pick up all the nodes in the closure of the facet column.

`firedrake.cython.extrusion_numbering.`**`entity_layers`**`()`

Compute the layers for a given entity type.

Parameters

- **`mesh`** – the extruded mesh to compute layers for.
- **`height`** – the height of the entity to consider (in the DMPlex sense). e.g. 0 -> cells, 1 -> facets, etc...
- **`label`** – optional label to select some subset of the points of the given height (may be None meaning select all points).

Returns

a numpy array of shape (num_entities, 2) providing the layer extents for iteration on the requested entities.

`firedrake.cython.extrusion_numbering.`**`facet_closure_nodes`**`()`

Extract nodes in the closure of facets with a given marker.

This works fine for interior as well as exterior facets.

Note: Don't call this function directly, but rather call *`facet_closure_nodes()`*, which will dispatch here if appropriate.

Parameters

- **`V`** – the function space
- **`sub_domain`** – a mesh marker selecting the part of the boundary

Returns

a numpy array of unique nodes on the boundary of the requested subdomain.

`firedrake.cython.extrusion_numbering.`**`layer_extents`**`()`

Compute the extents (start and stop layers) for an extruded mesh.

Parameters

- **dm** – The DMPlex.
- **cell_numbering** – The cell numbering (plex points to Firedrake points).
- **cell_extents** – The cell layers.

Returns
a numpy array of shape (npoints, 4) where npoints is the number of mesh points in the base mesh. `npoints[p, 0:2]` gives the start and stop layers for *allocation* for mesh point `p` (in plex ordering), while `npoints[p, 2:4]` gives the start and stop layers for *iteration* over mesh point `p` (in plex ordering).

Warning: The indexing of this array uses DMPlex point ordering, *not* Firedrake ordering. So you always need to iterate over plex points and translate to Firedrake numbers if necessary.

`firedrake.cython.extrusion_numbering.`**`node_classes`**`()`

Compute the node classes for a given extruded mesh.

Parameters

- **mesh** – the extruded mesh.
- **nodes_per_entity** – Number of nodes on, and on top of, each type of topological entity on the base mesh for a single cell layer. Multiplying up by the number of layers happens in this function.

Returns
A numpy array of shape (3,) giving the set entity sizes for the given nodes per entity.

`firedrake.cython.extrusion_numbering.`**`top_bottom_boundary_nodes`**`()`

Extract top or bottom boundary nodes from an extruded function space.

Parameters

- **mesh** – The extruded mesh.
- **cell_node_list** – The map from cells to nodes.
- **masks** – masks for dofs in the closure of the facets of the cell. First the vertical facets, then the horizontal facets (bottom then top).
- **offsets** – Offsets to apply walking up the column.
- **kind** – Whether we should select the bottom, or the top, nodes.

Returns
a numpy array of unique indices of nodes on the bottom or top of the mesh.

firedrake.cython.hdf5interface module

`firedrake.cython.hdf5interface.`**`get_h5py_file()`**

Attempt to convert PETSc viewer file handle to h5py File.

Parameters
`vwr` – The PETSc Viewer (must have type HDF5).

> **Warning:** For this to work, h5py and PETSc must both have been compiled against *the same* HDF5 library (otherwise the file handles are not interchangeable). This is the likeliest reason for failure when attempting the conversion.

firedrake.cython.mgimpl module

`firedrake.cython.mgimpl.`**`coarse_to_fine_cells()`**

Return a map from (renumbered) cells in a coarse mesh to those in a refined fine mesh.

Parameters

- **`mc`** – the coarse mesh to create the map from.
- **`mf`** – the fine mesh to map to.
- **`clgmaps`** – coarse lgmaps (non-overlapped and overlapped)
- **`flgmaps`** – fine lgmaps (non-overlapped and overlapped)

Returns
Two arrays, one mapping coarse to fine cells, the second fine to coarse cells.

`firedrake.cython.mgimpl.`**`coarse_to_fine_nodes()`**

`firedrake.cython.mgimpl.`**`create_lgmap()`**

Create a local to global map for all points in the given DM.

Parameters
`dm` – The DM to create the map for.

Returns a petsc4py LGMap.

`firedrake.cython.mgimpl.`**`filter_labels()`**

Remove labels from points that are not in keep. :arg dm: DM object with labels. :arg keep: subsection of the DMs chart on which to retain label values. :arg label_names: names of labels (strings) to clear. When refining, every point “underneath” the refined entity receives its label. But we typically have labels applied only to entities of a given stratum height (and rely on that elsewhere), so clear the labels from everything else.

`firedrake.cython.mgimpl.`**`fine_to_coarse_nodes()`**

`firedrake.cython.mgimpl.`**`get_entity_renumbering()`**

Given a section numbering a type of topological entity, return the renumberings from original plex numbers to new firedrake numbers (and vice versa)

Parameters

- **plex** – The DMPlex object
- **section** – The Section defining the renumbering
- **entity_type** – The type of entity (either `"cell"` or `"vertex"`)

firedrake.cython.patchimpl module

`firedrake.cython.patchimpl.`**`set_patch_jacobian`**`()`

`firedrake.cython.patchimpl.`**`set_patch_residual`**`()`

firedrake.cython.spatialindex module

`class` `firedrake.cython.spatialindex.`**`SpatialIndex`**

Bases: `object`

Python class for holding a native spatial index object.

`ctypes`

Returns a ctypes pointer to the native spatial index.

`firedrake.cython.spatialindex.`**`bounding_boxes`**`()`

Given a spatial index and a point, return the bounding boxes the point is in.

Parameters

- **sidx** – the SpatialIndex
- **x** – the point

Returns

a numpy array of candidate bounding boxes.

`firedrake.cython.spatialindex.`**`from_regions`**`()`

Builds a spatial index from a set of maximum bounding regions (MBRs).

regions_lo and regions_hi must have the same size. regions_lo[i] and regions_hi[i] contain the coordinates of the diagonally opposite lower and higher corners of the i-th MBR, respectively.

firedrake.cython.supermeshimpl module

`firedrake.cython.supermeshimpl.`**`assemble_mixed_mass_matrix`**`()`

`firedrake.cython.supermeshimpl.`**`intersection_finder`**`()`

Module contents

4.1.3 firedrake.matrix_free package

Submodules

firedrake.matrix_free.operators module

class firedrake.matrix_free.operators.**ImplicitMatrixContext**(*a*, *row_bcs=[]*, *col_bcs=[]*, *fc_params=None*, *appctx=None*)

Bases: `object`

createSubMatrix(*mat*, *row_is*, *col_is*, *target=None*)

duplicate(*mat*, *copy*)

getDiagonal(*mat*, *vec*)

getInfo(*mat*, *info=None*)

missingDiagonal(*mat*)

mult(*mat*, *X*, *Y*)

multTranspose(*mat*, *Y*, *X*)

EquationBC makes multTranspose different from mult.

Decompose M^T into bundles of columns associated with the rows of M corresponding to cell, facet, edge, and vertice equations (if exist) and add up their contributions.

```
                   Domain
    a a a a 0 a a    |
    a a a a 0 a a    |
    a a a a 0 a a    |   EBC1
M = b b b b b b b    |    |   EBC2 DBC1
    0 0 0 0 1 0 0    |    |    |    |
    c c c c 0 c c    |         |
    c c c c 0 c c    |         |
                                             To avoid copys, use␣
↪same _y, and update it
                                             from left (deepest ebc)␣
↪to right (least deep ebc or domain)
Multiplication algorithm:                       _y         update ->␣
↪    _y        update ->   _y

         a a a b 0 c c    _y0     0 0 0 0 c c c    *      0 0 0 b b 0␣
↪0    *     a a a a a a a    _y0         0
         a a a b 0 c c    _y1     0 0 0 0 c c c    *      0 0 0 b b 0␣
↪0    *     a a a a a a a    _y1         0
         a a a b 0 c c    _y2     0 0 0 0 c c c    *      0 0 0 b b 0␣
↪0    *     a a a a a a a    _y2         0
```

(continues on next page)

(continued from previous page)

```
M^T _y = a a a b 0 c c    _y3  =  0 0 0 0 c c c   *    + 0 0 0 b b 0␣
↪0   _y3  + a a a a a a a a    0      +    0
         0 0 0 0 1 0 0    _y4    0 0 0 0 c c c   0      0 0 0 b b 0␣
↪0   0      a a a a a a a a    0          _y4 (replace at the end)
         a a a b 0 c c    _y5    0 0 0 0 c c c   _y5    0 0 0 b b 0␣
↪0   *      a a a a a a a a    0          0
         a a a b 0 c c    _y6    0 0 0 0 c c c   _y6    0 0 0 b b 0␣
↪0   *      a a a a a a a a    0          0
                                         (uniform on           (uniform ␣
↪         (uniform on domain)
                                          on facet2)            on␣
↪facet1)

* = can be any number
```

on_diag = True

This class gives the Python context for a PETSc Python matrix.

Parameters

- **a** – The bilinear form defining the matrix
- **row_bcs** – An iterable of the :class.`.DirichletBC`s that are imposed on the test space. We distinguish between row and column boundary conditions in the case of submatrices off of the diagonal.
- **col_bcs** – An iterable of the :class.`.DirichletBC`s that are imposed on the trial space.
- **fcparams** – A dictionary of parameters to pass on to the form compiler.
- **appctx** – Any extra user-supplied context, available to preconditioners and the like.

view(*mat*, *viewer=None*)

Module contents

4.1.4 firedrake.mg package

Submodules

firedrake.mg.embedded module

class firedrake.mg.embedded.**TransferManager**(***, *native_transfers=None*, *use_averaging=True*)

Bases: `object`

An object for managing transfers between levels in a multigrid hierarchy (possibly via embedding in DG spaces).

Parameters

- **native_transfers** – dict mapping UFL element to "natively supported" transfer operators. This should be a three-tuple of (prolong, restrict, inject).
- **use_averaging** – Use averaging to approximate the projection out of the embedded DG space? If False, a global L2 projection will be performed.

class Cache(*element*)

Bases: `object`

A caching object for work vectors and matrices.

Parameters
element – The element to use for the caching.

DG_inv_mass(*DG*)

Inverse DG mass matrix :arg DG: the DG space :returns: A PETSc Mat.

DG_work(*V*)

A DG work Function matching V :arg V: a function space. :returns: A Function in the embedding DG space.

V_DG_mass(*V*, *DG*)

Mass matrix from between V and DG spaces. :arg V: a function space :arg DG: the DG space :returns: A PETSc Mat mapping from V -> DG

V_approx_inv_mass(*V*, *DG*)

Approximate inverse mass. Computes (cellwise) (V, V)^{-1} (V, DG). :arg V: a function space :arg DG: the DG space :returns: A PETSc Mat mapping from V -> DG.

V_dof_weights(*V*)

Dof weights for averaging projection.

Parameters
V – function space to compute weights for.

Returns
A PETSc Vec.

V_inv_mass_ksp(*V*)

A KSP inverting a mass matrix :arg V: a function space. :returns: A PETSc KSP for inverting (V, V).

cache(*element*)

inject(*uf*, *uc*)

Inject a function (primal restriction)

Parameters

- **uc** – The source (fine grid) function.
- **uf** – The target (coarse grid) function.

is_native(*element*)

op(*source*, *target*, *transfer_op*)

Primal transfer (either prolongation or injection).

Parameters

- **source** – The source function.
- **target** – The target function.
- **transfer_op** – The transfer operation for the DG space.

prolong(*uc*, *uf*)

Prolong a function.

Parameters

- **uc** – The source (coarse grid) function.
- **uf** – The target (fine grid) function.

restrict(*gf*, *gc*)

Restrict a dual function.

Parameters

- **gf** – The source (fine grid) dual function.
- **gc** – The target (coarse grid) dual function.

work_vec(*V*)

A work Vec for V :arg V: a function space. :returns: A PETSc Vec for V.

firedrake.mg.interface module

firedrake.mg.interface.**inject**(*fine*, *coarse*)

firedrake.mg.interface.**prolong**(*coarse*, *fine*)

firedrake.mg.interface.**restrict**(*fine_dual*, *coarse_dual*)

firedrake.mg.kernels module

class firedrake.mg.kernels.**MacroKernelBuilder**(*scalar_type*, *num_entities*)

Bases: `KernelBuilderBase`

Kernel builder for integration on a macro-cell.

Parameters

num_entities – the number of micro-entities to integrate over.

oriented = False

set_coefficients(*coefficients*)

set_coordinates(*domain*)

Prepare the coordinate field.

Parameters

domain – `ufl.Domain`

`firedrake.mg.kernels.`**compile_element**(*expression*, *dual_space=None*, *parameters=None*, *name='evaluate'*)

Generate code for point evaluations.

Parameters

- **expression** – A UFL expression (may contain up to one coefficient, or one argument)
- **dual_space** – if the expression has an argument, should we also distribute residual data?

Returns

Some coffee AST

`firedrake.mg.kernels.`**dg_injection_kernel**(*Vf*, *Vc*, *ncell*)

`firedrake.mg.kernels.`**inject_kernel**(*Vf*, *Vc*)

`firedrake.mg.kernels.`**prolong_kernel**(*expression*)

`firedrake.mg.kernels.`**restrict_kernel**(*Vf*, *Vc*)

`firedrake.mg.kernels.`**to_reference_coordinates**(*ufl_coordinate_element*, *parameters=None*)

firedrake.mg.mesh module

`firedrake.mg.mesh.`**ExtrudedMeshHierarchy**(*base_hierarchy*, *height*, *base_layer=-1*, *refinement_ratio=2*, *layers=None*, *kernel=None*, *extrusion_type='uniform'*, *gdim=None*, *mesh_builder=<cyfunction ExtrudedMesh>*)

Build a hierarchy of extruded meshes by extruding a hierarchy of meshes.

Parameters

- **base_hierarchy** – the unextruded base mesh hierarchy to extrude.
- **height** – the height of the domain to extrude to. This is in contrast to the extrusion routines, which take in layer_height, the height of an individual layer. This is because when refining in the extruded dimension, the height of an individual layer will vary.
- **base_layer** – the number of layers to use the extrusion of the coarsest grid.
- **refinement_ratio** – the ratio by which base_layer should be increased on every refinement. refinement_ratio = 2 means standard uniform refinement. refinement_ratio = 1 means to not refine in the extruded dimension, i.e. the multigrid hierarchy will use semicoarsening.

- **layers** – as an alternative to specifying base_layer and refinement_ratio, one may specify directly the number of layers to be used by each level in the extruded hierarchy. This option cannot be combined with base_layer and refinement_ratio. Note that the ratio of successive entries in this iterable must be an integer for the multigrid transfer operators to work.
- **mesh_builder** – function used to turn a `Mesh` into an extruded mesh. Used by pyadjoint.

See *ExtrudedMesh()* for the meaning of the remaining parameters.

class `firedrake.mg.mesh.`**HierarchyBase**(*meshes*, *coarse_to_fine_cells*, *fine_to_coarse_cells*, *refinements_per_level=1*, *nested=False*)

Bases: `object`

Create an encapsulation of an hierarchy of meshes.

Parameters

- **meshes** – list of meshes (coarse to fine)
- **coarse_to_fine_cells** – list of numpy arrays for each level pair, mapping each coarse cell into fine cells it intersects.
- **fine_to_coarse_cells** – list of numpy arrays for each level pair, mapping each fine cell into coarse cells it intersects.
- **refinements_per_level** – number of mesh refinements each multigrid level should "see".
- **nested** – Is this mesh hierarchy nested?

Note: Most of the time, you do not need to create this object yourself, instead using *MeshHierarchy()*, *ExtrudedMeshHierarchy()*, or *NonNestedHierarchy()*.

comm

`firedrake.mg.mesh.`**MeshHierarchy**(*mesh*, *refinement_levels*, *refinements_per_level=1*, *reorder=None*, *distribution_parameters=None*, *callbacks=None*, *mesh_builder=<cyfunction Mesh>*)

Build a hierarchy of meshes by uniformly refining a coarse mesh.

Parameters

- **mesh** – the coarse *Mesh()* to refine
- **refinement_levels** – the number of levels of refinement
- **refinements_per_level** – the number of refinements for each level in the hierarchy.
- **distribution_parameters** – options controlling mesh distribution, see *Mesh()* for details. If `None`, use the same distribution parameters as were used to distribute the coarse mesh, otherwise, these options override the default.

- **reorder** – optional flag indicating whether to reorder the refined meshes.
- **callbacks** – A 2-tuple of callbacks to call before and after refinement of the DM. The before callback receives the DM to be refined (and the current level), the after callback receives the refined DM (and the current level).
- **mesh_builder** – Function to turn a DM into a `Mesh`. Used by pyadjoint.

`firedrake.mg.mesh.`**`NonNestedHierarchy`**`(`**meshes*`)`

`firedrake.mg.mesh.`**`SemiCoarsenedExtrudedHierarchy`**`(`*base_mesh*, *height*, *nref=1*, *base_layer=-1*, *refinement_ratio=2*, *layers=None*, *kernel=None*, *extrusion_type='uniform'*, *gdim=None*, *mesh_builder=<cyfunction ExtrudedMesh>*`)`

Build a hierarchy of extruded meshes with refinement only in the extruded dimension.

Parameters

- **base_mesh** – the unextruded base mesh to extrude.
- **nref** – Number of refinements.
- **height** – the height of the domain to extrude to. This is in contrast to the extrusion routines, which take in layer_height, the height of an individual layer. This is because when refining in the extruded dimension, the height of an individual layer will vary.
- **base_layer** – the number of layers to use the extrusion of the coarsest grid.
- **refinement_ratio** – the ratio by which base_layer should be increased on every refinement. refinement_ratio = 2 means standard uniform refinement. refinement_ratio = 1 means to not refine in the extruded dimension, i.e. the multigrid hierarchy will use semicoarsening.
- **layers** – as an alternative to specifying base_layer and refinement_ratio, one may specify directly the number of layers to be used by each level in the extruded hierarchy. This option cannot be combined with base_layer and refinement_ratio. Note that the ratio of successive entries in this iterable must be an integer for the multigrid transfer operators to work.
- **mesh_builder** – function used to turn a `Mesh` into an extruded mesh. Used by pyadjoint.

See *`ExtrudedMesh()`* for the meaning of the remaining parameters.

See also *`ExtrudedMeshHierarchy()`* if you want to extruded a hierarchy of unstructured meshes.

firedrake.mg.opencascade_mh module

firedrake.mg.opencascade_mh.**OpenCascadeMeshHierarchy**(*stepfile*, *element_size*, *levels*, *comm=<mpi4py.MPI.Intracomm object>*, *distribution_parameters=None*, *callbacks=None*, *order=1*, *mh_constructor=<function MeshHierarchy>*, *cache=True*, *verbose=True*, *gmsh='gmsh'*, *project_refinements_to_cad=True*, *reorder=None*)

firedrake.mg.ufl_utils module

firedrake.mg.ufl_utils.**coarsen**(*expr*, *self*, *coefficient_mapping=None*)

firedrake.mg.ufl_utils.**coarsen**(*mesh: Mesh*, *self*, *coefficient_mapping=None*)

firedrake.mg.ufl_utils.**coarsen**(*expr: Expr*, *self*, *coefficient_mapping=None*)

firedrake.mg.ufl_utils.**coarsen**(*form: Form*, *self*, *coefficient_mapping=None*)

firedrake.mg.ufl_utils.**coarsen**(*bc:* DirichletBC, *self*, *coefficient_mapping=None*)

firedrake.mg.ufl_utils.**coarsen**(*V:* WithGeometry, *self*, *coefficient_mapping=None*)

firedrake.mg.ufl_utils.**coarsen**(*V:* FunctionSpace, *self*, *coefficient_mapping=None*)

firedrake.mg.ufl_utils.**coarsen**(*expr:* Function, *self*, *coefficient_mapping=None*)

firedrake.mg.ufl_utils.**coarsen**(*expr:* Constant, *self*, *coefficient_mapping=None*)

firedrake.mg.ufl_utils.**coarsen**(*problem:* NonlinearVariationalProblem, *self*, *coefficient_mapping=None*)

firedrake.mg.ufl_utils.**coarsen**(*basis:* VectorSpaceBasis, *self*, *coefficient_mapping=None*)

firedrake.mg.ufl_utils.**coarsen**(*mspbasis:* MixedVectorSpaceBasis, *self*, *coefficient_mapping=None*)

firedrake.mg.ufl_utils.**coarsen**(*context: _SNESContext*, *self*, *coefficient_mapping=None*)

firedrake.mg.utils module

firedrake.mg.utils.**coarse_cell_to_fine_node_map**(*Vc*, *Vf*)

firedrake.mg.utils.**coarse_node_to_fine_node_map**(*Vc*, *Vf*)

firedrake.mg.utils.**fine_node_to_coarse_node_map**(*Vf*, *Vc*)

firedrake.mg.utils.**get_level**(*obj*)

Try and obtain hierarchy and level info from an object.

If no level info is available, return `None, None`.

`firedrake.mg.utils.`**`has_level`**`(`*obj*`)`

Does the provided object have level info?

`firedrake.mg.utils.`**`physical_node_locations`**`(`*V*`)`

`firedrake.mg.utils.`**`set_level`**`(`*obj*, *hierarchy*, *level*`)`

Attach hierarchy and level info to an object.

Module contents

4.1.5 firedrake.preconditioners package

Submodules

firedrake.preconditioners.asm module

class `firedrake.preconditioners.asm.`**`ASMExtrudedStarPC`**

Bases: *`ASMStarPC`*

Patch-based PC using Star of mesh entities implmented as an *`ASMPatchPC`*.

ASMExtrudedStarPC is an additive Schwarz preconditioner where each patch consists of all DoFs on the topological star of the mesh entity specified by *pc_star_construct_dim*.

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize()`
- `update()`
- `apply()`
- `applyTranspose()`

`get_patches``(`*V*`)`

Get the patches used for PETSc PSASM

Parameters

V – the *`FunctionSpace`*.

Returns

a list of index sets defining the ASM patches in local numbering (before lgmap.apply has been called).

class `firedrake.preconditioners.asm.`**`ASMLinesmoothPC`**

Bases: *`ASMPatchPC`*

Linesmoother PC for extruded meshes implemented as an *`ASMPatchPC`*.

ASMLinesmoothPC is an additive Schwarz preconditioner where each patch consists of all dofs associated with a vertical column (and hence extruded meshes are necessary). Three types of columns are possible: columns of horizontal faces (each column built over a face of the base mesh), columns of vertical faces (each column built over an edge of the base mesh), and columns of vertical edges (each column built over a vertex of the base mesh).

To select the column type or types for the patches, use ‘pc_linesmooth_codims’ to set integers giving the codimension of the base mesh entities for the columns. For example, ‘pc_linesmooth_codims 0,1’ creates patches for each cell and each facet of the base mesh.

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize()`
- `update()`
- `apply()`
- `applyTranspose()`

`get_patches`(*V*)

Get the patches used for PETSc PSASM

Parameters

V – the *FunctionSpace*.

Returns

a list of index sets defining the ASM patches in local numbering (before lgmap.apply has been called).

`class firedrake.preconditioners.asm.`**`ASMPatchPC`**

Bases: *PCBase*

PC for PETSc PCASM

should implement: - *get_patches()*

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- *initialize()*
- *update()*
- *apply()*
- *applyTranspose()*

`apply`(*pc*, *x*, *y*)

Apply the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

`applyTranspose`(*pc*, *x*, *y*)

Apply the transpose of the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

`destroy`(*pc*)

`abstract get_patches`(*V*)

Get the patches used for PETSc PSASM

Parameters

V – the *FunctionSpace*.

Returns
a list of index sets defining the ASM patches in local numbering (before lgmap.apply has been called).

`initialize`(*pc*)
Initialize any state in this preconditioner.

`update`(*pc*)
Update any state in this preconditioner.

`view`(*pc*, *viewer=None*)

class `firedrake.preconditioners.asm.`**`ASMStarPC`**
Bases: *ASMPatchPC*

Patch-based PC using Star of mesh entities implmented as an *ASMPatchPC*.

ASMStarPC is an additive Schwarz preconditioner where each patch consists of all DoFs on the topological star of the mesh entity specified by *pc_star_construct_dim*.

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize()`
- `update()`
- `apply()`
- `applyTranspose()`

`get_patches`(*V*)
Get the patches used for PETSc PSASM

Parameters
V – the *FunctionSpace*.

Returns
a list of index sets defining the ASM patches in local numbering (before lgmap.apply has been called).

class `firedrake.preconditioners.asm.`**`ASMVankaPC`**
Bases: *ASMPatchPC*

Patch-based PC using closure of star of mesh entities implmented as an *ASMPatchPC*.

ASMVankaPC is an additive Schwarz preconditioner where each patch consists of all DoFs on the closure of the star of the mesh entity specified by *pc_vanka_construct_dim* (or codim).

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize()`
- `update()`
- `apply()`
- `applyTranspose()`

get_patches(*V*)

Get the patches used for PETSc PSASM

Parameters
V – the *FunctionSpace*.

Returns
a list of index sets defining the ASM patches in local numbering (before lgmap.apply has been called).

firedrake.preconditioners.assembled module

class firedrake.preconditioners.assembled.**AssembledPC**

Bases: *PCBase*

A matrix-free PC that assembles the operator.

Internally this makes a PETSc PC object that can be controlled by options using the extra options prefix `assembled_`.

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- *initialize()*
- *update()*
- *apply()*
- *applyTranspose()*

apply(*pc*, *x*, *y*)

Apply the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

applyTranspose(*pc*, *x*, *y*)

Apply the transpose of the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

form(*pc*, *test*, *trial*)

initialize(*pc*)

Initialize any state in this preconditioner.

update(*pc*)

Update any state in this preconditioner.

view(*pc*, *viewer=None*)

class firedrake.preconditioners.assembled.**AuxiliaryOperatorPC**

Bases: *AssembledPC*

A preconditioner that builds a PC on a specified form. Mainly used for describing approximations to Schur complements.

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize()`
- `update()`
- `apply()`
- `applyTranspose()`

abstract form(*pc*, *test*, *trial*)

Parameters

- **pc** – a *PETSc.PC* object. Use *self.get_appctx(pc)* to get the user-supplied application-context, if desired.
- **test** – a *TestFunction* on this *FunctionSpace*.
- **trial** – a *TrialFunction* on this *FunctionSpace*.

:returns *(a, bcs)*, where *a* is a bilinear *Form* and *bcs* is a list of *DirichletBC* boundary conditions (possibly *None*).

firedrake.preconditioners.base module

class firedrake.preconditioners.base.**PCBase**

Bases: *PCSNESBase*

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize()`
- `update()`
- *`apply()`*
- *`applyTranspose()`*

abstract apply(*pc*, *X*, *Y*)

Apply the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

abstract applyTranspose(*pc*, *X*, *Y*)

Apply the transpose of the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

needs_python_amat = False

Set this to True if the A matrix needs to be Python (matfree).

needs_python_pmat = False

Set this to False if the P matrix needs to be Python (matfree).

If the preconditioner also works with assembled matrices, then use False here.

setUp(*pc*)

Setup method called by PETSc.

Subclasses should probably not override this and instead implement `update()` and `initialize()`.

class firedrake.preconditioners.base.**PCSNESBase**

Bases: `object`

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- *`initialize()`*
- *`update()`*
- `apply()`
- `applyTranspose()`

destroy(*pc*)

static **get_appctx**(*pc*)

abstract **initialize**(*pc*)

Initialize any state in this preconditioner.

static **new_snes_ctx**(*pc*, *op*, *bcs*, *mat_type*, *fcp=None*, *options_prefix=None*)

Create a new SNES contex for nested preconditioning

setUp(*pc*)

Setup method called by PETSc.

Subclasses should probably not override this and instead implement *`update()`* and *`initialize()`*.

abstract **update**(*pc*)

Update any state in this preconditioner.

view(*pc*, *viewer=None*)

class firedrake.preconditioners.base.**SNESBase**

Bases: *`PCSNESBase`*

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize()`
- `update()`
- `apply()`
- `applyTranspose()`

firedrake.preconditioners.fdm module

class firedrake.preconditioners.fdm.**FDMPC**

Bases: *PCBase*

A preconditioner for tensor-product elements that changes the shape functions so that the H^1 Riesz map is diagonalized in the interior of a Cartesian cell, and assembles a global sparse matrix on which other preconditioners, such as *ASMStarPC*, can be applied.

Here we assume that the volume integrals in the Jacobian can be expressed as:

inner(grad(v), alpha(grad(u)))*dx + inner(v, beta(u))*dx

where alpha and beta are linear functions (tensor contractions). The sparse matrix is obtained by approximating alpha and beta by cell-wise constants and discarding the coefficients in alpha that couple together mixed derivatives and mixed components.

For spaces that are not H^1-conforming, this preconditioner will use the symmetric interior-penalty DG method. The penalty coefficient can be provided in the application context, keyed on `"eta"`.

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- *initialize()*
- *update()*
- *apply()*
- *applyTranspose()*

apply(*pc*, *x*, *y*)

Apply the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

applyTranspose(*pc*, *x*, *y*)

Apply the transpose of the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

assemble_coef(*J*, *quad_deg*, *discard_mixed=True*, *cell_average=True*)

Return the coefficients of the Jacobian form arguments and their gradient with respect to the reference coordinates.

Parameters

- **J** – the Jacobian bilinear form
- **quad_deg** – the quadrature degree used for the coefficients
- **discard_mixed** – discard entries in second order coefficient with mixed derivatives and mixed components
- **cell_average** – to return the coefficients as DG_0 Functions

Returns

a 2-tuple of coefficients: a dictionary mapping strings to `firedrake.Functions` with the coefficients of the form, assembly_callables: a list of assembly callables for each coefficient of the form

assemble_fdm_op(*V*, *J*, *bcs*, *appctx*)

Assemble the sparse preconditioner with cell-wise constant coefficients.

Parameters

- **V** – the `firedrake.FunctionSpace` of the form arguments
- **J** – the Jacobian bilinear form
- **bcs** – an iterable of boundary conditions on V
- **appctx** – the application context

Returns

2-tuple with the preconditioner `PETSc.Mat` and its assembly callable

assemble_kron(*A*, *V*, *bcs*, *eta*, *coefficients*, *Afdm*, *Dfdm*, *bcflags*)

Assemble the stiffness matrix in the FDM basis using Kronecker products of interval matrices

Parameters

- **A** – the `PETSc.Mat` to assemble
- **V** – the `firedrake.FunctionSpace` of the form arguments
- **bcs** – an iterable of `firedrake.DirichletBCs`
- **eta** – a `float` penalty parameter for the symmetric interior penalty method
- **coefficients** – a `dict` mapping strings to `firedrake.Functions` with the form coefficients
- **Afdm** – the list with sparse interval matrices
- **Dfdm** – the list with normal derivatives matrices
- **bcflags** – the `numpy.ndarray` with BC facet flags returned by *get_weak_bc_flags*

initialize(*pc*)

Initialize any state in this preconditioner.

update(*pc*)

Update any state in this preconditioner.

view(*pc*, *viewer=None*)

firedrake.preconditioners.gtmg module

class firedrake.preconditioners.gtmg.**GTMGPC**

Bases: *PCBase*

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- *initialize()*
- *update()*

- *apply()*
- *applyTranspose()*

apply(*pc*, *X*, *Y*)

Apply the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

applyTranspose(*pc*, *X*, *Y*)

Apply the transpose of the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

initialize(*pc*)

Initialize any state in this preconditioner.

needs_python_pmat = False

Set this to False if the P matrix needs to be Python (matfree).

If the preconditioner also works with assembled matrices, then use False here.

update(*pc*)

Update any state in this preconditioner.

view(*pc*, *viewer=None*)

firedrake.preconditioners.hypre_ads module

class firedrake.preconditioners.hypre_ads.**HypreADS**

Bases: *PCBase*

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- *initialize()*
- *update()*
- *apply()*
- *applyTranspose()*

apply(*pc*, *x*, *y*)

Apply the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

applyTranspose(*pc*, *x*, *y*)

Apply the transpose of the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

initialize(*obj*)

Initialize any state in this preconditioner.

update(*pc*)

Update any state in this preconditioner.

view(*pc*, *viewer=None*)

firedrake.preconditioners.hypre_ams module

class firedrake.preconditioners.hypre_ams.**HypreAMS**

Bases: *PCBase*

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- *initialize()*
- *update()*
- *apply()*
- *applyTranspose()*

apply(*pc*, *x*, *y*)

Apply the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

applyTranspose(*pc*, *x*, *y*)

Apply the transpose of the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

initialize(*obj*)

Initialize any state in this preconditioner.

update(*pc*)

Update any state in this preconditioner.

view(*pc*, *viewer=None*)

firedrake.preconditioners.low_order module

class firedrake.preconditioners.low_order.**P1PC**

Bases: *PMGPC*

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- initialize()
- update()
- apply()
- applyTranspose()

coarsen_element(*ele*)

Coarsen a given element to form the next problem down in the p-hierarchy.

If the supplied element should form the coarsest level of the p-hierarchy, raise *ValueError*. Otherwise, return a new `ufl.FiniteElement`.

By default, this does power-of-2 coarsening in polynomial degree until we reach the coarse degree specified through PETSc options (1 by default).

Parameters
ele – a `ufl.FiniteElement` to coarsen.

class firedrake.preconditioners.low_order.**P1SNES**

Bases: *PMGSNES*

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize()`
- `update()`
- `apply()`
- `applyTranspose()`

coarsen_element(*ele*)

Coarsen a given element to form the next problem down in the p-hierarchy.

If the supplied element should form the coarsest level of the p-hierarchy, raise *ValueError*. Otherwise, return a new `ufl.FiniteElement`.

By default, this does power-of-2 coarsening in polynomial degree until we reach the coarse degree specified through PETSc options (1 by default).

Parameters
ele – a `ufl.FiniteElement` to coarsen.

firedrake.preconditioners.massinv module

class firedrake.preconditioners.massinv.**MassInvPC**

Bases: *PCBase*

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- *initialize()*
- *update()*
- *apply()*
- *applyTranspose()*

apply(*pc*, *X*, *Y*)

Apply the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

applyTranspose(*pc*, *X*, *Y*)

Apply the transpose of the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

destroy(*pc*)

initialize(*pc*)

Initialize any state in this preconditioner.

needs_python_pmat = True

A matrix free operator that inverts the mass matrix in the provided space.

Internally this creates a PETSc KSP object that can be controlled by options using the extra options prefix `Mp_`.

For Stokes problems, to be spectrally equivalent to the Schur complement, the mass matrix should be weighted by the viscosity. This can be provided (defaulting to constant viscosity) by providing a field defining the viscosity in the application context, keyed on `"mu"`.

update(*pc*)

Update any state in this preconditioner.

view(*pc*, *viewer=None*)

firedrake.preconditioners.patch module

class firedrake.preconditioners.patch.**PatchPC**

Bases: *PCBase*, `PatchBase`

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize()`
- `update()`
- *apply()*
- *applyTranspose()*

apply(*pc*, *x*, *y*)

Apply the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

applyTranspose(*pc*, *x*, *y*)

Apply the transpose of the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

configure_patch(*patch*, *pc*)

class `firedrake.preconditioners.patch.`**`PatchSNES`**

Bases: *`SNESBase`*, `PatchBase`

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize()`
- `update()`
- `apply()`
- `applyTranspose()`

`configure_patch`(*patch*, *snes*)

`step`(*snes*, *x*, *f*, *y*)

class `firedrake.preconditioners.patch.`**`PlaneSmoother`**

Bases: `object`

static **`coords`**(*dm*, *p*, *coordinates*)

`sort_entities`(*dm*, *axis*, *dir*, *ndiv=None*, *divisions=None*)

firedrake.preconditioners.pcd module

class `firedrake.preconditioners.pcd.`**`PCDPC`**

Bases: *`PCBase`*

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- *`initialize()`*
- *`update()`*
- *`apply()`*
- *`applyTranspose()`*

`apply`(*pc*, *x*, *y*)

Apply the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

`applyTranspose`(*pc*, *x*, *y*)

Apply the transpose of the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

`destroy`(*pc*)

`initialize`(*pc*)

Initialize any state in this preconditioner.

needs_python_pmat = True

A Pressure-Convection-Diffusion preconditioner for Navier-Stokes.

This preconditioner approximates the inverse of the pressure schur complement for the Navier-Stokes equations by.

$$S^{-1} \sim K^{-1} F_p M^{-1}$$

Where $K = \nabla^2$, $F_p = (1/\mathrm{Re})\nabla^2 + u \cdot \nabla$ and $M = \mathbb{I}$.

The inverse of K is approximated by a KSP which can be controlled using the options prefix `pcd_Kp_`.

The inverse of M is similarly approximated by a KSP which can be controlled using the options prefix `pcd_Mp_`.

F_p requires both the Reynolds number and the current velocity. You must provide these with options using the glbaol option `Re` for the Reynolds number and the prefixed option `pcd_velocity_space` which should be the index into the full space that gives velocity field.

Note: Currently, the boundary conditions applied to the PCD operator are correct for characteristic velocity boundary conditions, but sub-optimal for in and outflow boundaries.

update(*pc*)

Update any state in this preconditioner.

view(*pc*, *viewer=None*)

firedrake.preconditioners.pmg module

class firedrake.preconditioners.pmg.**PMGPC**

Bases: *PCBase*, `PMGBase`

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize()`
- `update()`
- *apply()*
- *applyTranspose()*

apply(*pc*, *x*, *y*)

Apply the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

applyTranspose(*pc*, *x*, *y*)

Apply the transpose of the preconditioner to X, putting the result in Y.

Both X and Y are PETSc Vecs, Y is not guaranteed to be zero on entry.

coarsen_bc_value(*bc*, *cV*)

configure_pmg(*pc*, *pdm*)

class firedrake.preconditioners.pmg.**PMGSNES**

Bases: *SNESBase*, PMGBase

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- `initialize()`
- `update()`
- `apply()`
- `applyTranspose()`

coarsen_bc_value(*bc*, *cV*)

configure_pmg(*snes*, *pdm*)

step(*snes*, *x*, *f*, *y*)

Module contents

4.1.6 firedrake.slate package

Subpackages

firedrake.slate.slac package

Submodules

firedrake.slate.slac.compiler module

This is Slate's Linear Algebra Compiler. This module is responsible for generating C++ kernel functions representing symbolic linear algebra expressions written in Slate.

This linear algebra compiler uses both Firedrake's form compiler, the Two-Stage Form Compiler (TSFC) and COFFEE's kernel abstract syntax tree (AST) optimizer. TSFC provides this compiler with appropriate kernel functions (in C) for evaluating integral expressions (finite element variational forms written in UFL). COFFEE's AST base helps with the construction of code blocks throughout the kernel returned by: *compile_expression*.

The Eigen C++ library (http://eigen.tuxfamily.org/) is required, as all low-level numerical linear algebra operations are performed using this templated function library.

firedrake.slate.slac.compiler.**compile_expression**(*slate_expr*, *compiler_parameters=None*, *coffee=False*)

Takes a Slate expression *slate_expr* and returns the appropriate `firedrake.op2.Kernel` object representing the Slate expression.

Parameters

- **slate_expr** – a :class:'TensorBase' expression.
- **tsfc_parameters** – an optional *dict* of form compiler parameters to be passed to TSFC during the compilation of ufl forms.

Returns: A *tuple* containing a *SplitKernel(idx, kinfo)*

firedrake.slate.slac.kernel_builder module

class `firedrake.slate.slac.kernel_builder.`**`CellFacetKernelArg`**(*ast_arg*)

Bases: `KernelArg`

class `firedrake.slate.slac.kernel_builder.`**`CoefficientInfo`**(*space_index*, *offset_index*, *shape*, *vector*, *local_temp*)

Bases: `tuple`

Context information for creating coefficient temporaries.

Parameters

- **space_index** – An integer denoting the function space index.
- **offset_index** – An integer denoting the starting position in the vector temporary for assignment.
- **shape** – A singleton with an integer describing the shape of the coefficient temporary.
- **vector** – The `slate.AssembledVector` containing the relevant data to be placed into the temporary.
- **local_temp** – The local temporary for the coefficient vector.

Create new instance of CoefficientInfo(space_index, offset_index, shape, vector, local_temp)

local_temp

Alias for field number 4

offset_index

Alias for field number 1

shape

Alias for field number 2

space_index

Alias for field number 0

vector

Alias for field number 3

class `firedrake.slate.slac.kernel_builder.`**`IndexCreator`**

Bases: `object`

property domains

ISL domains for the currently known indices.

`class firedrake.slate.slac.kernel_builder.`**`LayerCountKernelArg`**`(`*ast_arg*`)`

Bases: `KernelArg`

`class firedrake.slate.slac.kernel_builder.`**`LocalKernelBuilder`**`(`*expression*, *tsfc_parameters=None*`)`

Bases: `object`

The primary helper class for constructing cell-local linear algebra kernels from Slate expressions.

This class provides access to all temporaries and subkernels associated with a Slate expression. If the Slate expression contains nodes that require operations on already assembled data (such as the action of a slate tensor on a *ufl.Coefficient*), this class provides access to the expression which needs special handling.

Instructions for assembling the full kernel AST of a Slate expression is provided by the method *construct_ast*.

Constructor for the LocalKernelBuilder class.

Parameters

- **expression** – a `TensorBase` object.
- **tsfc_parameters** – an optional *dict* of parameters to provide to TSFC when constructing subkernels associated with the expression.

`cell_facet_sym = <coffee.base.Symbol object>`

`cell_orientations_sym = <coffee.base.Symbol object>`

`cell_size_sym = <coffee.base.Symbol object>`

`coefficient``(`*coefficient*`)`

Extracts the kernel arguments corresponding to a particular coefficient. This handles both the case when the coefficient is defined on a mixed or non-mixed function space.

`coefficient_map`

Generates a mapping from a coefficient to its kernel argument symbol. If the coefficient is mixed, all of its split components will be returned.

`context_kernels`

Gathers all *`ContextKernel`*s containing all TSFC kernels, and integral type information.

`coord_sym = <coffee.base.Symbol object>`

`expression_flops`

`property` **`integral_type`**

Returns the integral type associated with a Slate kernel.

`it_sym = <coffee.base.Symbol object>`

`mesh_layer_count_sym = <coffee.base.Symbol object>`

`mesh_layer_sym = <coffee.base.Symbol object>`

needs_cell_facets

Searches for any embedded forms (by inspecting the ContextKernels) which require looping over cell facets. If any are found, this function returns *True* and *False* otherwise.

needs_mesh_layers

Searches for any embedded forms (by inspecting the ContextKernels) which require mesh level information (extrusion measures). If any are found, this function returns *True* and *False* otherwise.

supported_integral_types = ['cell', 'interior_facet', 'exterior_facet', 'interior_facet_horiz_top', 'interior_facet_horiz_bottom', 'interior_facet_vert', 'exterior_facet_top', 'exterior_facet_bottom', 'exterior_facet_vert']

supported_subdomain_types = ['subdomains_exterior_facet', 'subdomains_interior_facet']

terminal_flops

class firedrake.slate.slac.kernel_builder.**LocalLoopyKernelBuilder**(*expression*, *tsfc_parameters=None*)

Bases: `object`

Constructor for the LocalGEMKernelBuilder class.

Parameters

- **expression** – a `TensorBase` object.
- **tsfc_parameters** – an optional *dict* of parameters to provide to TSFC when constructing subkernels associated with the expression.

cell_facets_arg_name = 'cell_facets'

cell_orientations_arg_name = 'cell_orientations'

cell_sizes_arg_name = 'cell_sizes'

collect_coefficients()

Saves all coefficients of self.expression where non-mixed coefficients are dicts of form {coeff: (name, extent)} and mixed coefficients are double dicts of form {mixed_coeff: {coeff_per_space: (name, extent)}}.

collect_tsfc_kernel_data(*mesh*, *tsfc_coefficients*, *wrapper_coefficients*, *kinfo*)

Collect the kernel data aka the parameters fed into the subkernel, that are coordinates, orientations, cell sizes and cofficients.

coordinates_arg_name = 'coords'

extent(*coefficient*)

Calculation of the range of a coefficient.

facet_integral_predicates(*mesh*, *integral_type*, *kinfo*)

generate_lhs(*tensor*, *temp*)

Generation of an lhs for the loopy kernel, which contains the TSFC assembly of the tensor.

generate_tsfc_calls(*terminal*, *loopy_tensor*)

A setup method to initialize all the local assembly kernels generated by TSFC. This function also collects any information regarding orientations and extra include directories.

generate_wrapper_kernel_args(*tensor2temp*)

initialise_terminals(*var2terminal*, *coefficients*)

Initilisation of the variables in which coefficients and the Tensors coming from TSFC are saved.

Parameters

`var2terminal` – dictionary that maps Slate Tensors to gem Variables

is_integral_type(*integral_type*, *type*)

layer_arg_name = 'layer'

layer_count_name = 'layer_count'

layer_integral_predicates(*tensor*, *integral_type*)

local_facet_array_arg_name = 'facet_array'

loopify_tsfc_kernel_data(*kernel_data*)

This method generates loopy arguments from the kernel data, which are then fed to the TSFC loopy kernel. The arguments are arrays and have to be fed element by element to loopy aka they have to be subarrayrefed.

shape(*tensor*)

A helper method to retrieve tensor shape information. In particular needed for the right shape of scalar tensors.

slate_call(*prg*, *temporaries*)

supported_integral_types = ['cell', 'interior_facet', 'exterior_facet', 'interior_facet_horiz_top', 'interior_facet_horiz_bottom', 'interior_facet_vert', 'exterior_facet_top', 'exterior_facet_bottom', 'exterior_facet_vert']

supported_subdomain_types = ['subdomains_exterior_facet', 'subdomains_interior_facet']

tsfc_cxt_kernels(*terminal*)

Gathers all *ContextKernel*s containing all TSFC kernels, and integral type information.

class firedrake.slate.slac.kernel_builder.**SlateWrapperBag**(*coeffs*)

Bases: `object`

firedrake.slate.slac.optimise module

class `firedrake.slate.slac.optimise.`**`ActionBag`**(*coeff*, *pick_op*)

Bases: `tuple`

Create new instance of ActionBag(coeff, pick_op)

`coeff`

Alias for field number 0

`pick_op`

Alias for field number 1

`firedrake.slate.slac.optimise.`**`drop_double_transpose`**(*expr*)

Remove double transposes from optimised Slate expression.

`firedrake.slate.slac.optimise.`**`flip`**(*pick_op*)

Flip an index. Using this function essentially reverses the order of multiplication.

`firedrake.slate.slac.optimise.`**`optimise`**(*expression*, *parameters*)

Optimises a Slate expression, by pushing blocks and multiplications inside the expression and by removing double transposes.

Parameters

- **`expression`** – A (potentially unoptimised) Slate expression.
- **`parameters`** – A dict of compiler parameters.

Returns: An optimised Slate expression

`firedrake.slate.slac.optimise.`**`push_block`**(*expression*)

Executes a Slate compiler optimisation pass. The optimisation is achieved by pushing blocks from the outside to the inside of an expression. Without the optimisation the local TSFC kernels are assembled first and then the result of the assembly kernel gets indexed in the Slate kernel (and further linear algebra operations maybe done on it). The optimisation pass essentially changes the order of assembly and indexing.

Parameters

`expression` – A (potentially unoptimised) Slate expression.

Returns: An optimised Slate expression, where Blocks are terminal whereever possible.

`firedrake.slate.slac.optimise.`**`push_diag`**(*expression*)

Executes a Slate compiler optimisation pass. The optimisation is achieved by pushing DiagonalTensor from the outside to the inside of an expression.

Parameters

`expression` – A (potentially unoptimised) Slate expression.

Returns: An optimised Slate expression, where DiagonalTensors are sitting on terminal tensors whereever possible.

`firedrake.slate.slac.optimise.`**`push_mul`**(*tensor*, *options*)

Executes a Slate compiler optimisation pass. The optimisation is achieved by pushing coefficients from the outside to the inside of an expression. The optimisation pass essentially changes the order of operations in the expressions so that only matrix-vector products are executed.

Parameters

- **tensor** – A (potentially unoptimised) Slate expression.
- **options** – Optimisation pass options, e.g. if the multiplication should be replaced by an action.

Returns: An optimised Slate expression,
where only matrix-vector products are executed whereever possible.

firedrake.slate.slac.tsfc_driver module

class firedrake.slate.slac.tsfc_driver.**ContextKernel**(*tensor*, *coefficients*, *original_integral_type*, *tsfc_kernels*)

Bases: tuple

A bundled object containing TSFC subkernels corresponding to a particular integral type.

Parameters

- **tensor** – The terminal Slate tensor corresponding to the list of TSFC assembly kernels.
- **coefficients** – The local coefficients of the tensor contained in the integrands (arguments for TSFC subkernels).
- **original_integral_type** – The unmodified measure type of the form integrals.
- **tsfc_kernels** – A list of local tensor assembly kernels provided by TSFC.

Create new instance of ContextKernel(tensor, coefficients, original_integral_type, tsfc_kernels)

coefficients
Alias for field number 1

original_integral_type
Alias for field number 2

tensor
Alias for field number 0

tsfc_kernels
Alias for field number 3

firedrake.slate.slac.tsfc_driver.**compile_terminal_form**(*tensor*, *prefix*, ***, *tsfc_parameters=None*, *coffee=True*)

Compiles the TSFC form associated with a Slate Tensor object. This function will return a *ContextKernel* which stores information about the original tensor, integral types and the corresponding TSFC kernels.

Parameters

- **tensor** – A Slate *Tensor*.
- **prefix** – An optional *string* indicating the prefix for the subkernel.
- **tsfc_parameters** – An optional *dict* of parameters to provide TSFC.

Returns: A *ContextKernel* containing all relevant information.

`firedrake.slate.slac.tsfc_driver.`**`transform_integrals`**`(`*`integrals`*`)`

Generates a mapping of the form:

`{original_integral_type:  transformed_integrals}`

where the original_integral_type is the pre-transformed integral type. The transformed_integrals are an iterable of *ufl.Integral`s with the appropriately modified type. For example, an `interior_facet* integral will become an *exterior_facet* integral.

firedrake.slate.slac.utils module

class `firedrake.slate.slac.utils.`**`RemoveRestrictions`**

Bases: `MultiFunction`

UFL MultiFunction for removing any restrictions on the integrals of forms.

`expr``(`*`o`*`, `*`*ops`*`)`

Reuse object if operands are the same objects.

Use in your own subclass by setting e.g.

```
expr = MultiFunction.reuse_if_untouched
```

as a default rule.

`positive_restricted``(`*`o`*`)`

class `firedrake.slate.slac.utils.`**`SymbolWithFuncallIndexing`**`(`*`symbol`*`, `*`rank=None`*`, `*`offset=None`*`)`

Bases: `Symbol`

A functionally equivalent representation of a *coffee.Symbol*, with modified output for rank calls. This is syntactically necessary when referring to symbols of Eigen::MatrixBase objects.

class `firedrake.slate.slac.utils.`**`Transformer`**

Bases: `Visitor`

Replaces all out-put tensor references with a specified name of :type: *Eigen::Matrix* with appropriate shape. This class is primarily for COFFEE acrobatics, jumping through nodes and redefining where appropriate.

The default name of `"A"` is assigned, otherwise a specified name may be passed as the `name` keyword argument when calling the visitor.

`visit_Decl``(`*`o`*`, `*`*args`*`, `*`**kwargs`*`)`

Visits a declared tensor and changes its type to :template: result *Eigen::MatrixBase<Derived>*.

i.e. double A[n][m] —> const Eigen::MatrixBase<Derived> &A_

visit_FunDecl(*o*, **args*, ***kwargs*)

Visits a COFFEE FunDecl object and reconstructs the FunDecl body and header to generate `Eigen::MatrixBase` C++ template functions.

Creates a template function for each subkernel form.

```
template <typename Derived>
static inline void foo(Eigen::MatrixBase<Derived> const & A, ...)
{
  [Body...]
}
```

visit_Node(*o*, **args*, ***kwargs*)

A visit method that reconstructs nodes if their children have changed.

visit_Symbol(*o*, **args*, ***kwargs*)

Visits a COFFEE symbol and redefines it as a Symbol with FunCall indexing.

i.e. A[j][k] —> A(j, k)

visit_list(*o*, **args*, ***kwargs*)

Visits an input of COFFEE objects and returns the complete list of said objects.

visit_object(*o*, **args*, ***kwargs*)

Visits an object and returns it.

e.g. string —> string

firedrake.slate.slac.utils.**depth_first_search**(*graph*, *node*, *visited*, *schedule*)

A recursive depth-first search (DFS) algorithm for traversing a DAG consisting of Slate expressions.

Parameters

- **graph** – A DAG whose nodes (vertices) are Slate expressions with edges connected to dependent expressions.
- **node** – A starting vertex.
- **visited** – A set keeping track of visited nodes.
- **schedule** – A list of reverse-postordered nodes. This list is used to produce a topologically sorted list of Slate nodes.

firedrake.slate.slac.utils.**merge_loopy**(*slate_loopy*, *output_arg*, *builder*, *var2terminal*, *name*)

Merges tsfc loopy kernels and slate loopy kernel into a wrapper kernel.

firedrake.slate.slac.utils.**slate2gem**(*expression*, *options*)

firedrake.slate.slac.utils.**slate_to_gem**(*expression*, *options*)

Convert a slate expression to gem.

Parameters

expression – A slate expression.

Returns
A singleton list of gem expressions and a mapping from gem variables to UFL "terminal" forms.

`firedrake.slate.slac.utils.`**`topological_sort`**`(`*exprs*`)`

Topologically sorts a list of Slate expressions. The expression graph is constructed by relating each Slate node with a list of dependent Slate nodes.

Parameters
`exprs` – A list of Slate expressions.

Module contents

firedrake.slate.static_condensation package

Submodules

firedrake.slate.static_condensation.hybridization module

`class` `firedrake.slate.static_condensation.hybridization.`**`HybridizationPC`**

Bases: *`SCBase`*

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- *`initialize()`*
- *`update()`*
- `apply()`
- `applyTranspose()`

`backward_substitution``(`*pc*, *y*`)`

Perform the backwards recovery of eliminated fields.

Parameters

- **`pc`** – a Preconditioner instance.
- **`y`** – a PETSc vector for placing the resulting fields.

`forward_elimination``(`*pc*, *x*`)`

Perform the forward elimination of fields and provide the reduced right-hand side for the condensed system.

Parameters

- **`pc`** – a Preconditioner instance.
- **`x`** – a PETSc vector containing the incoming right-hand side.

`getSchurComplementBuilder``()`

initialize(*pc*)

Set up the problem context. Take the original mixed problem and reformulate the problem as a hybridized mixed system.

A KSP is created for the Lagrange multiplier system.

needs_python_pmat = True

A Slate-based python preconditioner that solves a mixed H(div)-conforming problem using hybridization. Currently, this preconditioner supports the hybridization of the RT and BDM mixed methods of arbitrary degree.

The forward eliminations and backwards reconstructions are performed element-local using the Slate language.

sc_solve(*pc*)

Solve the condensed linear system for the condensed field.

Parameters

pc – a Preconditioner instance.

update(*pc*)

Update by assembling into the operator. No need to reconstruct symbolic objects.

view(*pc*, *viewer=None*)

Viewer calls for the various configurable objects in this PC.

class firedrake.slate.static_condensation.hybridization.**SchurComplementBuilder**(*prefix*, *At-ilde*, *K*, *KT*, *pc*, *vidx*, *pidx*, *non_zero_sadd*

Bases: `object`

A Slate-based Schur complement expression builder. The expression is used in the trace system solve and parts of it in the reconstruction calls of the other two variables of the hybridised system. How the Schur complement if constructed, and in particular how the local inverse of the mixed matrix is built, is controlled with PETSc options. All corresponding PETSc options start with `hybridization_localsolve`. The following option sets are valid together with the usual set of hybridisation options:

```
{'localsolve': {'ksp_type': 'preonly',
                'pc_type': 'fieldsplit',
                'pc_fieldsplit_type': 'schur'}}
```

A Schur complement is requested for the mixed matrix inverse which appears inside the Schur complement of the trace system solve. The Schur complements are then nested. For details see defition of *`build_schur()`*. No fieldsplit options are set so all local inverses are calculated explicitly.

```
'localsolve': {'ksp_type': 'preonly',
               'pc_type': 'fieldsplit',
               'pc_fieldsplit_type': 'schur',
               'fieldsplit_1': {'ksp_type': 'default',
                                'pc_type': 'python',
                                'pc_python_type': __name__ + '.DGLaplacian
↪'}}
```

The inverse of the Schur complement inside the Schur decomposition of the mixed matrix inverse is approximated by a default solver (LU in the matrix-explicit case) which is preconditioned by a user-defined operator, e.g. a DG Laplacian, see *build_inner_S_inv()*. So $P_S * S * x = P_S * b$.

```
'localsolve': {'ksp_type': 'preonly',
                'pc_type': 'fieldsplit',
                'pc_fieldsplit_type': 'schur',
                'fieldsplit_1': {'ksp_type': 'default',
                                'pc_type': 'python',
                                'pc_python_type': __name__ + '.DGLaplacian
↪',
                                'aux_ksp_type': 'preonly'}
                                'aux_pc_type': 'jacobi'}}}}
```

The inverse of the Schur complement inside the Schur decomposition of the mixed matrix inverse is approximated by a default solver (LU in the matrix-explicit case) which is preconditioned by a user-defined operator, e.g. a DG Laplacian. The inverse of the preconditioning matrix is approximated through the inverse of only the diagonal of the provided operator, see *build_Sapprox_inv()*. So $diag(P_S).inv * S * x = diag(P_S).inv * b$.

```
'localsolve': {'ksp_type': 'preonly',
               'pc_type': 'fieldsplit',
               'pc_fieldsplit_type': 'schur',
               'fieldsplit_0': {'ksp_type': 'default',
                                'pc_type': 'jacobi'}
```

The inverse of the A_{00} block of the mixed matrix is approximated by a default solver (LU in the matrix-explicit case) which is preconditioned by the diagonal matrix of $A_{00}, see : meth : `build_A00_inv$. So $diag(A_{00}).inv * A_{00} * x = diag(A_{00}).inv * b$.

```
'localsolve': {'ksp_type': 'preonly',
               'pc_type': 'fieldsplit',
               'pc_fieldsplit_type': 'None',
               'fieldsplit_0':  ...
               'fieldsplit_1':  ...
```

All the options for `fieldsplit_` are still valid if `'pc_fieldsplit_type': 'None'`. In this case the mixed matrix inverse which appears inside the Schur complement of the trace system solve is calculated explicitly, but the local inverses of A_{00} and the Schur complement in the reconstructions calls are still treated according to the options in `fieldsplit_`.

build_A00_inv()

Calculates the inverse of A_{00}, the (0,0)-block of the mixed matrix Atilde. The inverse is potentially approximated through a solve which is potentially preconditioned with jacobi.

build_Sapprox_inv()

Calculates the inverse of preconditioner to the Schur complement, which can be either the schur complement approximation provided by the user or jacobi. The inverse is potentially approximated through a solve which is potentially preconditioned with jacobi.

build_inner_S()

Build the inner Schur complement.

build_inner_S_inv()

Calculates the inverse of the schur complement. The inverse is potentially approximated through a solve which is potentially preconditioned with the preconditioner P.

build_schur(*rhs*, *non_zero_saddle_rhs=None*)

The Schur complement in the operators of the trace solve contains the inverse on a mixed system. Users may want this inverse to be treated with another Schur complement.

Let the mixed matrix Atilde be called A here. Then, if a nested schur complement is requested, the inverse of Atilde is rewritten with help of a a Schur decomposition as follows.

```
A.inv=[[I, -A00.inv * A01] * [[A00.inv, 0    ] * [[I,             0]
      [0,  I             ]]  [0,        S.inv]]  [-A10* A00.inv, I]]
      --------------------   -----------------   ------------------
             block1                block2              block3
with the (inner) schur complement S = A11 - A10 * A00.inv * A01
```

inv(*A*, *P*, *prec*, *preonly=False*)

Calculates the inverse of an operator A. The inverse is potentially approximated through a solve which is potentially preconditioned with the preconditioner P if prec is True. The inverse of A may be just approximated with the inverse of P if prec and replace.

retrieve_user_S_approx(*pc*, *usercode*)

Retrieve a user-defined :class:firedrake.preconditioners.AuxiliaryOperator from the PETSc Options, which is an approximation to the Schur complement and its inverse is used to precondition the local solve in the reconstruction calls (e.g.).

firedrake.slate.static_condensation.la_utils module

class firedrake.slate.static_condensation.la_utils.**LAContext**(*lhs*, *rhs*, *field_idx*)

Bases: `tuple`

Context information for systems of equations after applying algebraic transformation via Slate-supported operations. This object provides the symbolic expressions for the transformed linear system of equations.

Parameters

- **lhs** – The resulting expression for the transformed left-hand side matrix.
- **rhs** – The resulting expression for the transformed right-hand side vector.
- **field_idx** – An integer or iterable of integers (if the system is mixed) denoting which field(s) the resulting solution is defined on.

Create new instance of LAContext(lhs, rhs, field_idx)

field_idx

Alias for field number 2

lhs

Alias for field number 0

rhs

Alias for field number 1

class firedrake.slate.static_condensation.la_utils.**SchurComplementBuilder**(*prefix*, *Atilde*, *K*, *KT*, *pc*, *vidx*, *pidx*, *non_zero_saddle_m*

Bases: `object`

A Slate-based Schur complement expression builder. The expression is used in the trace system solve and parts of it in the reconstruction calls of the other two variables of the hybridised system. How the Schur complement if constructed, and in particular how the local inverse of the mixed matrix is built, is controlled with PETSc options. All corresponding PETSc options start with `hybridization_localsolve`. The following option sets are valid together with the usual set of hybridisation options:

```
{'localsolve': {'ksp_type': 'preonly',
                'pc_type': 'fieldsplit',
                'pc_fieldsplit_type': 'schur'}}
```

A Schur complement is requested for the mixed matrix inverse which appears inside the Schur complement of the trace system solve. The Schur complements are then nested. For details see defition of *build_schur()*. No fieldsplit options are set so all local inverses are calculated explicitly.

```
'localsolve': {'ksp_type': 'preonly',
               'pc_type': 'fieldsplit',
               'pc_fieldsplit_type': 'schur',
               'fieldsplit_1': {'ksp_type': 'default',
                                'pc_type': 'python',
                                'pc_python_type': __name__ + '.DGLaplacian
↪'}}
```

The inverse of the Schur complement inside the Schur decomposition of the mixed matrix inverse is approximated by a default solver (LU in the matrix-explicit case) which is preconditioned by a user-defined operator, e.g. a DG Laplacian, see *build_inner_S_inv()*. So $P_S * S * x = P_S * b$.

```
'localsolve': {'ksp_type': 'preonly',
                'pc_type': 'fieldsplit',
                'pc_fieldsplit_type': 'schur',
                'fieldsplit_1': {'ksp_type': 'default',
                                'pc_type': 'python',
                                'pc_python_type': __name__ + '.DGLaplacian
↪',
                                'aux_ksp_type': 'preonly'}
                                'aux_pc_type': 'jacobi'}}}}
```

The inverse of the Schur complement inside the Schur decomposition of the mixed matrix inverse is approximated by a default solver (LU in the matrix-explicit case) which is preconditioned by a user-defined operator, e.g. a DG Laplacian. The inverse of the preconditioning matrix is approximated through the inverse of only the diagonal of the provided operator, see *build_Sapprox_inv()*. So $diag(P_S).inv * S * x = diag(P_S).inv * b$.

```
'localsolve': {'ksp_type': 'preonly',
               'pc_type': 'fieldsplit',
               'pc_fieldsplit_type': 'schur',
               'fieldsplit_0': {'ksp_type': 'default',
                                'pc_type': 'jacobi'}
```

The inverse of the A_{00} block of the mixed matrix is approximated by a default solver (LU in the matrix-explicit case) which is preconditioned by the diagonal matrix of $A_{00}, see : meth : `build_A00_inv$. So $diag(A_{00}).inv * A_{00} * x = diag(A_{00}).inv * b$.

```
'localsolve': {'ksp_type': 'preonly',
               'pc_type': 'fieldsplit',
               'pc_fieldsplit_type': 'None',
               'fieldsplit_0':  ...
               'fieldsplit_1':  ...
```

All the options for `fieldsplit_` are still valid if `'pc_fieldsplit_type':  'None'`. In this case the mixed matrix inverse which appears inside the Schur complement of the trace system solve is calculated explicitly, but the local inverses of A_{00} and the Schur complement in the reconstructions calls are still treated according to the options in `fieldsplit_`.

build_A00_inv()

Calculates the inverse of A_{00}, the (0,0)-block of the mixed matrix Atilde. The inverse is potentially approximated through a solve which is potentially preconditioned with jacobi.

build_Sapprox_inv()

Calculates the inverse of preconditioner to the Schur complement, which can be either the schur complement approximation provided by the user or jacobi. The inverse is potentially approximated through a solve which is potentially preconditioned with jacobi.

build_inner_S()

Build the inner Schur complement.

build_inner_S_inv()

Calculates the inverse of the schur complement. The inverse is potentially approximated through a solve which is potentially preconditioned with the preconditioner P.

build_schur(*rhs*, *non_zero_saddle_rhs=None*)

The Schur complement in the operators of the trace solve contains the inverse on a mixed system. Users may want this inverse to be treated with another Schur complement.

Let the mixed matrix Atilde be called A here. Then, if a nested schur complement is requested, the inverse of Atilde is rewritten with help of a a Schur decomposition as follows.

```
A.inv=[[I, -A00.inv * A01] * [[A00.inv, 0    ] * [[I,            0]
      [0,  I             ]]  [0,        S.inv]]  [-A10* A00.inv, I]]
      --------------------   -----------------   -------------------
             block1                block2              block3
with the (inner) schur complement S = A11 - A10 * A00.inv * A01
```

inv(*A*, *P*, *prec*, *preonly=False*)

Calculates the inverse of an operator A. The inverse is potentially approximated through a solve which is potentially preconditioned with the preconditioner P if prec is True. The inverse of A may be just approximated with the inverse of P if prec and replace.

retrieve_user_S_approx(*pc*, *usercode*)

Retrieve a user-defined :class:firedrake.preconditioners.AuxiliaryOperator from the PETSc Options, which is an approximation to the Schur complement and its inverse is used to precondition the local solve in the reconstruction calls (e.g.).

firedrake.slate.static_condensation.la_utils.**backward_solve**(*A*, *b*, *x*, *schur_builder*, *reconstruct_fields*)

Returns a sequence of linear algebra contexts containing Slate expressions for backwards substitution.

Parameters

- **A** – a *slate.Tensor* corresponding to the mixed UFL operator.
- **b** – a *firedrake.Function* corresponding to the right-hand side.

- **x** – a *firedrake.Function* corresponding to the solution.
- **schur_builder** – a *SchurComplementBuilder*
- **reconstruct_fields** – a *tuple* of indices denoting which fields to reconstruct.

Returns
a list of *LAContext* for the reconstruction

firedrake.slate.static_condensation.la_utils.**condense_and_forward_eliminate**(*A*, *b*, *elim_fields*, *prefix*, *pc*)

Returns Slate expressions for the operator and right-hand side vector after eliminating specified unknowns.

Parameters

- **A** – a *slate.Tensor* corresponding to the mixed UFL operator.
- **b** – a *firedrake.Function* corresponding to the right-hand side.
- **elim_fields** – a *tuple* of indices denoting which fields to eliminate.
- **prefix** – an option prefix for the condensed field.
- **pc** – a Preconditioner instance.

Returns
a tuple of *LAContext* and *SchurComplementBuilder*

firedrake.slate.static_condensation.sc_base module

class firedrake.slate.static_condensation.sc_base.**SCBase**

Bases: *PCBase*

A general-purpose base class for static condensation interfaces.

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- initialize()
- update()
- *apply()*
- *applyTranspose()*

apply(*pc*, *x*, *y*)

Applies the static condensation preconditioner.

Parameters

- **pc** – a Preconditioner instance.
- **x** – A PETSc vector containing the incoming right-hand side.

- **y** – A PETSc vector for the result.

applyTranspose(*pc*, *x*, *y*)

Apply the transpose of the preconditioner.

abstract **backward_substitution**(*pc*, *y*)

Perform the backwards recovery of eliminated fields.

Parameters

- **pc** – a Preconditioner instance.
- **y** – a PETSc vector for placing the resulting fields.

abstract **forward_elimination**(*pc*, *x*)

Perform the forward elimination of fields and provide the reduced right-hand side for the condensed system.

Parameters

- **pc** – a Preconditioner instance.
- **x** – a PETSc vector containing the incoming right-hand side.

abstract **sc_solve**(*pc*)

Solve the condensed linear system for the condensed field.

Parameters

pc – a Preconditioner instance.

firedrake.slate.static_condensation.scpc module

class firedrake.slate.static_condensation.scpc.**SCPC**

Bases: *SCBase*

Create a PC context suitable for PETSc.

Matrix free preconditioners should inherit from this class and implement:

- *initialize()*
- *update()*
- apply()
- applyTranspose()

backward_substitution(*pc*, *y*)

Perform the backwards recovery of eliminated fields.

Parameters

- **pc** – a Preconditioner instance.
- **y** – a PETSc vector for placing the resulting fields.

condensed_system(*A*, *rhs*, *elim_fields*, *prefix*, *pc*)

Forms the condensed linear system by eliminating specified unknowns.

Parameters

- **A** – A Slate Tensor containing the mixed bilinear form.
- **rhs** – A firedrake function for the right-hand side.
- **elim_fields** – An iterable of field indices to eliminate.
- **prefix** – an option prefix for the condensed field.
- **pc** – a Preconditioner instance.

forward_elimination(*pc*, *x*)

Perform the forward elimination of fields and provide the reduced right-hand side for the condensed system.

Parameters

- **pc** – a Preconditioner instance.
- **x** – a PETSc vector containing the incoming right-hand side.

initialize(*pc*)

Set up the problem context. This takes the incoming three-field system and constructs the static condensation operators using Slate expressions.

A KSP is created for the reduced system. The eliminated variables are recovered via back-substitution.

local_solver_calls(*A*, *rhs*, *x*, *elim_fields*, *schur_builder*)

Provides solver callbacks for inverting local operators and reconstructing eliminated fields.

Parameters

- **A** – A Slate Tensor containing the mixed bilinear form.
- **rhs** – A firedrake function for the right-hand side.
- **x** – A firedrake function for the solution.
- **elim_fields** – An iterable of eliminated field indices to recover.
- **schur_builder** – a *SchurComplementBuilder*.

needs_python_pmat = True

A Slate-based python preconditioner implementation of static condensation for problems with up to three fields.

sc_solve(*pc*)

Solve the condensed linear system for the condensed field.

Parameters

pc – a Preconditioner instance.

update(*pc*)

Update by assembling into the KSP operator. No need to reconstruct symbolic objects.

view(*pc*, *viewer=None*)

Viewer calls for the various configurable objects in this PC.

Module contents

Submodules

firedrake.slate.slate module

Slate is a symbolic language defining a framework for performing linear algebra operations on finite element tensors. It is similar in principle to most linear algebra libraries in notation.

The design of Slate was heavily influenced by UFL, and utilizes much of UFL's functionality for FEM-specific form manipulation.

Unlike UFL, however, once forms are assembled into Slate *Tensor* objects, one can utilize the operations defined in Slate to express complicated linear algebra operations (such as the Schur-complement reduction of a block-matrix system).

All Slate expressions are handled by a specialized linear algebra compiler, which interprets expressions and produces C++ kernel functions to be executed within the Firedrake architecture.

`class firedrake.slate.slate.`**`Add`**`(`*A*`,` *B*`)`

Bases: `BinaryOp`

Abstract Slate class representing matrix-matrix, vector-vector
or scalar-scalar addition.

Parameters

- **`A`** – a `TensorBase` object.
- **`B`** – another `TensorBase` object.

Constructor for the Add class.

`arg_function_spaces`

Returns a tuple of function spaces that the tensor is defined on.

`arguments``()`

Returns a tuple of arguments associated with the tensor.

`prec = 1`

`class firedrake.slate.slate.`**`AssembledVector`**`(`*function*`)`

Bases: `TensorBase`

This class is a symbolic representation of an assembled vector of data contained in a `firedrake.Function`.

Parameters
`function` – A firedrake function.

Initialise a cache for stashing results.

Mirrors `Form`.

`arg_function_spaces`

Returns a tuple of function spaces that the tensor is defined on.

arguments()

Returns a tuple of arguments associated with the tensor.

assembled = True

coefficients()

Returns a tuple of coefficients associated with the tensor.

form

property integrals

operands = ()

prec = 0

slate_coefficients()

Returns a tuple of coefficients associated with the tensor.

subdomain_data()

Returns a mapping on the tensor: `{domain:{integral_type: subdomain_data}}`.

terminal = True

ufl_domains()

Returns the integration domains of the integrals associated with the tensor.

class firedrake.slate.slate.**Block**(*tensor*, *indices*)

Bases: `TensorBase`

This class represents a tensor corresponding to particular block of a mixed tensor. Depending on the indices provided, the subblocks can span multiple test/trial spaces.

Parameters

- **tensor** – A (mixed) tensor.
- **indices** – Indices of the test and trial function spaces to extract. This should be a 0-, 1-, or 2-tuple (whose length is equal to the rank of the tensor.) The entries should be an iterable of integer indices.

For example, consider the mixed tensor defined by:

```
n = FacetNormal(m)
U = FunctionSpace(m, "DRT", 1)
V = FunctionSpace(m, "DG", 0)
M = FunctionSpace(m, "DGT", 0)
W = U * V * M
u, p, r = TrialFunctions(W)
w, q, s = TestFunctions(W)
A = Tensor(dot(u, w)*dx + p*div(w)*dx + r*dot(w, n)*dS
           + div(u)*q*dx + p*q*dx + r*s*ds)
```

This describes a block 3x3 mixed tensor of the form:

$$\begin{bmatrix} A & B & C \\ D & E & F \\ G & H & J \end{bmatrix}$$

Providing the 2-tuple ((0, 1), (0, 1)) returns a tensor corresponding to the upper 2x2 block:

$$\begin{bmatrix} A & B \\ D & E \end{bmatrix}$$

More generally, argument indices of the form *(idr, idc)* produces a tensor of block-size *len(idr)* x *len(idc)* spanning the specified test/trial spaces.

Constructor for the Block class.

arg_function_spaces

Returns a tuple of function spaces that the tensor is defined on.

arguments()

Returns a tuple of arguments associated with the tensor.

assembled

coefficients()

Returns a tuple of coefficients associated with the tensor.

form

prec = 0

slate_coefficients()

Returns a tuple of coefficients associated with the tensor.

subdomain_data()

Returns a mapping on the tensor: `{domain:{integral_type: subdomain_data}}`.

terminal

Blocks are only terminal when they sit on Tensors or AssembledVectors

ufl_domains()

Returns the integration domains of the integrals associated with the tensor.

class firedrake.slate.slate.**BlockAssembledVector**(*function*, *expr*, *indices*)

Bases: *AssembledVector*

This class is a symbolic representation of an assembled vector of data contained in a set of `firedrake.Function` s defined on pieces of a split mixed function space.

Parameters

functions – A tuple of firedrake functions.

Initialise a cache for stashing results.

Mirrors `Form`.

arg_function_spaces

Returns a tuple of function spaces associated to the corresponding block.

arguments()

Returns a tuple of arguments associated with the corresponding block.

coefficients()

Returns a tuple of coefficients associated with the tensor.

form

slate_coefficients()

Returns a BlockFunction in a tuple which carries all information to generate the right coefficients and maps.

subdomain_data()

Returns mappings on the tensor: `{domain:{integral_type: subdomain_data}}`.

ufl_domains()

Returns the integration domains of the integrals associated with the tensor.

class firedrake.slate.slate.**DiagonalTensor**(*A*)

Bases: `UnaryOp`

An abstract Slate class representing the diagonal of a tensor.

Warning: This class will raise an error if the tensor is not square.

Constructor for the Diagonal class.

arg_function_spaces

Returns a tuple of function spaces that the tensor is defined on.

arguments()

Returns a tuple of arguments associated with the tensor.

diagonal = True

prec = 0

class firedrake.slate.slate.**Factorization**(*tensor*, *decomposition=None*)

Bases: `TensorBase`

An abstract Slate class for the factorization of matrices. The factorizations available are the following:

(1) LU with full or partial pivoting (‘FullPivLU’ and ‘PartialPivLU’);

(2) QR using Householder reflectors (‘HouseholderQR’) with the option to use column pivoting (‘ColPivHouseholderQR’) or full pivoting (‘FullPivHouseholderQR’);

(3) standard Cholesky (‘LLT’) and stabilized Cholesky factorizations with pivoting (‘LDLT’);

(4) a rank-revealing complete orthogonal decomposition using Householder transformations (‘CompleteOrthogonalDecomposition’); and

(5) singular-valued decompositions (‘JacobiSVD’ and ‘BDCSVD’). For larger matrices, ‘BDCSVD’ is recommended.

Constructor for the Factorization class.

arg_function_spaces

Returns a tuple of function spaces that the tensor is defined on.

arguments()

Returns a tuple of arguments associated with the tensor.

coefficients()

Returns a tuple of coefficients associated with the tensor.

prec = 0

slate_coefficients()

Returns a tuple of coefficients associated with the tensor.

subdomain_data()

Returns a mapping on the tensor: `{domain:{integral_type: subdomain_data}}`.

ufl_domains()

Returns the integration domains of the integrals associated with the tensor.

class firedrake.slate.slate.**Inverse**(*A*)

Bases: `UnaryOp`

An abstract Slate class representing the inverse of a tensor.

Warning: This class will raise an error if the tensor is not square.

Constructor for the Inverse class.

arg_function_spaces

Returns a tuple of function spaces that the tensor is defined on.

arguments()

Returns the expected arguments of the resulting tensor of performing a specific unary operation on a tensor.

class firedrake.slate.slate.**Mul**(*A*, *B*)

Bases: `BinaryOp`

Abstract Slate class representing the interior product or two tensors. By interior product, we mean an operation that results in a tensor of equal or lower rank via performing a contraction on arguments. This includes Matrix-Matrix and Matrix-Vector multiplication.

Parameters

- **A** – a `TensorBase` object.
- **B** – another `TensorBase` object.

Constructor for the Mul class.

arg_function_spaces

Returns a tuple of function spaces that the tensor is defined on.

arguments()

Returns the arguments of a tensor resulting from multiplying two tensors A and B.

prec = 2

class firedrake.slate.slate.**Negative**(**operands*)

Bases: `UnaryOp`

Abstract Slate class representing the negation of a tensor object.

Constructor for the TensorOp class.

arg_function_spaces

Returns a tuple of function spaces that the tensor is defined on.

arguments()

Returns the expected arguments of the resulting tensor of performing a specific unary operation on a tensor.

class firedrake.slate.slate.**Reciprocal**(*A*)

Bases: `UnaryOp`

An abstract Slate class representing the reciprocal of a vector.

Constructor for the Inverse class.

arg_function_spaces

Returns a tuple of function spaces that the tensor is defined on.

arguments()

Returns the expected arguments of the resulting tensor of performing a specific unary operation on a tensor.

prec = 0

class firedrake.slate.slate.**Solve**(*A*, *B*, *decomposition=None*)

Bases: `BinaryOp`

Abstract Slate class describing a local linear system of equations. This object is a direct solver, utilizing the application of the inverse of matrix in a decomposed form.

Parameters

- **A** – The left-hand side operator.
- **B** – The right-hand side.
- **decomposition** – A string denoting the type of matrix decomposition to used. The factorizations available are detailed in the *Factorization* documentation.

Constructor for the Solve class.

arg_function_spaces

Returns a tuple of function spaces that the tensor is defined on.

arguments()

Returns the arguments of a tensor resulting from applying the inverse of A onto B.

prec = 3

class firedrake.slate.slate.**Tensor**(*form*, *diagonal=False*)

Bases: `TensorBase`

This class is a symbolic representation of a finite element tensor derived from a bilinear or linear form. This class implements all supported ranks of general tensor (rank-0, rank-1 and rank-2 tensor objects). This class is the primary user-facing class that the Slate symbolic algebra supports.

Parameters
form – a `ufl.Form` object.

A `ufl.Form` is currently the only supported input of creating a *slate.Tensor* object:

(1) If the form is a bilinear form, namely a form with two `ufl.Argument` objects, then the Slate Tensor will be a rank-2 Matrix.

(2) If the form has one *ufl.Argument* as in the case of a typical linear form, then this will create a rank-1 Vector.

(3) A zero-form will create a rank-0 Scalar.

These are all under the same type *slate.Tensor*. The attribute *self.rank* is used to determine what kind of tensor object is being handled.

Constructor for the Tensor class.

arg_function_spaces

Returns a tuple of function spaces that the tensor is defined on.

arguments()

Returns a tuple of arguments associated with the tensor.

coefficients()

Returns a tuple of coefficients associated with the tensor.

operands = ()

prec = 0

slate_coefficients()

Returns a tuple of coefficients associated with the tensor.

subdomain_data()

Returns a mapping on the tensor: `{domain:{integral_type: subdomain_data}}`.

terminal = True

ufl_domains()

Returns the integration domains of the integrals associated with the tensor.

class firedrake.slate.slate.**Transpose**(**operands*)

Bases: `UnaryOp`

An abstract Slate class representing the transpose of a tensor.

Constructor for the TensorOp class.

arg_function_spaces

Returns a tuple of function spaces that the tensor is defined on.

arguments()

Returns the expected arguments of the resulting tensor of performing a specific unary operation on a tensor.

Module contents

4.1.7 firedrake.slope_limiter package

Submodules

firedrake.slope_limiter.limiter module

class firedrake.slope_limiter.limiter.**Limiter**(*space*)

Bases: object

Abstract Limiter class for all limiters to implement its methods.

Parameters

space – FunctionSpace instance

abstract **apply**(*field*)

Re-computes centroids and applies limiter to given field

abstract **apply_limiter**(*field*)

Only applies limiting loop on the given field

abstract **compute_bounds**(*field*)

Only computes min and max bounds of neighbouring cells

firedrake.slope_limiter.vertex_based_limiter module

class firedrake.slope_limiter.vertex_based_limiter.**VertexBasedLimiter**(*space*)

Bases: *Limiter*

A vertex based limiter for P1DG fields.

This limiter implements the vertex-based limiting scheme described in Dmitri Kuzmin, "A vertex-based hierarchical slope limiter for p-adaptive discontinuous Galerkin methods". J. Comp. Appl. Maths (2010) http://dx.doi.org/10.1016/j.cam.2009.05.028

Initialise limiter

:param space : FunctionSpace instance

apply(*field*)

Re-computes centroids and applies limiter to given field

apply_limiter(*field*)

Only applies limiting loop on the given field

compute_bounds(*field*)

Only computes min and max bounds of neighbouring cells

Module contents

4.2 Submodules

4.3 firedrake.assemble module

`firedrake.assemble.`**`assemble`**(*expr*, **args*, ***kwargs*)

Evaluate expr.

Parameters

- **`expr`** – a `Form`, `Expr` or a `TensorBase` expression.
- **`tensor`** – Existing tensor object to place the result in.
- **`bcs`** – Iterable of boundary conditions to apply.
- **`diagonal`** – If assembling a matrix is it diagonal?
- **`form_compiler_parameters`** – Dictionary of parameters to pass to the form compiler. Ignored if not assembling a `Form`. Any parameters provided here will be overridden by parameters set on the `Measure` in the form. For example, if a `quadrature_degree` of 4 is specified in this argument, but a degree of 3 is requested in the measure, the latter will be used.
- **`mat_type`** – String indicating how a 2-form (matrix) should be assembled – either as a monolithic matrix (`"aij"` or `"baij"`), a block matrix (`"nest"`), or left as a *`ImplicitMatrix`* giving matrix-free actions (`'matfree'`). If not supplied, the default value in `parameters["default_matrix_type"]` is used. BAIJ differs from AIJ in that only the block sparsity rather than the dof sparsity is constructed. This can result in some memory savings, but does not work with all PETSc preconditioners. BAIJ matrices only make sense for non-mixed matrices.
- **`sub_mat_type`** – String indicating the matrix type to use *inside* a nested block matrix. Only makes sense if `mat_type` is `nest`. May be one of `"aij"` or `"baij"`. If not supplied, defaults to `parameters["default_sub_matrix_type"]`.
- **`appctx`** – Additional information to hang on the assembled matrix if an implicit matrix is requested (mat_type `"matfree"`).
- **`options_prefix`** – PETSc options prefix to apply to matrices.
- **`zero_bc_nodes`** – If `True`, set the boundary condition nodes in the output tensor to zero rather than to the values prescribed by the boundary condition. Default is `False`.

Returns

See below.

If expr is a `Form` or Slate tensor expression then this evaluates the corresponding integral(s) and returns a `float` for 0-forms, a *`Function`* for 1-forms and a *`Matrix`* or

ImplicitMatrix for 2-forms. In the case of 2-forms the rows correspond to the test functions and the columns to the trial functions.

If expr is an expression other than a form, it will be evaluated pointwise on the *Functions* in the expression. This will only succeed if all the Functions are on the same *FunctionSpace*.

If `tensor` is supplied, the assembled result will be placed there, otherwise a new object of the appropriate type will be returned.

If `bcs` is supplied and `expr` is a 2-form, the rows and columns of the resulting *Matrix* corresponding to boundary nodes will be set to 0 and the diagonal entries to 1. If `expr` is a 1-form, the vector entries at boundary nodes are set to the boundary condition values.

Note: For 1-form assembly, the resulting object should in fact be a *cofunction* instead of a *Function*. However, since cofunctions are not currently supported in UFL, functions are used instead.

4.4 firedrake.assign module

class firedrake.assign.**Assigner**(*assignee*, *expression*, *subset=None*)

Bases: `object`

Class performing pointwise assignment of an expression to a `firedrake.Function`.

Parameters

- **assignee** – The `firedrake.Function` being assigned to.
- **expression** – The `ufl.Expr` to evaluate.
- **subset** – Optional subset (`op2.Subset`) to apply the assignment over.

assign()

Perform the assignment.

symbol = '='

class firedrake.assign.**CoefficientCollector**

Bases: `MultiFunction`

Multifunction used for converting an expression into a weighted sum of coefficients.

Calling `map_expr_dag(CoefficientCollector(), expr)` will return a tuple whose entries are of the form `(coefficient, weight)`. Expressions that cannot be expressed as a weighted sum will raise an exception.

Note: As well as being simple weighted sums (e.g. `u.assign(2*v1 + 3*v2)`), one can also assign constant expressions of the appropriate shape (e.g. `u.assign(1.0)` or `u.assign(2*v + 3)`). Therefore the returned tuple must be split since `coefficient` may be either a *firedrake.constant.Constant* or *firedrake.function.Function*.

abs(*o*, *a*)

coefficient(*o*)

component_tensor(*o*, *a*, *_*)

division(*o*, *a*, *b*)

expr(*o*, **operands*)

Trigger error for types with missing handlers.

float_value(*o*)

indexed(*o*, *a*, *_*)

int_value(*o*)

multi_index(*o*)

power(*o*, *a*, *b*)

product(*o*, *a*, *b*)

sum(*o*, *a*, *b*)

zero(*o*)

class firedrake.assign.**IAddAssigner**(*assignee*, *expression*, *subset=None*)

Bases: *Assigner*

Assigner class for `firedrake.Function.__iadd__()`.

symbol = '+='

class firedrake.assign.**IDivAssigner**(*assignee*, *expression*, *subset=None*)

Bases: *Assigner*

Assigner class for `firedrake.Function.__itruediv__()`.

symbol = '/='

class firedrake.assign.**IMulAssigner**(*assignee*, *expression*, *subset=None*)

Bases: *Assigner*

Assigner class for `firedrake.Function.__imul__()`.

symbol = '*='

class firedrake.assign.**ISubAssigner**(*assignee*, *expression*, *subset=None*)

Bases: *Assigner*

Assigner class for `firedrake.Function.__isub__()`.

symbol = '-='

4.5 firedrake.bcs module

class `firedrake.bcs.`**`DirichletBC`**(*V*, *g*, *sub_domain*, *method=None*)

Bases: `BCBase`, *`DirichletBCMixin`*

Implementation of a strong Dirichlet boundary condition.

Parameters

- **V** – the *`FunctionSpace`* on which the boundary condition should be applied.
- **g** – the boundary condition values. This can be a *`Function`* on `V`, or a UFL expression that can be interpolated into `V`, for example, a *`Constant`*, an iterable of literal constants (converted to a UFL expression), or a literal constant which can be pointwise evaluated at the nodes of `V`.
- **sub_domain** – the integer id(s) of the boundary region over which the boundary condition should be applied. The string "on_boundary" may be used to indicate all of the boundaries of the domain. In the case of extrusion the `top` and `bottom` strings are used to flag the bcs application on the top and bottom boundaries of the extruded mesh respectively.
- **method** – the method for determining boundary nodes. DEPRECATED. The only way boundary nodes are identified is by topological association.

`apply`(*r*, *u=None*)

Apply this boundary condition to `r`.

Parameters

- **r** – a *`Function`* or *`Matrix`* to which the boundary condition should be applied.
- **u** – an optional current state. If `u` is supplied then `r` is taken to be a residual and the boundary condition nodes are set to the value `u-bc`. Supplying `u` has no effect if `r` is a *`Matrix`* rather than a *`Function`*. If `u` is absent, then the boundary condition nodes of `r` are set to the boundary condition values.

If `r` is a *`Matrix`*, it will be assembled with a 1 on diagonals where the boundary condition applies and 0 in the corresponding rows and columns.

`dirichlet_bcs`()

`extract_form`(*form_type*)

property function_arg

The value of this boundary condition.

`homogenize`()

Convert this boundary condition into a homogeneous one.

Set the value to zero.

integrals()

reconstruct(*field=None*, *V=None*, *g=None*, *sub_domain=None*, *use_split=False*)

restore()

Restore the original value of this boundary condition.

This uses the value passed on instantiation of the object.

set_value(*val*)

Set the value of this boundary condition.

Parameters

val – The boundary condition values. See *DirichletBC* for valid values.

class firedrake.bcs.**EquationBC**(**args*, *bcs=None*, *J=None*, *Jp=None*, *V=None*, *is_linear=False*, *Jp_eq_J=False*)

Bases: `object`

Construct and store EquationBCSplit objects (for *F*, *J*, and *Jp*).

Parameters

- **eq** – the linear/nonlinear form equation
- **u** – the *Function* to solve for
- **sub_domain** – see *DirichletBC*.
- **bcs** – a list of `DirichletBC`s and/or :class:`.EquationBC`s to be applied to this boundary condition equation (optional)
- **J** – the Jacobian for this boundary equation (optional)
- **Jp** – a form used for preconditioning the linear system, optional, if not supplied then the Jacobian itself will be used.
- **V** – the *FunctionSpace* on which the equation boundary condition is applied (optional)
- **is_linear** – this flag is used only with the *reconstruct* method
- **Jp_eq_J** – this flag is used only with the *reconstruct* method

dirichlet_bcs()

extract_form(*form_type*)

Return `EquationBCSplit` associated with the given 'form_type'.

Parameters

form_type – Form to extract; 'F', 'J', or 'Jp'.

reconstruct(*V*, *subu*, *u*, *field*)

firedrake.bcs.**homogenize**(*bc*)

Create a homogeneous version of a *DirichletBC* object and return it. If `bc` is an iterable containing one or more *DirichletBC* objects, then return a list of the homogeneous versions of those *DirichletBC*s.

Parameters

bc – a *DirichletBC*, or iterable object comprising *DirichletBC*(s).

4.6 firedrake.checkpointing module

class firedrake.checkpointing.**CheckpointFile**(*filename*, *mode*, *comm=<mpi4py.MPI.Intracomm object>*)

Bases: `object`

Checkpointing meshes and *Function* s in an HDF5 file.

Parameters

- **filename** – the name of the HDF5 checkpoint file (.h5 or .hdf5).
- **mode** – the file access mode (*FILE_READ*, *FILE_CREATE*, *FILE_UPDATE*) or ('r', 'w', 'a').
- **comm** – the communicator.

This object allows for a scalable and flexible checkpointing of states. One can save and load meshes and *Function* s entirely in parallel without needing to gather them to or scatter them from a single process. One can also use different number of processes for saving and for loading.

close()

Close the checkpoint file.

create_group(*name*, *track_order=None*)

Mimic `h5py.Group.create_group()`.

Parameters

- **name** – The name of the group.
- **track_order** – Whether to track dataset/group/attribute creation order.

In this method we customise the `h5py.h5p.PropGCID` object from which we create the *h5py.h5g.GroupID* object to avoid the "object header message is too large" error and/or "record is not in B-tree" error when storing many (hundreds of) attributes; see this PR.

TODO: Lift this to upstream somehow.

get_attr(*path*, *key*)

Get an HDF5 attribute at specified path.

Parameters

- **path** – The path at which the attribute is found.
- **key** – The attribute key.

Returns

The attribute value.

property h5pyfile

An h5py File object pointing at the open file handle.

has_attr(*path*, *key*)

Check if an HDF5 attribute exists at specified path.

Parameters

- **path** – The path at which the attribute is sought.
- **key** – The attribute key.

Returns

True if the attribute is found.

load_function(*mesh*, *name*, *idx=None*)

Load a `Function` defined on *mesh*.

Parameters

- **mesh** – the mesh on which the function is defined.
- **name** – the name of the `Function` to load.
- **idx** – optional timestepping index. A function can be loaded with idx only when it was saved with idx.

Returns

the loaded `Function`.

load_mesh(*name='firedrake_default'*, *reorder=None*, *distribution_parameters=None*)

Load a mesh.

Parameters

- **name** – the name of the mesh to load (default to `DEFAULT_MESH_NAME`).
- **reorder** – whether to reorder the mesh (bool); see `Mesh()`.
- **distribution_parameters** – the *distribution_parameters* used for distributing the mesh; see `Mesh()`.

Returns

the loaded mesh.

opts

DMPlex HDF5 version options.

require_group(*name*)

Mimic `h5py.Group.require_group()`.

Parameters

name – name of the group.

This method uses `create_group()` instead of `h5py.Group.create_group()` to create an `h5py.Group` object from an `h5py.h5g.GroupID` constructed with a custom `h5py.h5p.PropGCID` object (often named *gcpl*); see `h5py.Group.create_group()`.

TODO: Lift this to upstream somehow.

save_function(*f*, *idx=None*, *name=None*)

Save a `Function`.

Parameters

- **f** – the *Function* to save.
- **idx** – optional timestepping index. A function can either be saved in timestepping mode or in normal mode (non-timestepping); for each function of interest, this method must always be called with the idx parameter set or never be called with the idx parameter set.
- **name** – optional alternative name to save the function under.

save_mesh(*mesh*, *distribution_name=None*, *permutation_name=None*)

Save a mesh.

Parameters

- **mesh** – the mesh to save.
- **distribution_name** – the name under which distribution is saved; if *None*, auto-generated name will be used.
- **permutation_name** – the name under which permutation is saved; if *None*, auto-generated name will be used.

set_attr(*path*, *key*, *val*)

Set an HDF5 attribute at specified path.

Parameters

- **path** – The path at which the attribute is set.
- **key** – The attribute key.
- **val** – The attribute value.

class firedrake.checkpointing.**DumbCheckpoint**(*basename*, *single_file=True*, *mode=2*, *comm=None*)

Bases: `object`

A very dumb checkpoint object.

This checkpoint object is capable of writing *Function*s to disk in parallel (using HDF5) and reloading them on the same number of processes and a *Mesh()* constructed identically.

Parameters

- **basename** – the base name of the checkpoint file.
- **single_file** – Should the checkpoint object use only a single on-disk file (irrespective of the number of stored timesteps)? See *new_file()* for more details.
- **mode** – the access mode (one of *FILE_READ*, *FILE_CREATE*, or *FILE_UPDATE*)
- **comm** – (optional) communicator the writes should be collective over.

This object can be used in a context manager (in which case it closes the file when the scope is exited).

Note: This object contains both a PETSc `Viewer`, used for storing and loading *Function* data, and an `h5py:File` opened on the same file handle. *DO NOT* call `h5py:File.close()` on the latter, this will cause breakages.

> **Warning:** DumbCheckpoint class will be deprecated after 01/01/2023. Use *`CheckpointFile`* class instead.

close()

Close the checkpoint file (flushing any pending writes)

get_timesteps()

Return all the time steps (and time indices) in the current checkpoint file.

This is useful when reloading from a checkpoint file that contains multiple timesteps and one wishes to determine the final available timestep in the file.

property h5file

An h5py File object pointing at the open file handle.

has_attribute(*obj*, *name*)

Check for existance of an HDF5 attribute on a specified data object.

Parameters

- **obj** – The path to the data object.
- **name** – The name of the attribute.

load(*function*, *name=None*)

Store a function from the checkpoint file.

Parameters

- **function** – The function to load values into.
- **name** – an (optional) name used to find the function values. If not provided, uses `function.name()`.

This function is timestep-aware and reads from the appropriate place if *`set_timestep()`* has been called.

new_file(*name=None*)

Open a new on-disk file for writing checkpoint data.

Parameters

name – An optional name to use for the file, an extension of `.h5` is automatically appended.

If `name` is not provided, a filename is generated from the `basename` used when creating the *`DumbCheckpoint`* object. If `single_file` is `True`, then we write to `BASENAME.h5` otherwise, each time *`new_file()`* is called, we create a new file with an increasing index. In this case the files created are:

```
BASENAME_0.h5
BASENAME_1.h5
```

(continues on next page)

(continued from previous page)

```
...
BASENAME_n.h5
```

with the index incremented on each invocation of *new_file()* (whenever the custom name is not provided).

read_attribute(*obj*, *name*, *default=None*)

Read an HDF5 attribute on a specified data object.

Parameters

- **obj** – The path to the data object.
- **name** – The name of the attribute.
- **default** – Optional default value to return. If not provided an `AttributeError` is raised if the attribute does not exist.

set_timestep(*t*, *idx=None*)

Set the timestep for output.

Parameters

- **t** – The timestep value.
- **idx** – An optional timestep index to use, otherwise an internal index is used, incremented by 1 every time *set_timestep()* is called.

store(*function*, *name=None*)

Store a function in the checkpoint file.

Parameters

- **function** – The function to store.
- **name** – an (optional) name to store the function under. If not provided, uses `function.name()`.

This function is timestep-aware and stores to the appropriate place if *set_timestep()* has been called.

property vwr

The PETSc Viewer used to store and load function data.

write_attribute(*obj*, *name*, *val*)

Set an HDF5 attribute on a specified data object.

Parameters

- **obj** – The path to the data object.
- **name** – The name of the attribute.
- **val** – The attribute value.

Raises `AttributeError` if writing the attribute fails.

firedrake.checkpointing.**FILE_CREATE = 1**

Create a checkpoint file. Truncates the file if it exists.

`firedrake.checkpointing.`**`FILE_READ = 0`**

Open a checkpoint file for reading. Raises an error if file does not exist.

`firedrake.checkpointing.`**`FILE_UPDATE = 2`**

Open a checkpoint file for updating. Creates the file if it does not exist, providing both read and write access.

`class` `firedrake.checkpointing.`**`HDF5File`**`(`*filename*, *file_mode*, *comm=None*`)`

Bases: `object`

An object to facilitate checkpointing.

This checkpoint object is capable of writing *`Function`*s to disk in parallel (using HDF5) and reloading them on the same number of processes and a *`Mesh()`* constructed identically.

Parameters

- **`filename`** – filename (including suffix .h5) of checkpoint file.
- **`file_mode`** – the access mode, passed directly to h5py, see `h5py:File` for details on the meaning.
- **`comm`** – communicator the writes should be collective over.

This object can be used in a context manager (in which case it closes the file when the scope is exited).

Warning: HDF5File class will be deprecated after 01/01/2023. Use *`CheckpointFile`* class instead.

`attributes``(`*obj*`)`

Parameters
`obj` – The path to the group.

`close``()`

Close the checkpoint file (flushing any pending writes)

`flush``()`

Flush any pending writes.

`get_timestamps``()`

Get the timestamps this HDF5File knows about.

`read``(`*function*, *path*, *timestamp=None*`)`

Store a function from the checkpoint file.

Parameters

- **`function`** – The function to load values into.
- **`path`** – the path under which the function is stored.

`write``(`*function*, *path*, *timestamp=None*`)`

Store a function in the checkpoint file.

Parameters

- **function** – The function to store.
- **path** – the path to store the function under.
- **timestamp** – timestamp associated with function, or None for stationary data

4.7 firedrake.constant module

class firedrake.constant.**Constant**(**args*, ***kwargs*)

Bases: `Coefficient`, *ConstantMixin*

A "constant" coefficient

A *Constant* takes one value over the whole *Mesh()*. The advantage of using a *Constant* in a form rather than a literal value is that the constant will be passed as an argument to the generated kernel which avoids the need to recompile the kernel if the form is assembled for a different value of the constant.

Parameters

- **value** – the value of the constant. May either be a scalar, an iterable of values (for a vector-valued constant), or an iterable of iterables (or numpy array with 2-dimensional shape) for a tensor-valued constant.
- **domain** – an optional *Mesh()* on which the constant is defined.

Note: If you intend to use this *Constant* in a `Form` on its own you need to pass a *Mesh()* as the domain argument.

assign(*value*)

Set the value of this constant.

Parameters
value – A value of the appropriate shape

cell_node_map(*bcs=None*)

Return a null cell to node map.

evaluate(*x*, *mapping*, *component*, *index_values*)

Return the evaluation of this *Constant*.

Parameters

- **x** – The coordinate to evaluate at (ignored).
- **mapping** – A mapping (ignored).
- **component** – The requested component of the constant (may be `None` or `()` to obtain all components).
- **index_values** – ignored.

exterior_facet_node_map(*bcs=None*)

Return a null exterior facet to node map.

function_space()

Return a null function space.

interior_facet_node_map(*bcs=None*)

Return a null interior facet to node map.

split()

values()

Return a (flat) view of the value of the Constant.

4.8 firedrake.dmhooks module

Firedrake uses PETSc for its linear and nonlinear solvers. The interaction is carried out through DM objects. These carry around any user-defined application context and can be used to inform the solvers how to create field decompositions (for fieldsplit preconditioning) as well as creating sub-DMs (which only contain some fields), along with multilevel information (for geometric multigrid)

The way Firedrake interacts with these DMs is, broadly, as follows:

A DM is tied to a *FunctionSpace* and remembers what function space that is. To avoid reference cycles defeating the garbage collector, the DM holds a weakref to the FunctionSpace (which holds a strong reference to the DM). Use *get_function_space()* to get the function space attached to the DM, and *set_function_space()* to attach it.

Similarly, when a DM is used in a solver, an application context is attached to it, such that when PETSc calls back into Firedrake, we can grab the relevant information (how to make the Jacobian, etc…). This functions in a similar way using *push_appctx()* and *get_appctx()* on the DM. You can set whatever you like in here, but most of the rest of Firedrake expects to find either `None` or else a `firedrake.solving_utils._SNESContext` object.

A crucial part of this, for composition with multi-level solvers (`-pc_type mg` and `-snes_type fas`) is decomposing the DMs. When a field decomposition is created, the callback *create_field_decomposition()* checks to see if an application context exists. If so, it splits it apart (one for each of fields) and attaches these split contexts to the subdms returned to PETSc. This facilitates runtime composition with multilevel solvers. When coarsening a DM, the application context is coarsened and transferred to the coarse DM. The combination of these two symbolic transfer operations allow us to nest geometric multigrid preconditioning inside fieldsplit preconditioning, without having to set everything up in advance.

class firedrake.dmhooks.**SetupHooks**

Bases: `object`

Hooks run for setup and teardown of DMs inside solvers.

Used for transferring problem-specific data onto subproblems.

You probably don't want to use this directly, instead see *add_hooks* or *add_hook()*.

add_setup(*f*)

add_teardown(*f*)

setup()

teardown()

firedrake.dmhooks.**add_hook**(*dm*, *setup=None*, *teardown=None*, *call_setup=False*, *call_teardown=False*)

Add a hook to a DM to be called for setup/teardown of subproblems.

Parameters

- **dm** – The DM to save the hooks on. This is normally the DM associated with the Firedrake solver.
- **setup** – function of no arguments to call to set up subproblem data.
- **teardown** – function of no arguments to call to remove subproblem data.
- **call_setup** – Should the setup function be called now?
- **call_teardown** – Should the teardown function be called now?

See also *add_hooks* which provides a context manager which manages everything.

class firedrake.dmhooks.**add_hooks**(*dm*, *obj*, ***, *save=True*, *appctx=None*)

Bases: object

Context manager for adding subproblem setup hooks to a DM.

Parameters

- **DM** – The DM to remember setup/teardown for.
- **obj** – The object that we're going to setup, typically a solver of some kind: this is where the hooks are saved.
- **save** – Save this round of setup? Set this to False if all you're going to do is setFromOptions.
- **appctx** – An application context to attach to the top-level DM that describes the problem-specific data.

This is your normal entry-point for setting up problem specific data on subdms. You would likely do something like, for a Python PC.

```
# In setup
pc = ...
pc.setDM(dm)
with dmhooks.add_hooks(dm, self, appctx=ctx, save=False):
    pc.setFromOptions()

...

# in apply
dm = pc.getDM()
with dmhooks.add_hooks(dm, self, appctx=self.ctx):
   pc.apply(...)
```

firedrake.dmhooks.**attach_hooks**(*dm*, *level=None*, *sf=None*, *section=None*)

Attach callback hooks to a DM.

Parameters

- **DM** – The DM to attach callbacks to.
- **level** – Optional refinement level.
- **sf** – Optional PETSc SF object describing the DM's `points`.
- **section** – Optional PETSc Section object describing the DM's data layout.

`firedrake.dmhooks.`**`coarsen`**(*dm*, *comm*)

Callback to coarsen a DM.

Parameters

- **DM** – The DM to coarsen.
- **comm** – The communicator for the new DM (ignored)

This transfers a coarse application context over to the coarsened DM (if found on the input DM).

`firedrake.dmhooks.`**`create_field_decomposition`**(*dm*, **args*, ***kwargs*)

Callback to decompose a DM.

Parameters

DM – The DM.

This grabs the function space in the DM, splits it apart (only makes sense for mixed function spaces) and returns the DMs on each of the subspaces. If an application context is present on the input DM, it is split into individual field contexts and set on the appropriate subdms as well.

`firedrake.dmhooks.`**`create_matrix`**(*dm*)

Callback to create a matrix from this DM.

Parameters

DM – The DM.

Note: This only works if an application context is set, in which case it returns the stored Jacobian. This *does not* make a new matrix.

`firedrake.dmhooks.`**`create_subdm`**(*dm*, *fields*, **args*, ***kwargs*)

Callback to create a sub-DM describing the specified fields.

Parameters

- **DM** – The DM.
- **fields** – The fields in the new sub-DM.

class `firedrake.dmhooks.`**`ctx_coarsener`**(*V*, *coarsen=None*)

Bases: `object`

`firedrake.dmhooks.`**`get_appctx`**(*dm*, *default=None*)

`firedrake.dmhooks.`**`get_attr`**(*attr*, *dm*, *default=None*)

`firedrake.dmhooks.`**`get_ctx_coarsener`**(*dm*) `firedrake.dmhooks.`**`get_function_space`**`(`*dm*`)`

Get the *`FunctionSpace`* attached to this DM.

Parameters
`dm` – The DM to get the function space from.

Raises
`RuntimeError` – if no function space was found.

`firedrake.dmhooks.`**`get_parent`**`(`*dm*`)`

`firedrake.dmhooks.`**`get_transfer_manager`**`(`*dm*`)`

`firedrake.dmhooks.`**`pop_appctx`**`(`*dm*, *match=None*`)`

`firedrake.dmhooks.`**`pop_attr`**`(`*attr*, *dm*, *match=None*`)`

`firedrake.dmhooks.`**`pop_ctx_coarsener`**`(`*dm*, *match=None*`)`

`firedrake.dmhooks.`**`pop_parent`**`(`*dm*, *match=None*`)`

`firedrake.dmhooks.`**`push_appctx`**`(`*dm*, *obj*`)`

`firedrake.dmhooks.`**`push_attr`**`(`*attr*, *dm*, *obj*`)`

`firedrake.dmhooks.`**`push_ctx_coarsener`**`(`*dm*, *obj*`)`

`firedrake.dmhooks.`**`push_parent`**`(`*dm*, *obj*`)`

`firedrake.dmhooks.`**`refine`**`(`*dm*, *comm*`)`

Callback to refine a DM.

Parameters

- **`DM`** – The DM to refine.
- **`comm`** – The communicator for the new DM (ignored)

`firedrake.dmhooks.`**`set_function_space`**`(`*dm*, *V*`)`

Set the *`FunctionSpace`* on this DM.

Parameters

- **`dm`** – The DM
- **`V`** – The function space.

Note: This stores the information necessary to make a function space given a DM.

4.9 firedrake.embedding module

Module for utility functions for scalable HDF5 I/O.

`firedrake.embedding.`**`get_embedding_dg_element`**(*element*)

`firedrake.embedding.`**`get_embedding_element_for_checkpointing`**(*element*)

Convert the given UFL element to an element that *`CheckpointFile`* can handle.

`firedrake.embedding.`**`get_embedding_method_for_checkpointing`**(*element*)

Return the method used to embed element in dg space.

4.10 firedrake.ensemble module

`class` `firedrake.ensemble.`**`Ensemble`**(*comm*, *M*)

Bases: `object`

Create a set of space and ensemble subcommunicators.

Parameters

- **`comm`** – The communicator to split.
- **`M`** – the size of the communicators used for spatial parallelism.

Raises

`ValueError` – if `M` does not divide `comm.size` exactly.

`allreduce`(*f*, *f_reduced*, *op=<mpi4py.MPI.Op object>*)

Allreduce a function f into f_reduced over `ensemble_comm`.

Parameters

- **`f`** – The a *`Function`* to allreduce.
- **`f_reduced`** – the result of the reduction.
- **`op`** – MPI reduction operator. Defaults to MPI.SUM.

Raises

`ValueError` – if function communicators mismatch each other or the ensemble spatial communicator, or if the functions are in different spaces

`bcast`(*f*, *root=0*)

Broadcast a function f over `ensemble_comm` from rank root

Parameters

- **`f`** – The *`Function`* to broadcast.
- **`root`** – rank to broadcast from. Defaults to 0.

Raises

`ValueError` – if function communicator mismatches the ensemble spatial communicator.

iallreduce(*f*, *f_reduced*, *op=<mpi4py.MPI.Op object>*)

Allreduce (non-blocking) a function f into f_reduced over `ensemble_comm`.

Parameters

- **f** – The a *Function* to allreduce.
- **f_reduced** – the result of the reduction.
- **op** – MPI reduction operator. Defaults to MPI.SUM.

Returns

list of MPI.Request objects (one for each of f.split()).

Raises

ValueError – if function communicators mismatch each other or the ensemble spatial communicator, or if the functions are in different spaces

ibcast(*f*, *root=0*)

Broadcast (non-blocking) a function f over `ensemble_comm` from rank root

Parameters

- **f** – The *Function* to broadcast.
- **root** – rank to broadcast from. Defaults to 0.

Returns

list of MPI.Request objects (one for each of f.split()).

Raises

ValueError – if function communicator mismatches the ensemble spatial communicator.

irecv(*f*, *source=-2*, *tag=-1*)

Receive (non-blocking) a function f over `ensemble_comm` from another ensemble rank.

Parameters

- **f** – The a *Function* to receive into
- **source** – the rank to receive from. Defaults to MPI.ANY_SOURCE.
- **tag** – the tag of the message. Defaults to MPI.ANY_TAG.

Returns

list of MPI.Request objects (one for each of f.split()).

Raises

ValueError – if function communicator mismatches the ensemble spatial communicator.

ireduce(*f*, *f_reduced*, *op=<mpi4py.MPI.Op object>*, *root=0*)

Reduce (non-blocking) a function f into f_reduced over `ensemble_comm` to rank root

Parameters

- **f** – The a *Function* to reduce.
- **f_reduced** – the result of the reduction on rank root.

- **op** – MPI reduction operator. Defaults to MPI.SUM.
- **root** – rank to reduce to. Defaults to 0.

Returns
list of MPI.Request objects (one for each of f.split()).

Raises
ValueError – if function communicators mismatch each other or the ensemble spatial communicator, or is the functions are in different spaces

isend(*f*, *dest*, *tag=0*)
Send (non-blocking) a function f over `ensemble_comm` to another ensemble rank.

Parameters

- **f** – The a *Function* to send
- **dest** – the rank to send to
- **tag** – the tag of the message. Defaults to 0.

Returns
list of MPI.Request objects (one for each of f.split()).

Raises
ValueError – if function communicator mismatches the ensemble spatial communicator.

isendrecv(*fsend*, *dest*, *sendtag=0*, *frecv=None*, *source=-2*, *recvtag=-1*)
Send a function fsend and receive a function frecv over `ensemble_comm` to another ensemble rank.

Parameters

- **fsend** – The a *Function* to send.
- **dest** – the rank to send to.
- **sendtag** – the tag of the send message. Defaults to 0.
- **frecv** – The a *Function* to receive into.
- **source** – the rank to receive from. Defaults to MPI.ANY_SOURCE.
- **recvtag** – the tag of the received message. Defaults to MPI.ANY_TAG.

Returns
list of MPI.Request objects (one for each of fsend.split() and frecv.split()).

Raises
ValueError – if function communicator mismatches the ensemble spatial communicator.

recv(*f*, *source=-2*, *tag=-1*, *statuses=None*)
Receive (blocking) a function f over `ensemble_comm` from another ensemble rank.

Parameters

- **f** – The a *Function* to receive into

- **source** – the rank to receive from. Defaults to MPI.ANY_SOURCE.
- **tag** – the tag of the message. Defaults to MPI.ANY_TAG.
- **statuses** – MPI.Status objects (one for each of f.split() or None).

Raises
`ValueError` – if function communicator mismatches the ensemble spatial communicator.

reduce(*f*, *f_reduced*, *op=<mpi4py.MPI.Op object>*, *root=0*)

Reduce a function f into f_reduced over `ensemble_comm` to rank root

Parameters

- **f** – The a *Function* to reduce.
- **f_reduced** – the result of the reduction on rank root.
- **op** – MPI reduction operator. Defaults to MPI.SUM.
- **root** – rank to reduce to. Defaults to 0.

Raises
`ValueError` – if function communicators mismatch each other or the ensemble spatial communicator, or is the functions are in different spaces

send(*f*, *dest*, *tag=0*)

Send (blocking) a function f over `ensemble_comm` to another ensemble rank.

Parameters

- **f** – The a *Function* to send
- **dest** – the rank to send to
- **tag** – the tag of the message. Defaults to 0

Raises
`ValueError` – if function communicator mismatches the ensemble spatial communicator.

sendrecv(*fsend*, *dest*, *sendtag=0*, *frecv=None*, *source=-2*, *recvtag=-1*, *status=None*)

Send (blocking) a function fsend and receive a function frecv over `ensemble_comm` to another ensemble rank.

Parameters

- **fsend** – The a *Function* to send.
- **dest** – the rank to send to.
- **sendtag** – the tag of the send message. Defaults to 0.
- **frecv** – The a *Function* to receive into.
- **source** – the rank to receive from. Defaults to MPI.ANY_SOURCE.
- **recvtag** – the tag of the received message. Defaults to MPI.ANY_TAG.
- **status** – MPI.Status object or None.

Raises
: `ValueError` – if function communicator mismatches the ensemble spatial communicator.

4.11 firedrake.exceptions module

exception `firedrake.exceptions.`**`ConvergenceError`**

Bases: `Exception`

Error raised when a solver fails to converge

4.12 firedrake.extrusion_utils module

`firedrake.extrusion_utils.`**`calculate_dof_offset`**`(`*finat_element*`)`

Return the offset between the neighbouring cells of a column for each DoF.

Parameters
: **`finat_element`** – A FInAT element.

Returns
: A numpy array containing the offset for each DoF.

`firedrake.extrusion_utils.`**`entity_closures`**`(`*cell*`)`

Map entities in a cell to points in the topological closure of the entity.

Parameters
: **`cell`** – a FIAT cell.

`firedrake.extrusion_utils.`**`entity_indices`**`(`*cell*`)`

Return a dict mapping topological entities on a cell to their integer index.

This provides an iteration ordering for entities on extruded meshes.

Parameters
: **`cell`** – a FIAT cell.

`firedrake.extrusion_utils.`**`entity_reordering`**`(`*cell*`)`

Return an array reordering extruded cell entities.

If we iterate over the base cell, it is natural to then go over all the entities induced by the product with an interval. This iteration order is not the same as the natural iteration order, so we need a reordering.

Parameters
: **`cell`** – a FIAT tensor product cell.

`firedrake.extrusion_utils.`**`flat_entity_dofs`**`(`*entity_dofs*`)`

`firedrake.extrusion_utils.`**`flat_entity_permutations`**`(`*entity_permutations*`)`

`firedrake.extrusion_utils.`**`is_real_tensor_product_element`**`(`*element*`)`

Is the provided FInAT element a tensor product involving the real space?

Parameters
: **element** – A scalar FInAT element.

`firedrake.extrusion_utils.`**`make_extruded_coords`**(*extruded_topology*, *base_coords*, *ext_coords*, *layer_height*, *extrusion_type='uniform'*, *kernel=None*)

Given either a kernel or a (fixed) layer_height, compute an extruded coordinate field for an extruded mesh.

Parameters

- **extruded_topology** – an `ExtrudedMeshTopology` to extrude a coordinate field for.
- **base_coords** – a *Function* to read the base coordinates from.
- **ext_coords** – a *Function* to write the extruded coordinates into.
- **layer_height** – the height for each layer. Either a scalar, where layers will be equi-spaced at the specified height, or a 1D array of variable layer heights to use through the extrusion.
- **extrusion_type** – the type of extrusion to use. Predefined options are either "uniform" (creating equi-spaced layers by extruding in the (n+1)dth direction), "radial" (creating equi-spaced layers by extruding in the outward direction from the origin) or "radial_hedgehog" (creating equi-spaced layers by extruding coordinates in the outward cell-normal direction, needs a P1dgxP1 coordinate field).
- **kernel** – an optional kernel to carry out coordinate extrusion.

The kernel signature (if provided) is:

```
void kernel(double **base_coords, double **ext_coords,
            double *layer_height, int layer)
```

The kernel iterates over the cells of the mesh and receives as arguments the coordinates of the base cell (to read), the coordinates on the extruded cell (to write to), the fixed layer height, and the current cell layer.

4.13 firedrake.formmanipulation module

class `firedrake.formmanipulation.`**`ExtractSubBlock`**

Bases: `MultiFunction`

Extract a sub-block from a form.

class **`IndexInliner`**

Bases: `MultiFunction`

Inline fixed index of list tensors

`expr`(*o*, **ops*)

Reuse object if operands are the same objects.

Use in your own subclass by setting e.g.

```
expr = MultiFunction.reuse_if_untouched
```

as a default rule.

indexed(*o*, *child*, *multiindex*)

multi_index(*o*)

argument(*o*)

coefficient_derivative(*o*, *expr*, *coefficients*, *arguments*, *cds*)

expr(*o*, **ops*)

Reuse object if operands are the same objects.

Use in your own subclass by setting e.g.

```
expr = MultiFunction.reuse_if_untouched
```

as a default rule.

expr_list(*o*, **operands*)

index_inliner = <firedrake.formmanipulation.ExtractSubBlock.IndexInliner object>

multi_index(*o*)

split(*form*, *argument_indices*)

Split a form.

Parameters

- **form** – the form to split.
- **argument_indices** – indices of test and trial spaces to extract. This should be 0-, 1-, or 2-tuple (whose length is the same as the number of arguments as the `form`) whose entries are either an integer index, or else an iterable of indices.

Returns a new `ufl.classes.Form` on the selected subspace.

class firedrake.formmanipulation.**SplitForm**(*indices*, *form*)

Bases: `tuple`

Create new instance of SplitForm(indices, form)

form

Alias for field number 1

indices

Alias for field number 0

`firedrake.formmanipulation.`**`split_form`**`(`*form*, *diagonal=False*`)`

Split a form into a tuple of sub-forms defined on the component spaces.

Each entry is a *`SplitForm`* tuple of the indices into the component arguments and the form defined on that block.

For example, consider the following code:

```
V = FunctionSpace(m, 'CG', 1)
W = V*V*V
u, v, w = TrialFunctions(W)
p, q, r = TestFunctions(W)
a = q*u*dx + p*w*dx
```

Then splitting the form returns a tuple of two forms.

```
((0, 2), w*p*dx),
 (1, 0), q*u*dx))
```

Due to the limited amount of simplification that UFL does, some of the returned forms may eventually evaluate to zero. The form compiler will remove these in its more complex simplification stages.

4.14 firedrake.function module

`class` `firedrake.function.`**`Function`**`(`*function_space*, *val=None*, *name=None*, *dtype=dtype('float64')*, *count=None*`)`

Bases: `Coefficient`, *`FunctionMixin`*

A *`Function`* represents a discretised field over the domain defined by the underlying *`Mesh()`*. Functions are represented as sums of basis functions:

$$f =$$
$$sum_i f_i \phi_i(x)$$

The *`Function`* class provides storage for the coefficients f_i and associates them with a *`FunctionSpace`* object which provides the basis functions
$phi_i(x)$.

Note that the coefficients are always scalars: if the *`Function`* is vector-valued then this is specified in the *`FunctionSpace`*.

Parameters

- **`function_space`** – the *`FunctionSpace`*, or *`MixedFunctionSpace`* on which to build this *`Function`*. Alternatively, another *`Function`* may be passed here and its function space will be used to build this *`Function`*. In this case, the function values are copied.
- **`val`** – NumPy array-like (or `pyop2.Dat`) providing initial values (optional). If val is an existing *`Function`*, then the data will be shared.
- **`name`** – user-defined name for this *`Function`* (optional).
- **`dtype`** – optional data type for this *`Function`* (defaults to `ScalarType`).

- **count** – The `ufl.Coefficient` count which creates the symbolic identity of this *Function*.

assign(*expr*, *subset=None*)

Set the *Function* value to the pointwise value of expr. expr may only contain *Function*s on the same *FunctionSpace* as the *Function* being assigned to.

Similar functionality is available for the augmented assignment operators +=, -=, *= and /=. For example, if *f* and *g* are both Functions on the same *FunctionSpace* then:

```
f += 2 * g
```

will add twice *g* to *f*.

If present, subset must be an `pyop2.Subset` of this *Function*'s `node_set`. The expression will then only be assigned to the nodes on that subset.

Note: Assignment can only be performed for simple weighted sum expressions and constant values. Things like `u.assign(2*v + Constant(3.0))`. For more complicated expressions (e.g. involving the product of functions) *Function.interpolate()* should be used.

at(*arg*, **args*, ***kwargs*)

Evaluate function at points.

Parameters

- **arg** – The point to locate.
- **args** – Additional points.
- **dont_raise** – Do not raise an error if a point is not found.
- **tolerance** – Tolerance to use when checking for points in cell.

copy(*deepcopy=False*)

Return a copy of this Function.

Parameters

deepcopy – If `True`, the new *Function* will allocate new space and copy values. If `False`, the default, then the new *Function* will share the dof values.

evaluate(*coord*, *mapping*, *component*, *index_values*)

Get *self* from *mapping* and return the component asked for.

function_space()

Return the *FunctionSpace*, or *MixedFunctionSpace* on which this *Function* is defined.

interpolate(*expression*, *subset=None*, *ad_block_tag=None*)

Interpolate an expression onto this *Function*.

Parameters

- **expression** – a UFL expression to interpolate

- **ad_block_tag** – string for tagging the resulting block on the Pyadjoint tape

Returns
this *Function* object

project(*b*, **args*, ***kwargs*)

Project `b` onto `self`. `b` must be a *Function* or a UFL expression.

This is equivalent to `project(b, self)`. Any of the additional arguments to *project()* may also be passed, and they will have their usual effect.

split()

Extract any sub *Function*s defined on the component spaces of this this *Function*'s *FunctionSpace*.

sub(*i*)

Extract the ith sub *Function* of this *Function*.

Parameters
i – the index to extract

See also *split()*.

If the *Function* is defined on a `VectorFunctionSpace` or `TensorFunctiionSpace` this returns a proxy object indexing the ith component of the space, suitable for use in boundary condition application.

property topological

The underlying coordinateless function.

vector()

Return a *Vector* wrapping the data in this *Function*

exception `firedrake.function.`**PointNotInDomainError**(*domain*, *point*)

Bases: `Exception`

Raised when attempting to evaluate a function outside its domain, and no fill value was given.

Attributes: domain, point

4.15 firedrake.functionspace module

This module implements the user-visible API for constructing *FunctionSpace* and *MixedFunctionSpace* objects. The API is functional, rather than object-based, to allow for simple backwards-compatibility, argument checking, and dispatch.

`firedrake.functionspace.`**FunctionSpace**(*mesh*, *family*, *degree=None*, *name=None*, *vfamily=None*, *vdegree=None*)

Create a *FunctionSpace*.

Parameters

- **mesh** – The mesh to determine the cell from.
- **family** – The finite element family.

- **degree** – The degree of the finite element.
- **name** – An optional name for the function space.
- **vfamily** – The finite element in the vertical dimension (extruded meshes only).
- **vdegree** – The degree of the element in the vertical dimension (extruded meshes only).

The `family` argument may be an existing `ufl.FiniteElementBase`, in which case all other arguments are ignored and the appropriate *FunctionSpace* is returned.

`firedrake.functionspace.`**`MixedFunctionSpace`**`(`*spaces*, *name=None*, *mesh=None*`)`

Create a *MixedFunctionSpace*.

Parameters

- **spaces** – An iterable of constituent spaces, or a `MixedElement`.
- **name** – An optional name for the mixed function space.
- **mesh** – An optional mesh. Must be provided if spaces is a `MixedElement`, ignored otherwise.

`firedrake.functionspace.`**`TensorFunctionSpace`**`(`*mesh*, *family*, *degree=None*, *shape=None*, *symmetry=None*, *name=None*, *vfamily=None*, *vdegree=None*`)`

Create a rank-2 *FunctionSpace*.

Parameters

- **mesh** – The mesh to determine the cell from.
- **family** – The finite element family.
- **degree** – The degree of the finite element.
- **shape** – An optional shape for the tensor-valued degrees of freedom at each function space node (defaults to a square tensor using the geometric dimension of the mesh).
- **symmetry** – Optional symmetries in the tensor value.
- **name** – An optional name for the function space.
- **vfamily** – The finite element in the vertical dimension (extruded meshes only).
- **vdegree** – The degree of the element in the vertical dimension (extruded meshes only).

The `family` argument may be an existing `FiniteElementBase`, in which case all other arguments are ignored and the appropriate *FunctionSpace* is returned. In this case, the provided element must have an empty `value_shape()`.

Note: The element that you provide must be a scalar element (with empty `value_shape`). If you already have an existing `TensorElement`, you should pass it to *FunctionSpace()*

directly instead.

firedrake.functionspace.**VectorFunctionSpace**(*mesh*, *family*, *degree=None*, *dim=None*, *name=None*, *vfamily=None*, *vdegree=None*)

Create a rank-1 *FunctionSpace*.

Parameters

- **mesh** – The mesh to determine the cell from.
- **family** – The finite element family.
- **degree** – The degree of the finite element.
- **dim** – An optional number of degrees of freedom per function space node (defaults to the geometric dimension of the mesh).
- **name** – An optional name for the function space.
- **vfamily** – The finite element in the vertical dimension (extruded meshes only).
- **vdegree** – The degree of the element in the vertical dimension (extruded meshes only).

The `family` argument may be an existing `ufl.FiniteElementBase`, in which case all other arguments are ignored and the appropriate *FunctionSpace* is returned. In this case, the provided element must have an empty `ufl.FiniteElementBase.value_shape()`.

Note: The element that you provide need be a scalar element (with empty `value_shape`), however, it should not be an existing `VectorElement`. If you already have an existing `VectorElement`, you should pass it to *FunctionSpace()* directly instead.

4.16 firedrake.functionspacedata module

This module provides an object that encapsulates data that can be shared between different *FunctionSpace* objects.

The sharing is based on the idea of compatibility of function space node layout. The shared data is stored on the *Mesh()* the function space is created on, since the created objects are mesh-specific. The sharing is done on an individual key basis. So, for example, Sets can be shared between all function spaces with the same number of nodes per topological entity. However, maps are specific to the node *ordering*.

This means, for example, that function spaces with the same *node* ordering, but different numbers of dofs per node (e.g. FiniteElement vs VectorElement) can share the PyOP2 Set and Map data.

firedrake.functionspacedata.**get_shared_data**(*mesh*, *ufl_element*)

Return the `FunctionSpaceData` for the given element.

Parameters

- **mesh** – The mesh to build the function space data on.
- **ufl_element** – A UFL element.

Raises
`ValueError` – if mesh or ufl_element are invalid.

Returns
a `FunctionSpaceData` object with the shared data.

4.17 firedrake.functionspaceimpl module

This module provides the implementations of *FunctionSpace* and *MixedFunctionSpace* objects, along with some utility classes for attaching extra information to instances of these.

firedrake.functionspaceimpl.**ComponentFunctionSpace**(*parent*, *component*)

Build a new FunctionSpace that remembers it represents a particular component. Used for applying boundary conditions to components of a *VectorFunctionSpace()* or *TensorFunctionSpace()*.

Parameters

- **parent** – The parent space (a FunctionSpace with a VectorElement or TensorElement).
- **component** – The component to represent.

Returns
A new *ProxyFunctionSpace* with the component set.

class firedrake.functionspaceimpl.**FunctionSpace**(*mesh*, *element*, *name=None*)

Bases: `object`

A representation of a function space.

A *FunctionSpace* associates degrees of freedom with topological mesh entities. The degree of freedom mapping is determined from the provided element.

Parameters

- **mesh** – The *Mesh()* to build the function space on.
- **element** – The `FiniteElementBase` describing the degrees of freedom.
- **name** – An optional name for this *FunctionSpace*, useful for later identification.

The element can be a essentially any `FiniteElementBase`, except for a `MixedElement`, for which one should use the *MixedFunctionSpace* constructor.

To determine whether the space is scalar-, vector- or tensor-valued, one should inspect the *rank* of the resulting object. Note that function spaces created on *intrinsically* vector-valued finite elements (such as the Raviart-Thomas space) have `rank` 0.

Warning: Users should not build a *FunctionSpace* directly, instead they should use the utility *FunctionSpace()* function, which provides extra error checking and argument sanitising.

boundary_nodes(*sub_domain*)

Return the boundary nodes for this *FunctionSpace*.

Parameters

sub_domain – the mesh marker selecting which subset of facets to consider.

Returns

A numpy array of the unique function space nodes on the selected portion of the boundary.

See also *DirichletBC* for details of the arguments.

cell_node_list

A numpy array mapping mesh cells to function space nodes.

cell_node_map()

Return the `pyop2.Map` from cels to function space nodes.

collapse()

component = None

The component of this space in its parent VectorElement space, or `None`.

dim()

The global number of degrees of freedom for this function space.

See also *dof_count* and *node_count*.

dm

A PETSc DM describing the data layout for this FunctionSpace.

dof_count

The number of degrees of freedom (includes halo dofs) of this function space on this process. Cf. *node_count*.

dof_dset

A `pyop2.DataSet` representing the function space degrees of freedom.

exterior_facet_node_map()

Return the `pyop2.Map` from exterior facets to function space nodes.

index = None

The position of this space in its parent *MixedFunctionSpace*, or `None`.

interior_facet_node_map()

Return the `pyop2.Map` from interior facets to function space nodes.

local_to_global_map(*bcs*, *lgmap=None*)

Return a map from process local dof numbering to global dof numbering.

If BCs is provided, mask out those dofs which match the BC nodes.

make_dat(*val=None*, *valuetype=None*, *name=None*)

Return a newly allocated `pyop2.Dat` defined on the *dof_dset* of this *Function*.

mesh()

name
: The (optional) descriptive name for this space.

node_count
: The number of nodes (includes halo nodes) of this function space on this process. If the *FunctionSpace* has *rank* 0, this is equal to the *dof_count*, otherwise the *dof_count* is *dim* times the *node_count*.

node_set
: A `pyop2.Set` representing the function space nodes.

parent = None
: The parent space if this space was extracted from one, or `None`.

rank
: The rank of this *FunctionSpace*. Spaces where the element is scalar-valued (or intrinsically vector-valued) have rank zero. Spaces built on `VectorElement` or `TensorElement` instances have rank equivalent to the number of components of their `value_shape()`.

split()
: Split into a tuple of constituent spaces.

sub(*i*)
: Return a view into the ith component.

topological
: Function space on a mesh topology.

ufl_element()
: The `FiniteElementBase` associated with this space.

ufl_function_space()
: The `FunctionSpace` associated with this space.

value_size
: The total number of degrees of freedom at each function space node.

class firedrake.functionspaceimpl.**FunctionSpaceCargo**(*topological:* FunctionSpace, *parent:* WithGeometry | *None*)

Bases: `object`

Helper class carrying data for a *WithGeometry*.

It is required because it permits Firedrake to have stripped forms that still know Firedrake-specific information (e.g. that they are a component of a parent function space).

parent: ***WithGeometry*** **| None**

topological: ***FunctionSpace***

firedrake.functionspaceimpl.**IndexedFunctionSpace**(*index*, *space*, *parent*)

Build a new FunctionSpace that remembers it is a particular subspace of a *MixedFunctionSpace*.

Parameters

- **index** – The index into the parent space.
- **space** – The subspace to represent
- **parent** – The parent mixed space.

Returns
A new *ProxyFunctionSpace* with index and parent set.

class firedrake.functionspaceimpl.**MixedFunctionSpace**(*spaces*, *name=None*)

Bases: `object`

A function space on a mixed finite element.

This is essentially just a bag of individual *FunctionSpace* objects.

Parameters

- **spaces** – The constituent spaces.
- **name** – An optional name for the mixed space.

Warning: Users should not build a *MixedFunctionSpace* directly, but should instead use the functional interface provided by *MixedFunctionSpace()*.

cell_node_map()

A `pyop2.MixedMap` from the `Mesh.cell_set` of the underlying mesh to the *node_set* of this *MixedFunctionSpace*. This is composed of the *FunctionSpace.cell_node_map*s of the underlying *FunctionSpace*s of which this *MixedFunctionSpace* is composed.

component = None

dim()

The global number of degrees of freedom for this function space.

See also *dof_count* and *node_count*.

dm

A PETSc DM describing the data layout for fieldsplit solvers.

dof_count

Return a tuple of *FunctionSpace.dof_count*s of the *FunctionSpace*s of which this *MixedFunctionSpace* is composed.

dof_dset

A `pyop2.MixedDataSet` containing the degrees of freedom of this *MixedFunctionSpace*. This is composed of the *FunctionSpace.dof_dset*s of the underlying *FunctionSpace*s of which this *MixedFunctionSpace* is composed.

exterior_facet_node_map()

Return the `pyop2.Map` from exterior facets to function space nodes.

index = None

interior_facet_node_map()

Return the pyop2.MixedMap from interior facets to function space nodes.

local_to_global_map(*bcs*)

Return a map from process local dof numbering to global dof numbering.

If BCs is provided, mask out those dofs which match the BC nodes.

make_dat(*val=None*, *valuetype=None*, *name=None*)

Return a newly allocated pyop2.MixedDat defined on the *dof_dset* of this *MixedFunctionSpace*.

mesh()

node_count

Return a tuple of *FunctionSpace.node_count*s of the *FunctionSpace*s of which this *MixedFunctionSpace* is composed.

node_set

A pyop2.MixedSet containing the nodes of this *MixedFunctionSpace*. This is composed of the *FunctionSpace.node_set*s of the underlying *FunctionSpace*s this *MixedFunctionSpace* is composed of one or (for VectorFunctionSpaces) more degrees of freedom are stored at each node.

num_sub_spaces()

Return the number of *FunctionSpace*s of which this *MixedFunctionSpace* is composed.

parent = None

rank = 1

split()

The list of *FunctionSpace*s of which this *MixedFunctionSpace* is composed.

sub(*i*)

Return the *i`th :class:`FunctionSpace* in this *MixedFunctionSpace*.

property topological

Function space on a mesh topology.

ufl_element()

The MixedElement associated with this space.

ufl_function_space()

The FunctionSpace associated with this space.

value_size

Return the sum of the *FunctionSpace.value_size*s of the *FunctionSpace*s this *MixedFunctionSpace* is composed of.

class firedrake.functionspaceimpl.**ProxyFunctionSpace**(*mesh*, *element*, *name=None*)

Bases: *FunctionSpace*

A *FunctionSpace* that one can attach extra properties to.

Parameters

- **mesh** – The mesh to use.
- **element** – The UFL element.
- **name** – The name of the function space.

Warning: Users should not build a *ProxyFunctionSpace* directly, it is mostly used as an internal implementation detail.

identifier = None

An optional identifier, for debugging purposes.

make_dat(**args*, ***kwargs*)

Create a `pyop2.Dat`.

Raises

ValueError – if *no_dats* is `True`.

no_dats = False

Can this proxy make `pyop2.Dat` objects

class firedrake.functionspaceimpl.**RealFunctionSpace**(*mesh*, *element*, *name*)

Bases: *FunctionSpace*

FunctionSpace based on elements of family “Real”. A :class`RealFunctionSpace` only has a single global value for the whole mesh.

This class should not be directly instantiated by users. Instead, FunctionSpace objects will transform themselves into *RealFunctionSpace* objects as appropriate.

bottom_nodes()

RealFunctionSpace objects have no bottom nodes.

cell_node_map(*bcs=None*)

RealFunctionSpace objects have no cell node map.

dim()

The global number of degrees of freedom for this function space.

See also `dof_count` and `node_count`.

exterior_facet_node_map(*bcs=None*)

RealFunctionSpace objects have no exterior facet node map.

finat_element = None

interior_facet_node_map(*bcs=None*)

RealFunctionSpace objects have no interior facet node map.

local_to_global_map(*bcs*, *lgmap=None*)

Return a map from process local dof numbering to global dof numbering.

If BCs is provided, mask out those dofs which match the BC nodes.

make_dat(*val=None*, *valuetype=None*, *name=None*)

Return a newly allocated `pyop2.Global` representing the data for a *Function* on this space.

rank = 0

The rank of this *FunctionSpace*. Spaces where the element is scalar-valued (or intrinsically vector-valued) have rank zero. Spaces built on `VectorElement` or `TensorElement` instances have rank equivalent to the number of components of their `value_shape()`.

shape = ()

top_nodes()

RealFunctionSpace objects have no bottom nodes.

value_size = 1

The total number of degrees of freedom at each function space node.

class firedrake.functionspaceimpl.**WithGeometry**(*mesh*, *element*, *component=None*, *cargo=None*)

Bases: `FunctionSpace`

Attach geometric information to a *FunctionSpace*.

Function spaces on meshes with different geometry but the same topology can share data, except for their UFL cell. This class facilitates that.

Users should not instantiate a *WithGeometry* object explicitly except in a small number of cases.

When instantiating a *WithGeometry*, users should call *create()* rather than `__init__()`.

Parameters

- **mesh** – The mesh with geometric information to use.
- **element** – The UFL element.
- **component** – The component of this space in a parent vector element space, or `None`.
- **cargo** – *FunctionSpaceCargo* instance carrying Firedrake-specific data that is not required for code generation.

boundary_nodes(*sub_domain*)

Return the boundary nodes for this *WithGeometry*.

Parameters

sub_domain – the mesh marker selecting which subset of facets to consider.

Returns

A numpy array of the unique function space nodes on the selected portion of the boundary.

See also *DirichletBC* for details of the arguments.

collapse()

classmethod create(*function_space*, *mesh*)

Create a *WithGeometry*.

Parameters

- **function_space** – The topological function space to attach geometry to.
- **mesh** – The mesh with geometric information to use.

dm

get_work_function(*zero=True*)

Get a temporary work *Function* on this *FunctionSpace*.

Parameters

zero – Should the *Function* be guaranteed zero? If `zero` is `False` the returned function may or may not be zeroed, and the user is responsible for appropriate zeroing.

Raises

ValueError – if *max_work_functions* are already checked out.

Note: This method is intended to be used for short-lived work functions, if you actually need a function for general usage use the *Function* constructor.

When you are finished with the work function, you should restore it to the pool of available functions with *restore_work_function()*.

property max_work_functions

The maximum number of work functions this *FunctionSpace* supports.

See *get_work_function()* for obtaining work functions.

mesh()

Return ufl domain.

property num_work_functions

The number of checked out work functions.

property parent

restore_work_function(*function*)

Restore a work function obtained with *get_work_function()*.

Parameters

function – The work function to restore

Raises

ValueError – if the provided function was not obtained with *get_work_function()* or it has already been restored.

Warning: This does *not* invalidate the name in the calling scope, it is the user's responsibility not to use a work function after restoring it.

split()
Split into a tuple of constituent spaces.

sub(*i*)

property topological

ufl_cell()
The `Cell` this FunctionSpace is defined on.

ufl_function_space()
The `FunctionSpace` this object represents.

4.18 firedrake.halo module

class firedrake.halo.**Halo**(*dm*, *section*, *comm*)
Bases: `Halo`

Build a Halo for a function space.

Parameters

- **dm** – The DM describing the topology.
- **section** – The data layout.

The halo is implemented using a PETSc SF (star forest) object and is usable as a PyOP2 `pyop2.Halo`.

comm

global_to_local_begin(*dat*, *insert_mode*)
Begin an exchange from global (assembled) to local (ghosted) representation.

Parameters

- **dat** – The `Dat` to exchange.
- **insert_mode** – The insertion mode.

global_to_local_end(*dat*, *insert_mode*)
Finish an exchange from global (assembled) to local (ghosted) representation.

Parameters

- **dat** – The `Dat` to exchange.
- **insert_mode** – The insertion mode.

local_to_global_begin(*dat*, *insert_mode*)
Begin an exchange from local (ghosted) to global (assembled) representation.

Parameters

- **dat** – The `Dat` to exchange.
- **insert_mode** – The insertion mode.

local_to_global_end(*dat*, *insert_mode*)

Finish an exchange from local (ghosted) to global (assembled) representation.

Parameters

- **dat** – The `Dat` to exchange.
- **insert_mode** – The insertion mode.

local_to_global_numbering

sf

`firedrake.halo.`**reduction_op**(*op*, *invec*, *inoutvec*, *datatype*)

4.19 firedrake.interpolation module

class `firedrake.interpolation.`**Interpolator**(*expr*, *V*, *subset=None*, *freeze_expr=False*, *access=Access.WRITE*)

Bases: `object`

A reusable interpolation object.

Parameters

- **expr** – The expression to interpolate.
- **V** – The *FunctionSpace* or *Function* to interpolate into.
- **subset** – An optional `pyop2.Subset` to apply the interpolation over.
- **freeze_expr** – Set to True to prevent the expression being re-evaluated on each call.

This object can be used to carry out the same interpolation multiple times (for example in a timestepping loop).

Note: The *Interpolator* holds a reference to the provided arguments (such that they won't be collected until the *Interpolator* is also collected).

interpolate(**function*, *output=None*, *transpose=False*)

Compute the interpolation.

Parameters

- **function** – If the expression being interpolated contains an `ufl.Argument`, then the *Function* value to interpolate.
- **output** – Optional. A *Function* to contain the output.
- **transpose** – Set to true to apply the transpose (adjoint) of the interpolation operator.

Returns

The resulting interpolated *Function*.

firedrake.interpolation.**interpolate**(*expr*, *V*, *subset=None*, *access=Access.WRITE*, *ad_block_tag=None*)

Interpolate an expression onto a new function in V.

Parameters

- **expr** – a UFL expression.
- **V** – the *FunctionSpace* to interpolate into (or else an existing *Function*).
- **subset** – An optional pyop2.Subset to apply the interpolation over.
- **access** – The access descriptor for combining updates to shared dofs.
- **ad_block_tag** – string for tagging the resulting block on the Pyadjoint tape

Returns

a new *Function* in the space V (or V if it was a Function).

Note: If you use an access descriptor other than WRITE, the behaviour of interpolation is changes if interpolating into a function space, or an existing function. If the former, then the newly allocated function will be initialised with appropriate values (e.g. for MIN access, it will be initialised with MAX_FLOAT). On the other hand, if you provide a function, then it is assumed that its values should take part in the reduction (hence using MIN will compute the MIN between the existing values and any new values).

Note: If you find interpolating the same expression again and again (for example in a time loop) you may find you get better performance by using an *Interpolator* instead.

4.20 firedrake.linear_solver module

class firedrake.linear_solver.**LinearSolver**(*A*, ***, *P=None*, *solver_parameters=None*, *nullspace=None*, *transpose_nullspace=None*, *near_nullspace=None*, *options_prefix=None*)

Bases: *OptionsManager*

A linear solver for assembled systems (Ax = b).

Parameters

- **A** – a *MatrixBase* (the operator).
- **P** – an optional *MatrixBase* to construct any preconditioner from; if none is supplied A is used to construct the preconditioner.
- **parameters** – (optional) dict of solver parameters.
- **nullspace** – an optional *VectorSpaceBasis* (or *MixedVectorSpaceBasis* spanning the null space of the operator.

- **transpose_nullspace** – as for the nullspace, but used to make the right hand side consistent.
- **near_nullspace** – as for the nullspace, but used to set the near null-pace.
- **options_prefix** – an optional prefix used to distinguish PETSc options. If not provided a unique prefix will be created. Use this option if you want to pass options to the solver from the command line in addition to through the `solver_parameters` dict.

Note: Any boundary conditions for this solve *must* have been applied when assembling the operator.

DEFAULT_KSP_PARAMETERS = {'ksp_rtol': 1e-07, 'ksp_type': 'preonly', 'mat_mumps_icntl_14': 200, 'mat_type': 'aij', 'pc_factor_mat_solver_type': 'mumps', 'pc_type': 'lu'}

solve(*x*, *b*)

test_space

trial_space

4.21 firedrake.logging module

firedrake.logging.**critical**(*msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'CRITICAL'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.critical("Houston, we have a %s", "major disaster", exc_info=1)

firedrake.logging.**debug**(*msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'DEBUG'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.debug("Houston, we have a %s", "thorny problem", exc_info=1)

firedrake.logging.**error**(*msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'ERROR'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.error("Houston, we have a %s", "major problem", exc_info=1)

firedrake.logging.**info**(*msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'INFO'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.info("Houston, we have a %s", "interesting problem", exc_info=1) `firedrake.logging.`**`info_blue`**`(`*message*, **args*, ***kwargs*`)`

Write info message in blue.

Parameters
`message` – the message to be printed.

`firedrake.logging.`**`info_green`**`(`*message*, **args*, ***kwargs*`)`

Write info message in green.

Parameters
`message` – the message to be printed.

`firedrake.logging.`**`info_red`**`(`*message*, **args*, ***kwargs*`)`

Write info message in red.

Parameters
`message` – the message to be printed.

`firedrake.logging.`**`log`**`(`*level*, *msg*, **args*, ***kwargs*`)`

Log ‘msg % args’ with the integer severity ‘level’.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.log(level, “We have a %s”, “mysterious problem”, exc_info=1)

`firedrake.logging.`**`set_level`**`(`*level*`)`

Set the log level for Firedrake components.

Parameters
`level` – The level to use.

This controls what level of logging messages are printed to stderr. The higher the level, the fewer the number of messages.

`firedrake.logging.`**`set_log_handlers`**`(`*handlers=None*, *comm=<mpi4py.MPI.Intracomm object>*`)`

Set handlers for the log messages of the different Firedrake components.

Parameters

- **`handlers`** – Optional dict of handlers keyed by the name of the logger. If not provided, a separate `logging.StreamHandler` will be created for each logger.
- **`comm`** – The communicator the handler should be collective over. If provided, only rank-0 on that communicator will write to the handler, other ranks will use a `logging.NullHandler`. If set to `None`, all ranks will use the provided handler. This could be used, for example, if you want to log to one file per rank.

`firedrake.logging.`**`set_log_level`**`(`*level*`)`

Set the log level for Firedrake components.

Parameters
`level` – The level to use.

This controls what level of logging messages are printed to stderr. The higher the level, the fewer the number of messages.

firedrake.logging.**warning**(*msg*, **args*, ***kwargs*)

Log 'msg % args' with severity 'WARNING'.

To pass exception information, use the keyword argument exc_info with a true value, e.g.

logger.warning("Houston, we have a %s", "bit of a problem", exc_info=1)

4.22 firedrake.matrix module

class firedrake.matrix.**ImplicitMatrix**(*a*, *bcs*, **args*, ***kwargs*)

Bases: *MatrixBase*

A representation of the action of bilinear form operating without explicitly assembling the associated matrix. This class wraps the relevant information for Python PETSc matrix.

Parameters

- **a** – the bilinear form this *Matrix* represents.
- **bcs** – an iterable of boundary conditions to apply to this *Matrix*. May be *None* if there are no boundary conditions to apply.

Note: This object acts to the right on an assembled *Function* and to the left on an assembled cofunction (currently represented by a *Function*).

---

assemble()

class firedrake.matrix.**Matrix**(*a*, *bcs*, *mat_type*, **args*, ***kwargs*)

Bases: *MatrixBase*

A representation of an assembled bilinear form.

Parameters

- **a** – the bilinear form this *Matrix* represents.
- **bcs** – an iterable of boundary conditions to apply to this *Matrix*. May be *None* if there are no boundary conditions to apply.
- **mat_type** – matrix type of assembled matrix.

A pyop2.Mat will be built from the remaining arguments, for valid values, see pyop2.Mat.

Note: This object acts to the right on an assembled *Function* and to the left on an assembled cofunction (currently represented by a *Function*).

---

assemble()

class firedrake.matrix.**MatrixBase**(*a*, *bcs*, *mat_type*)

Bases: object

A representation of the linear operator associated with a bilinear form and bcs. Explicitly assembled matrices and matrix-free matrix classes will derive from this

Parameters

- **a** – the bilinear form this *MatrixBase* represents.
- **bcs** – an iterable of boundary conditions to apply to this *MatrixBase*. May be *None* if there are no boundary conditions to apply.
- **mat_type** – matrix type of assembled matrix, or 'matfree' for matrix-free

property bcs

The set of boundary conditions attached to this *MatrixBase* (may be empty).

property has_bcs

Return True if this *MatrixBase* has any boundary conditions attached to it.

mat_type

Matrix type.

Matrix type used in the assembly of the PETSc matrix: 'aij', 'baij', 'dense' or 'nest', or 'matfree' for matrix-free.

4.23 firedrake.mesh module

firedrake.mesh.**DEFAULT_MESH_NAME = 'firedrake_default'**

The default name of the mesh.

class firedrake.mesh.**DistributedMeshOverlapType**(*value*, *names=None*, ***, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*)

Bases: Enum

How should the mesh overlap be grown for distributed meshes?

Possible options are:

- ***NONE*: Don't overlap distributed meshes, only useful for problems with**
  no interior facet integrals.
- ***FACET*: Add ghost entities in the closure of the star of**
  facets.
- ***VERTEX*: Add ghost entities in the closure of the star**
  of vertices.

Defaults to *FACET*.

FACET = 2

NONE = 1

VERTEX = 3

firedrake.mesh.**ExtrudedMesh**(*mesh*, *layers*, *layer_height=None*, *extrusion_type='uniform'*, *kernel=None*, *gdim=None*, *name=None*)

Build an extruded mesh from an input mesh

Parameters

- **mesh** – the unstructured base mesh
- **layers** – number of extruded cell layers in the "vertical" direction. One may also pass an array of shape (cells, 2) to specify a variable number of layers. In this case, each entry is a pair `[a, b]` where `a` indicates the starting cell layer of the column and `b` the number of cell layers in that column.
- **layer_height** – the layer height. A scalar value will result in evenly-spaced layers, whereas an array of values will vary the layer height through the extrusion. If this is omitted, the value defaults to 1/layers (i.e. the extruded mesh has total height 1.0) unless a custom kernel is used. Must be provided if using a variable number of layers.
- **extrusion_type** – the algorithm to employ to calculate the extruded coordinates. One of "uniform", "radial", "radial_hedgehog" or "custom". See below.
- **kernel** – a `pyop2.Kernel` to produce coordinates for the extruded mesh. See *make_extruded_coords()* for more details.
- **gdim** – number of spatial dimensions of the resulting mesh (this is only used if a custom kernel is provided)
- **name** – optional name for the extruded mesh.

The various values of `extrusion_type` have the following meanings:

"uniform"
the extruded mesh has an extra spatial dimension compared to the base mesh. The layers exist in this dimension only.

"radial"
the extruded mesh has the same number of spatial dimensions as the base mesh; the cells are radially extruded outwards from the origin. This requires the base mesh to have topological dimension strictly smaller than geometric dimension.

"radial_hedgehog"
similar to *radial*, but the cells are extruded in the direction of the outward-pointing cell normal (this produces a P1dgxP1 coordinate field). In this case, a radially extruded coordinate field (generated with `extrusion_type="radial"`) is available in the `radial_coordinates` attribute.

"custom"
use a custom kernel to generate the extruded coordinates

For more details see the *manual section on extruded meshes*.

`firedrake.mesh.`**`Mesh`**`(`*meshfile*, ***kwargs*`)`

Construct a mesh object.

Meshes may either be created by reading from a mesh file, or by providing a PETSc DMPlex object defining the mesh topology.

Parameters

- **meshfile** – Mesh file name (or DMPlex object) defining mesh topology. See below for details on supported mesh formats.
- **name** – optional name of the mesh object.

- **dim** – optional specification of the geometric dimension of the mesh (ignored if not reading from mesh file). If not supplied the geometric dimension is deduced from the topological dimension of entities in the mesh.
- **reorder** – optional flag indicating whether to reorder meshes for better cache locality. If not supplied the default value in `parameters["reorder_meshes"]` is used.
- **distribution_parameters** – an optional dictionary of options for parallel mesh distribution. Supported keys are:
  - **"partition": which may take the value `None` (use** the default choice), `False` (do not) `True` (do), or a 2-tuple that specifies a partitioning of the cells (only really useful for debugging).
  - **"partitioner_type": which may take "chaco",** `"ptscotch"`, `"parmetis"`, or `"shell"`.
  - **"overlap_type": a 2-tuple indicating how to grow** the mesh overlap. The first entry should be a *`DistributedMeshOverlapType`* instance, the second the number of levels of overlap.
- **distribution_name** – the name of parallel distribution used when checkpointing; if not given, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if not given, the name is automatically generated.
- **comm** – the communicator to use when creating the mesh. If not supplied, then the mesh will be created on COMM_WORLD. If `meshfile` is a DMPlex object then must be indentical to or congruent with the DMPlex communicator.

When the mesh is read from a file the following mesh formats are supported (determined, case insensitively, from the filename extension):

- GMSH: with extension *.msh*
- Exodus: with extension *.e*, *.exo*
- CGNS: with extension *.cgns*
- Triangle: with extension *.node*
- HDF5: with extension *.h5*, *.hdf5* (Can only load HDF5 files created by `MeshGeometry.save()` method.)

Note: When the mesh is created directly from a DMPlex object, the `dim` parameter is ignored (the DMPlex already knows its geometric and topological dimensions).

`firedrake.mesh.`**`SubDomainData`**`(`*geometric_expr*`)`

Creates a subdomain data object from a boolean-valued UFL expression.

The result can be attached as the subdomain_data field of a `ufl.Measure`. For example:

```
x = mesh.coordinates
sd = SubDomainData(x[0] < 0.5)
assemble(f*dx(subdomain_data=sd))
```

firedrake.mesh.**VertexOnlyMesh**(*mesh*, *vertexcoords*, *missing_points_behaviour=None*, *tolerance=None*)

Create a vertex only mesh, immersed in a given mesh, with vertices defined by a list of coordinates.

Parameters

- **mesh** – The unstructured mesh in which to immerse the vertex only mesh.
- **vertexcoords** – A list of coordinate tuples which defines the vertices.
- **missing_points_behaviour** – optional string argument for what to do when vertices which are outside of the mesh are discarded. If `'warn'`, will print a warning. If `'error'` will raise a ValueError. Note that setting this will cause all MPI ranks to check that they have the same list of vertices (else the test is not possible): this operation scales with number of vertices and number of ranks.
- **tolerance** – the amount by which the local coordinates of a point are allowed to fall outside the cell while still having the point count as in the cell. Increase the default (1.0e-14) somewhat if vertices interior to the domain are being lost in the *VertexOnlyMesh* construction process.

Note: The vertex only mesh uses the same communicator as the input `mesh`.

Note: Extruded and immersed manifold meshes are not yet supported.

Note: Modifying the coordinates of the parent mesh is not currently supported. Doing so will cause interpolation to Functions defined on the VertexOnlyMesh to return the wrong values.

Note: When running in parallel, `vertexcoords` are strictly confined to the local `mesh` cells of that rank. This means that if rank A has `vertexcoords` {X} that are not found in the mesh cells owned by rank A but are found in the mesh cells owned by rank B, **and rank B has not been supplied with those** `vertexcoords`, then the `vertexcoords` {X} will be lost.

This can be avoided by either

1. making sure that all ranks are supplied with the same `vertexcoords` or by
2. ensuring that `vertexcoords` are already found in cells owned by the `mesh` partition of the given rank.

For more see this github issue.

`firedrake.mesh.unmarked = -1`

A mesh marker that selects all entities that are not explicitly marked.

4.24 firedrake.norms module

`firedrake.norms.errornorm`(*u*, *uh*, *norm_type='L2'*, *degree_rise=None*, *mesh=None*)

Compute the error $e = u - u_h$ in the specified norm.

Parameters

- **u** – a *Function* or UFL expression containing an "exact" solution
- **uh** – a *Function* containing the approximate solution
- **norm_type** – the type of norm to compute, see *norm()* for details of supported norm types.
- **degree_rise** – ignored.
- **mesh** – an optional mesh on which to compute the error norm (currently ignored).

`firedrake.norms.norm`(*v*, *norm_type='L2'*, *mesh=None*)

Compute the norm of `v`.

Parameters

- **v** – a ufl expression (`Expr`) to compute the norm of
- **norm_type** – the type of norm to compute, see below for options.
- **mesh** – an optional mesh on which to compute the norm (currently ignored).

Available norm types are:

- Lp $||v||_{L^p} = (\int |v|^p)^{\frac{1}{p}} \mathrm{d}x$
- H1 $||v||^2_{H^1} = \int (v, v) + (\nabla v, \nabla v) \mathrm{d}x$
- Hdiv $||v||^2_{H_{\mathrm{div}}} = \int (v, v) + (\nabla \cdot v, \nabla \cdot v) \mathrm{d}x$
- Hcurl $||v||^2_{H_{\mathrm{curl}}} = \int (v, v) + (\nabla \wedge v, \nabla \wedge v) \mathrm{d}x$

4.25 firedrake.nullspace module

`class firedrake.nullspace.MixedVectorSpaceBasis`(*function_space*, *bases*)

Bases: `object`

A basis for a mixed vector space

Parameters

- **function_space** – the *MixedFunctionSpace* this vector space is a basis for.
- **bases** – an iterable of bases for the null spaces of the subspaces in the mixed space.

You can use this to express the null space of a singular operator on a mixed space. The bases you supply will be used to set null spaces for each of the diagonal blocks in the operator. If you only care about the null space on one of the blocks, you can pass an indexed function space as a placeholder in the positions you don't care about.

For example, consider a mixed poisson discretisation with pure Neumann boundary conditions:

```
V = FunctionSpace(mesh, "BDM", 1)
Q = FunctionSpace(mesh, "DG", 0)

W = V*Q

sigma, u = TrialFunctions(W)
tau, v = TestFunctions(W)

a = (inner(sigma, tau) + div(sigma)*v + div(tau)*u)*dx
```

The null space of this operator is a constant function in `Q`. If we solve the problem with a Schur complement, we only care about projecting the null space out of the `QxQ` block. We can do this like so

```
nullspace = MixedVectorSpaceBasis(W, [W[0],␣
↪VectorSpaceBasis(constant=True)])
solve(a == ..., nullspace=nullspace)
```

class firedrake.nullspace.**VectorSpaceBasis**(*vecs=None*, *constant=False*, *comm=None*)

Bases: `object`

Build a basis for a vector space.

You can use this basis to express the null space of a singular operator.

Parameters

- **vecs** – a list of *Vector*s or `Function`s spanning the space.
- **constant** – does the null space include the constant vector? If you pass `constant=True` you should not also include the constant vector in the list of `vecs` you supply.
- **comm** – Communicator to create the nullspace on.

Note: Before using this object in a solver, you must ensure that the basis is orthonormal. You can do this by calling *orthonormalize()*, this modifies the provided vectors *in place*.

Warning: The vectors you pass in to this object are *not* copied. You should therefore not modify them after instantiation since the basis will then be incorrect.

check_orthogonality(*orthonormal=True*)

Check if the basis is orthogonal.

Parameters
`orthonormal` – If True check that the basis is also orthonormal.

Raises
`ValueError` – If the basis is not orthogonal/orthonormal.

is_orthogonal()

Is this vector space basis orthogonal?

is_orthonormal()

Is this vector space basis orthonormal?

nullspace(*comm=None*)

The PETSc NullSpace object for this *VectorSpaceBasis*.

Parameters
`comm` – DEPRECATED pass to VectorSpaceBasis.__init__().

orthogonalize(*b*)

Orthogonalize `b` with respect to this *VectorSpaceBasis*.

Parameters
`b` – a *Function*

Note: Modifies `b` in place.

orthonormalize()

Orthonormalize the basis.

Warning: This modifies the basis *in place*.

4.26 firedrake.optimizer module

`firedrake.optimizer.`**slope**(*mesh*, *debug=False*)

Initialize the SLOPE library by providing information about the mesh, including:

- Mesh coordinates
- All available maps binding sets of mesh components

4.27 firedrake.output module

class firedrake.output.**File**(*filename*, *project_output=False*, *comm=None*, *mode='w'*, *target_degree=None*, *target_continuity=None*, *adaptive=False*)

Bases: `object`

Create an object for outputting data for visualisation.

This produces output in VTU format, suitable for visualisation with Paraview or other VTK-capable visualisation packages.

Parameters

- **filename** – The name of the output file (must end in `.pvd`).
- **project_output** – Should the output be projected to a computed output space? Default is to use interpolation.
- **comm** – The MPI communicator to use.
- **mode** – “w” to overwrite any existing file, “a” to append to an existing file.
- **target_degree** – override the degree of the output space.
- **target_continuity** – override the continuity of the output space; A UFL `SobolevSpace` object: *H1* for a continuous output and *L2* for a discontinuous output.
- **adaptive** – allow different meshes at different exports if *True*.

Note: Visualisation is only possible for Lagrange fields (either continuous or discontinuous). All other fields are first either projected or interpolated to Lagrange elements before storing for visualisation purposes.

write(**functions*, ***kwargs*)

Write functions to this *File*.

Parameters

- **functions** – list of functions to write.
- **time** – optional timestep value.

You may save more than one function to the same file. However, all calls to *write()* must use the same set of functions.

4.28 firedrake.parameters module

The parameters dictionary contains global parameter settings.

class firedrake.parameters.**Parameters**(*name=None*, ***kwargs*)

Bases: `dict`

add(*key*, *value=None*)

name()

rename(*name*)

set_update_function(*callable*)

Set a function to be called whenever a dictionary entry is changed.

Parameters

callable – the function.

The function receives two arguments, the key-value pair of updated entries.

firedrake.parameters.**disable_performance_optimisations**()

Switches off performance optimisations in Firedrake.

This is mostly useful for debugging purposes.

This switches off all of COFFEE's kernel compilation optimisations and enables PyOP2's runtime checking of par_loop arguments in all cases (even those where they are claimed safe). Additionally, it switches to compiling generated code in debug mode.

Returns a function that can be called with no arguments, to restore the state of the parameters dict.

firedrake.parameters.**parameters = {'coffee': {'optlevel': 'Ov'}, 'default_matrix_type': 'aij', 'default_sub_matrix_type': 'baij', 'form_compiler': {'mode': 'spectral', 'quadrature_degree': 'auto', 'quadrature_rule': 'auto', 'scalar_type': dtype('float64'), 'scalar_type_c': 'double', 'unroll_indexsum': 3}, 'pyop2_options': {'block_sparsity': True, 'cache_dir': '/data/jbetteri/fd_3.11_opt/.cache/pyop2', 'cc': '', 'cflags': '', 'check_src_hashes': True, 'compute_kernel_flops': False, 'cxx': '', 'cxxflags': '', 'debug': False, 'ld': '', 'ldflags': '', 'log_level': 'WARNING', 'matnest': True, 'no_fork_available': False, 'node_local_compilation': True, 'opt_level': 'Ov', 'print_cache_size': False, 'simd_width': 4, 'type_check': True}, 'reorder_meshes': True, 'slate_compiler': {'optimise': True, 'replace_mul': False}, 'type_check_safe_par_loops': False}**

A nested dictionary of parameters used by Firedrake

4.29 firedrake.paraview_reordering module

firedrake.paraview_reordering.**bary_to_cart**(*bar*)

firedrake.paraview_reordering.**firedrake_local_to_cart**(*element*)

Gets the list of nodes for an element (provided they exist.) :arg element: a ufl element. :returns: a list of arrays of floats where each array is a node.

firedrake.paraview_reordering.**invert**(*list1*, *list2*)

Given two maps (lists) from [0..N] to nodes, finds a permutations between them. :arg list1: a list of nodes. :arg list2: a second list of nodes. :returns: a list of integers, l, such that list1[x] = list2[l[x]]

firedrake.paraview_reordering.**tet_barycentric_index**(*tet*, *index*, *order*)

Wrapper for vtkLagrangeTetra::BarycentricIndex.

firedrake.paraview_reordering.**vtk_hex8_to_hex9**(*orders*)

Produce a list where element i is the vtk9 node number of node i in vtk8. For hexes only. :arg orders: the orders of the hex (the same integer 3 times) :return a list of integers

firedrake.paraview_reordering.**vtk_hex_local_to_cart**(*orders*)

Produces a list of nodes for VTK's lagrange hex basis. :arg order: the three orders of the hex basis. :return a list of arrays of floats.

firedrake.paraview_reordering.**vtk_interval_local_coord**(*i*, *order*)

See vtkLagrangeCurve::PointIndexFromIJK.

firedrake.paraview_reordering.**vtk_lagrange_hex_reorder**(*ufl_element*)

firedrake.paraview_reordering.**vtk_lagrange_interval_reorder**(*ufl_element*)

firedrake.paraview_reordering.**vtk_lagrange_quad_reorder**(*ufl_element*)

firedrake.paraview_reordering.**vtk_lagrange_tet_reorder**(*ufl_element*)

firedrake.paraview_reordering.**vtk_lagrange_triangle_reorder**(*ufl_element*)

firedrake.paraview_reordering.**vtk_lagrange_wedge_reorder**(*ufl_element*)

firedrake.paraview_reordering.**vtk_quad_local_to_cart**(*orders*)

Produces a list of nodes for VTK's lagrange quad basis. :arg order: the order of the quad basis. :return a list of arrays of floats.

firedrake.paraview_reordering.**vtk_tet_local_to_cart**(*order*)

Produces a list of nodes for VTK's lagrange tet basis. :arg order: the order of the tet :return a list of arrays of floats

firedrake.paraview_reordering.**vtk_triangle_index_cart**(*tri*, *index*, *order*)

Wrapper for vtkLagrangeTriangle::BarycentricIndex

firedrake.paraview_reordering.**vtk_triangle_local_to_cart**(*order*)

firedrake.paraview_reordering.**vtk_wedge_local_to_cart**(*ordersp*)

Produces a list of nodes for VTK's lagrange wedge basis. :arg order: the orders of the wedge (triangle, interval) :return a list of arrays of floats

4.30 firedrake.parloops module

This module implements parallel loops reading and writing *Function*s. This provides a mechanism for implementing non-finite element operations such as slope limiters.

`firedrake.parloops.`**`direct`**` = `**`direct`**

A singleton object which can be used in a *par_loop()* in place of the measure in order to indicate that the loop is a direct loop over degrees of freedom.

`firedrake.parloops.`**`par_loop`**(*kernel*, *measure*, *args*, *kernel_kwargs=None*, *is_loopy_kernel=False*, ***kwargs*)

A *par_loop()* is a user-defined operation which reads and writes *Function*s by looping over the mesh cells or facets and accessing the degrees of freedom on adjacent entities.

Parameters

- **kernel** – a string containing the C code to be executed. Or a 2-tuple of (domains, instructions) to create a loopy kernel (must also set `is_loopy_kernel=True`). If loopy syntax is used, the domains and instructions should be specified in loopy kernel syntax. See the loopy tutorial for details.
- **measure** – is a UFL `Measure` which determines the manner in which the iteration over the mesh is to occur. Alternatively, you can pass *direct* to designate a direct loop.
- **args** – is a dictionary mapping variable names in the kernel to *Function*s or components of mixed *Function*s and indicates how these *Function*s are to be accessed.
- **kernel_kwargs** – keyword arguments to be passed to the `Kernel` constructor
- **kwargs** – additional keyword arguments are passed to the underlying `par_loop`
- **iterate** – Optionally specify which region of an `ExtrudedSet` to iterate over. Valid values are the following objects from pyop2:
  - `ON_BOTTOM`: iterate over the bottom layer of cells.
  - `ON_TOP` iterate over the top layer of cells.
  - `ALL` iterate over all cells (the default if unspecified)
  - `ON_INTERIOR_FACETS` iterate over all the layers except the top layer, accessing data two adjacent (in the extruded direction) cells at a time.

Example

Assume that *A* is a *Function* in CG1 and *B* is a *Function* in DG0. Then the following code sets each DoF in *A* to the maximum value that *B* attains in the cells adjacent to that DoF:

```
A.assign(numpy.finfo(0.).min)
par_loop('for (int i=0; i<A.dofs; i++) A[i] = fmax(A[i], B[0]);', dx,
    {'A' : (A, RW), 'B': (B, READ)})
```

The equivalent using loopy kernel syntax is:

```
domain = '{[i]: 0 <= i < A.dofs}'
instructions = '''
for i
    A[i] = max(A[i], B[0])
end
'''
par_loop((domain, instructions), dx, {'A' : (A, RW), 'B': (B, READ)}, is_
↪loopy_kernel=True)
```

Argument definitions

Each item in the *args* dictionary maps a string to a tuple containing a `Function` or `Constant` and an argument intent. The string is the c language variable name by which this function will be accessed in the kernel. The argument intent indicates how the kernel will access this variable:

READ
: The variable will be read but not written to.

WRITE
: The variable will be written to but not read. If multiple kernel invocations write to the same DoF, then the order of these writes is undefined.

RW
: The variable will be both read and written to. If multiple kernel invocations access the same DoF, then the order of these accesses is undefined, but it is guaranteed that no race will occur.

INC
: The variable will be added into using +=. As before, the order in which the kernel invocations increment the variable is undefined, but there is a guarantee that no races will occur.

Note: Only *READ* intents are valid for `Constant` coefficients, and an error will be raised in other cases.

The measure

The measure determines the mesh entities over which the iteration will occur, and the size of the kernel stencil. The iteration will occur over the same mesh entities as if the measure had been used to define an integral, and the stencil will likewise be the same as the integral case. That is to say, if the measure is a volume measure, the kernel will be called once per cell and the DoFs accessible to the kernel will be those associated with the cell, its facets, edges and vertices. If the measure is a facet measure then the iteration will occur over the corresponding class of facets and the accessible DoFs will be those on the cell(s) adjacent to the facet, and on the facets, edges and vertices adjacent to those facets.

For volume measures the DoFs are guaranteed to be in the FInAT local DoFs order. For facet measures, the DoFs will be in sorted first by the cell to which they are adjacent. Within each cell, they will be in FInAT order. Note that if a continuous `Function` is accessed via an internal facet measure, the DoFs on the interface between the two facets will be accessible twice: once via each cell. The orientation of the cell(s) relative to the current facet is currently arbitrary.

A direct loop over nodes without any indirections can be specified by passing *direct* as the measure. In this case, all of the arguments must be *Functions* in the same *FunctionSpace*.

The kernel code

The kernel code is plain C in which the variables specified in the *args* dictionary are available to be read or written in according to the argument intent specified. Most basic C operations are permitted. However there are some restrictions:

- Only functions from math.h may be called.
- Pointer operations other than dereferencing arrays are prohibited.

Indirect free variables referencing *Functions* are all of type *double**. For spaces with rank greater than zero (Vector or TensorElement), the data are laid out XYZ… XYZ… XYZ…. With the vector/tensor component moving fastest.

In loopy syntax, these may be addressed using 2D indexing:

```
A[i, j]
```

Where `i` runs over nodes, and `j` runs over components.

In a direct *par_loop()*, the variables will all be of type *double** with the single index being the vector component.

Constants are always of type *double**, both for indirect and direct *par_loop()* calls.

4.31 firedrake.petsc module

class firedrake.petsc.**OptionsManager**(*parameters*, *options_prefix*)

Bases: `object`

commandline_options = frozenset({'W', 'b', 'd'})

count = count(0)

Mixin class that helps with managing setting petsc options.

Parameters

- **parameters** – The dictionary of parameters to use.
- **options_prefix** – The prefix to look up items in the global options database (may be `None`, in which case only entries from `parameters` will be considered. If no trailing underscore is provided, one is appended. Hence `foo_` and `foo` are treated equivalently. As an exception, if the prefix is the empty string, no underscore is appended.

To use this, you must call its constructor to with the parameters you want in the options database.

You then call *set_from_options()*, passing the PETSc object you'd like to call `setFromOptions` on. Note that this will actually only call `setFromOptions` the first time (so really this parameters object is a once-per-PETSc-object thing).

So that the runtime monitors which look in the options database actually see options, you need to ensure that the options database is populated at the time of a `SNESSolve` or `KSPSolve` call. Do that using the *`inserted_options()`* context manager.

```
with self.inserted_options():
    self.snes.solve(...)
```

This ensures that the options database has the relevant entries for the duration of the `with` block, before removing them afterwards. This is a much more robust way of dealing with the fixed-size options database than trying to clear it out using destructors.

This object can also be used only to manage insertion and deletion into the PETSc options database, by using the context manager.

`inserted_options`()

Context manager inside which the petsc options database contains the parameters from this object.

`options_object = <petsc4py.PETSc.Options object>`

`set_default_parameter`(*key*, *val*)

Set a default parameter value.

Parameters

- **key** – The parameter name
- **val** – The parameter value.

Ensures that the right thing happens cleaning up the options database.

`set_from_options`(*petsc_obj*)

Set up petsc_obj from the options database.

Parameters

petsc_obj – The PETSc object to call setFromOptions on.

Matt says: "Only ever call setFromOptions once". This function ensures we do so.

`firedrake.petsc.`**`get_petsc_variables`**()

Get dict of PETSc environment variables from the file: $PETSC_DIR/$PETSC_ARCH/lib/petsc/conf/petscvariables

The result is memoized to avoid constantly reading the file.

4.32 firedrake.plot module

class `firedrake.plot.`**`FunctionPlotter`**(*mesh*, *num_sample_points*)

Bases: `object`

`firedrake.plot.`**`plot`**(*function*, **args*, *bezier=False*, *num_sample_points=10*, *complex_component='real'*, ***kwargs*)

Plot a 1D Firedrake *`Function`*

Parameters

- **function** – The *Function* to plot
- **args** – same as for matplotlib `plot`
- **num_sample_points** – number of sample points for high-degree functions
- **complex_component** – If plotting complex data, which component? (`'real'` or `'imag'`). Default is `'real'`.
- **kwargs** – same as for matplotlib

Returns
list of matplotlib `Line2D`

`firedrake.plot.`**`quiver`**(*function*, ***, *complex_component='real'*, ***kwargs*)

Make a quiver plot of a 2D vector Firedrake *Function*

Parameters

- **function** – the vector field to plot
- **complex_component** – If plotting complex data, which component? (`'real'` or `'imag'`). Default is `'real'`.
- **kwargs** – same as for matplotlib `quiver`

Returns
matplotlib `Quiver` object

`firedrake.plot.`**`streamplot`**(*function*, *resolution=None*, *min_length=None*, *max_time=None*, *start_width=0.5*, *end_width=1.5*, *tolerance=0.003*, *loc_tolerance=1e-10*, *seed=None*, *complex_component='real'*, ***kwargs*)

Create a streamline plot of a vector field

Similar to matplotlib `streamplot`

Parameters

- **function** – the Firedrake *Function* to plot
- **resolution** – minimum spacing between streamlines (defaults to domain size / 20)
- **min_length** – minimum length of a streamline (defaults to 4x resolution)
- **max_time** – maximum time to integrate a streamline
- **start_width** – line width at beginning of streamline
- **end_width** – line width at end of streamline, to convey direction
- **tolerance** – dimensionless tolerance for adaptive ODE integration
- **loc_tolerance** – point location tolerance for `at()`
- **complex_component** – If plotting complex data, which component? (`'real'` or `'imag'`). Default is `'real'`.
- **kwargs** – same as for matplotlib `LineCollection`

`firedrake.plot.`**`tricontour`**(*function*, **args*, *complex_component='real'*, ***kwargs*)

Create a contour plot of a 2D Firedrake *`Function`*

If the input function is a vector field, the magnitude will be plotted.

Parameters

- **`function`** – the Firedrake *`Function`* to plot
- **`args`** – same as for matplotlib `tricontour`
- **`complex_component`** – If plotting complex data, which component? (`'real'` or `'imag'`). Default is `'real'`.
- **`kwargs`** – same as for matplotlib

Returns
matplotlib `ContourSet` object

`firedrake.plot.`**`tricontourf`**(*function*, **args*, *complex_component='real'*, ***kwargs*)

Create a filled contour plot of a 2D Firedrake *`Function`*

If the input function is a vector field, the magnitude will be plotted.

Parameters

- **`function`** – the Firedrake *`Function`* to plot
- **`args`** – same as for matplotlib `tricontourf`
- **`complex_component`** – If plotting complex data, which component? (`'real'` or `'imag'`). Default is `'real'`.
- **`kwargs`** – same as for matplotlib

Returns
matplotlib `ContourSet` object

`firedrake.plot.`**`tripcolor`**(*function*, **args*, *complex_component='real'*, ***kwargs*)

Create a pseudo-color plot of a 2D Firedrake *`Function`*

If the input function is a vector field, the magnitude will be plotted.

Parameters

- **`function`** – the function to plot
- **`args`** – same as for matplotlib `tripcolor`
- **`complex_component`** – If plotting complex data, which component? (`'real'` or `'imag'`). Default is `'real'`.
- **`kwargs`** – same as for matplotlib

Returns
matplotlib `PolyCollection` object

`firedrake.plot.`**`triplot`**(*mesh*, *axes=None*, *interior_kw={}*, *boundary_kw={}*)

Plot a mesh colouring marked facet segments

Typically boundary segments will be marked and coloured, but interior facets that are marked will also be coloured.

The interior and boundary keyword arguments can be any keyword argument for `LineCollection` and related types.

Parameters

- **`mesh`** – mesh to be plotted
- **`axes`** – matplotlib `Axes` object on which to plot mesh
- **`interior_kw`** – keyword arguments to apply when plotting the mesh interior
- **`boundary_kw`** – keyword arguments to apply when plotting the mesh boundary

Returns
list of matplotlib `Collection` objects

`firedrake.plot.`**`trisurf`**(*function*, **args*, *complex_component='real'*, ***kwargs*)

Create a 3D surface plot of a 2D Firedrake *`Function`*

If the input function is a vector field, the magnitude will be plotted.

Parameters

- **`function`** – the Firedrake *`Function`* to plot
- **`args`** – same as for matplotlib `plot_trisurf`
- **`complex_component`** – If plotting complex data, which component? (`'real'` or `'imag'`). Default is `'real'`.
- **`kwargs`** – same as for matplotlib

Returns
matplotlib `Poly3DCollection` object

4.33 firedrake.pointeval_utils module

`firedrake.pointeval_utils.`**`compile_element`**(*expression*, *coordinates*, *parameters=None*)

Generates C code for point evaluations.

Parameters

- **`expression`** – UFL expression
- **`coordinates`** – coordinate field
- **`parameters`** – form compiler parameters

Returns
C code as string

4.34 firedrake.pointquery_utils module

`firedrake.pointquery_utils.`**`X_isub_dX`**(*topological_dimension*)

`firedrake.pointquery_utils.`**`compile_coordinate_element`**(*ufl_coordinate_element*, *contains_eps*, *parameters=None*)

Generates C code for changing to reference coordinates.

Parameters
: **`ufl_coordinate_element`** – UFL element of the coordinates

Returns
: C code as string

`firedrake.pointquery_utils.`**`compute_celldist`**(*fiat_cell*, *X='X'*, *celldist='celldist'*)

`firedrake.pointquery_utils.`**`dX_norm_square`**(*topological_dimension*)

`firedrake.pointquery_utils.`**`init_X`**(*fiat_cell*, *parameters*)

`firedrake.pointquery_utils.`**`inside_check`**(*fiat_cell*, *eps*, *X='X'*)

`firedrake.pointquery_utils.`**`is_affine`**(*ufl_element*)

`firedrake.pointquery_utils.`**`make_args`**(*function*)

`firedrake.pointquery_utils.`**`make_wrapper`**(*function*, ***kwargs*)

`firedrake.pointquery_utils.`**`src_locate_cell`**(*mesh*, *tolerance=None*)

`firedrake.pointquery_utils.`**`to_reference_coordinates`**(*ufl_coordinate_element*, *parameters*)

4.35 firedrake.progress_bar module

A module providing progress bars.

`class` `firedrake.progress_bar.`**`ProgressBar`**(**args*, *comm=<mpi4py.MPI.Intracomm object>*, ***kwargs*)

Bases: `FillingSquaresBar`

A progress bar for simulation execution.

This is a subclass of `progress.bar.FillingSquaresBar` which is configured to be suitable for tracking progress in forward and adjoint simulations. It is also extended to only output on rank 0 in parallel.

Parameters

message
: [str] An identifying string to be prepended to the progress bar. This defaults to an empty string.

comm
[mpi4py.MPI.Intracomm] The MPI communicator over which the simulation is run. Defaults to *COMM_WORLD*

Notes

Further parameters can be passed as per the progress package documentation, or you can customise further by subclassing.

Examples

To apply a progress bar to a loop, wrap the loop iterator in the `iter()` method of a *ProgressBar*:

```
>>> for t in ProgressBar("Timestep").iter(np.linspace(0.0, 1.0, 10)):
...     sleep(0.2)
...
Timestep␣
↪⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡⊡␣
↪10/10 [0:00:02]
```

To see progress bars for functional, adjoint and Hessian evaluations in an adjoint simulation, set the `progress_bar` attribute of the tape to *ProgressBar*:

```
>>> get_working_tape().progress_bar = ProgressBar
```

This use case is covered in the documentation for `pyadjoint.Tape`.

`check_tty = False`

`suffix = '%(index)s/%(max)s [%(elapsed_td)s]'`

`width = 50`

4.36 firedrake.projection module

`firedrake.projection.`**`Projector`**(*v*, *v_out*, *bcs=None*, *solver_parameters=None*, *form_compiler_parameters=None*, *constant_jacobian=True*, *use_slate_for_inverse=False*)

A projector projects a UFL expression into a function space and places the result in a function from that function space, allowing the solver to be reused. Projection reverts to an assign operation if `v` is a *`Function`* and belongs to the same function space as `v_out`. It is possible to project onto the trace space 'DGT', but not onto other trace spaces e.g. into the restriction of CG onto the facets.

Parameters

- **v** – the `ufl.Expr` or *`Function`* to project
- **V** – *`Function`* (or *`FunctionSpace`*) to put the result in.

- **bcs** – an optional set of *DirichletBC* objects to apply on the target function space.
- **solver_parameters** – parameters to pass to the solver used when projecting.
- **constant_jacobian** – Is the projection matrix constant between calls? Say False if you have moving meshes.
- **use_slate_for_inverse** – compute mass inverse cell-wise using SLATE (only valid for DG function spaces).

firedrake.projection.**project**(*v*, *V*, *bcs=None*, *solver_parameters=None*, *form_compiler_parameters=None*, *use_slate_for_inverse=True*, *name=None*, *ad_block_tag=None*)

Project a UFL expression into a *FunctionSpace* It is possible to project onto the trace space 'DGT', but not onto other trace spaces e.g. into the restriction of CG onto the facets.

Parameters

- **v** – the `ufl.Expr` to project
- **V** – the *FunctionSpace* or *Function* to project into
- **bcs** – boundary conditions to apply in the projection
- **solver_parameters** – parameters to pass to the solver used when projecting.
- **form_compiler_parameters** – parameters to the form compiler
- **use_slate_for_inverse** – compute mass inverse cell-wise using SLATE (ignored for non-DG function spaces).
- **name** – name of the resulting *Function*
- **ad_block_tag** – string for tagging the resulting block on the Pyadjoint tape

If `V` is a *Function* then `v` is projected into `V` and `V` is returned. If *V* is a *FunctionSpace* then `v` is projected into a new *Function* and that *Function* is returned.

4.37 firedrake.randomfunctiongen module

4.37.1 Overview

This module wraps numpy.random, and enables users to generate a randomised *Function* from a *FunctionSpace*. This module inherits almost all attributes from numpy.random with the following changes:

Generator

A `Generator` wraps numpy.random.Generator. `Generator` inherits almost all distribution methods from numpy.random.Generator, and they can be used to generate a randomised *Function* by passing a *FunctionSpace* as the first argument.

Example:

```
from firedrake import *

mesh = UnitSquareMesh(2, 2)
V = FunctionSpace(mesh, 'CG', 1)
pcg = PCG64(seed=123456789)
rg = Generator(pcg)
f_beta = rg.beta(V, 1.0, 2.0)
print(f_beta.dat.data)
# prints:
# [0.0075147 0.40893448 0.18390776 0.46192167 0.20055854 0.02231147 0.
↪47424777 0.24177973 0.55937075]
```

BitGenerator

A `BitGenerator` is the base class for bit generators; see numpy.random.BitGenerator. A `BitGenerator` takes an additional keyword argument `comm` (defaulting to `COMM_WORLD`). If `comm.Get_rank() > 1`, `PCG64`, `PCG64DXSM`, or `Philox` should be used, as these bit generators are known to be parallel-safe.

PCG64

`PCG64` wraps numpy.random.PCG64. If `seed` keyword is not provided by the user, it is set using numpy.random.SeedSequence. To make `PCG64` automatically generate multiple streams in parallel, Firedrake preprocesses the `seed` as the following before passing it to numpy.random.PCG64:

```
rank = comm.Get_rank()
size = comm.Get_size()
sg = numpy.random.SeedSequence(seed)
seed = sg.spawn(size)[rank]
```

Note: `inc` is no longer a valid keyword for `PCG64` constructor. However, one can reset the `state` after construction as:

```
pcg = PCG64()
state = pcg.state
state['state'] = {'state': seed, 'inc': inc}
pcg.state = state
```

PCG64DXSM

`PCG64DXSM` wraps numpy.random.PCG64DXSM. If `seed` keyword is not provided by the user, it is set using numpy.random.SeedSequence. To make `PCG64DXSM` automatically generate multiple streams in parallel, Firedrake preprocesses the `seed` as the following before passing it to numpy.random.PCG64DXSM:

```
rank = comm.Get_rank()
size = comm.Get_size()
sg = numpy.random.SeedSequence(seed)
seed = sg.spawn(size)[rank]
```

Note: `inc` is no longer a valid keyword for `PCG64DXSM` constructor. However, one can reset the `state` after construction as:

```
pcg = PCG64DXSM()
state = pcg.state
state['state'] = {'state': seed, 'inc': inc}
pcg.state = state
```

Philox

`Philox` wraps numpy.random.Philox. If the `key` keyword is not provided by the user, `Philox` computes a default key as:

```
key = np.zeros(2, dtype=np.uint64)
key[0] = comm.Get_rank()
```

4.38 firedrake.solving module

`firedrake.solving.`**`solve`**`(`**args*, ***kwargs*`)`

Solve linear system Ax = b or variational problem a == L or F == 0.

The Firedrake solve() function can be used to solve either linear systems or variational problems. The following list explains the various ways in which the solve() function can be used.

1. Solving linear systems

A linear system Ax = b may be solved by calling

```
solve(A, x, b, bcs=bcs, solver_parameters={...})
```

where *A* is a *`Matrix`* and *x* and *b* are *`Function`*s. If present, *bcs* should be a list of *`DirichletBC`*s and *`EquationBC`*s specifying, respectively, the strong boundary conditions to apply and PDEs to solve on the boundaries. For the format of *solver_parameters* see below.

2. *Solving linear variational problems*

A linear variational problem a(u, v) = L(v) for all v may be solved by calling solve(a == L, u, ...), where a is a bilinear form, L is a linear form, u is a *Function* (the solution). Optional arguments may be supplied to specify boundary conditions or solver parameters. Some examples are given below:

```
solve(a == L, u)
solve(a == L, u, bcs=bc)
solve(a == L, u, bcs=[bc1, bc2])

solve(a == L, u, bcs=bcs,
      solver_parameters={"ksp_type": "gmres"})
```

The linear solver uses PETSc under the hood and accepts all PETSc options as solver parameters. For example, to solve the system using direct factorisation use:

```
solve(a == L, u, bcs=bcs,
      solver_parameters={"ksp_type": "preonly", "pc_type": "lu"})
```

3. *Solving nonlinear variational problems*

A nonlinear variational problem F(u; v) = 0 for all v may be solved by calling solve(F == 0, u, ...), where the residual F is a linear form (linear in the test function v but possibly nonlinear in the unknown u) and u is a *Function* (the solution). Optional arguments may be supplied to specify boundary conditions, the Jacobian form or solver parameters. If the Jacobian is not supplied, it will be computed by automatic differentiation of the residual form. Some examples are given below:

The nonlinear solver uses a PETSc SNES object under the hood. To pass options to it, use the same options names as you would for pure PETSc code. See `NonlinearVariationalSolver` for more details.

```
solve(F == 0, u)
solve(F == 0, u, bcs=bc)
solve(F == 0, u, bcs=[bc1, bc2])

solve(F == 0, u, bcs, J=J,
      # Use Newton-Krylov iterations to solve the nonlinear
      # system, using direct factorisation to solve the linear system.
      solver_parameters={"snes_type": "newtonls",
                         "ksp_type" : "preonly",
                         "pc_type" : "lu"})
```

In all three cases, if the operator is singular you can pass a *VectorSpaceBasis* (or *MixedVectorSpaceBasis*) spanning the null space of the operator to the solve call using the `nullspace` keyword argument.

If you need to project the transpose nullspace out of the right hand side, you can do so by using the `transpose_nullspace` keyword argument.

In the same fashion you can add the near nullspace using the `near_nullspace` keyword argument.

4.39 firedrake.solving_utils module

`firedrake.solving_utils.`**`check_snes_convergence`**`(`*snes*`)`

`firedrake.solving_utils.`**`set_defaults`**`(`*solver_parameters*, *arguments*, ***, *ksp_defaults={}*, *snes_defaults={}*`)`

Set defaults for solver parameters.

Parameters

- **`solver_parameters`** – dict of user solver parameters to override/extend defaults
- **`arguments`** – arguments for the bilinear form (need to know if we have a Real block).
- **`ksp_defaults`** – Default KSP parameters.
- **`snes_defaults`** – Default SNES parameters.

4.40 firedrake.supermeshing module

`firedrake.supermeshing.`**`assemble_mixed_mass_matrix`**`(`*V_A*, *V_B*`)`

Construct the mixed mass matrix of two function spaces, using the TrialFunction from V_A and the TestFunction from V_B.

`firedrake.supermeshing.`**`intersection_finder`**`()`

4.41 firedrake.tsfc_interface module

Provides the interface to TSFC for compiling a form, and transforms the TSFC-generated code to make it suitable for passing to the backends.

`class` `firedrake.tsfc_interface.`**`KernelInfo`**`(`*kernel*, *integral_type*, *oriented*, *subdomain_id*, *domain_number*, *coefficient_map*, *needs_cell_facets*, *pass_layer_arg*, *needs_cell_sizes*, *arguments*, *events*`)`

Bases: `tuple`

Create new instance of KernelInfo(kernel, integral_type, oriented, subdomain_id, domain_number, coefficient_map, needs_cell_facets, pass_layer_arg, needs_cell_sizes, arguments, events)

`arguments`

Alias for field number 9

`coefficient_map`

Alias for field number 5

`domain_number`

Alias for field number 4

events
Alias for field number 10

integral_type
Alias for field number 1

kernel
Alias for field number 0

needs_cell_facets
Alias for field number 6

needs_cell_sizes
Alias for field number 8

oriented
Alias for field number 2

pass_layer_arg
Alias for field number 7

subdomain_id
Alias for field number 3

class firedrake.tsfc_interface.**SplitKernel**(*indices*, *kinfo*)

Bases: `tuple`

Create new instance of SplitKernel(indices, kinfo)

indices
Alias for field number 0

kinfo
Alias for field number 1

class firedrake.tsfc_interface.**TSFCKernel**(**args*, ***kwargs*)

Bases: `Cached`

A wrapper object for one or more TSFC kernels compiled from a given `Form`.

Parameters

- **form** – the `Form` from which to compile the kernels.
- **name** – a prefix to be applied to the compiled kernel names. This is primarily useful for debugging.
- **parameters** – a dict of parameters to pass to the form compiler.
- **number_map** – a map from local coefficient numbers to the global coefficient numbers.
- **interface** – the KernelBuilder interface for TSFC (may be None)

firedrake.tsfc_interface.**as_pyop2_local_kernel**(*ast*, *name*, *nargs*, *access=Access.INC*, ***kwargs*)

Convert a loopy kernel to a PyOP2 `pyop2.LocalKernel`.

Parameters

- **ast** – The kernel code. This could be, for example, a loopy kernel.
- **name** – The kernel name.
- **nargs** – The number of arguments expected by the kernel.
- **access** – Access descriptor for the first kernel argument.

firedrake.tsfc_interface.**clear_cache**(*comm=None*)

Clear the Firedrake TSFC kernel cache.

firedrake.tsfc_interface.**compile_form**(*form*, *name*, *parameters=None*, *split=True*, *interface=None*, *coffee=False*, *diagonal=False*)

Compile a form using TSFC.

Parameters

- **form** – the `Form` to compile.
- **name** – a prefix for the generated kernel functions.
- **parameters** – optional dict of parameters to pass to the form compiler. If not provided, parameters are read from the `form_compiler` slot of the Firedrake *parameters* dictionary (which see).
- **split** – If `False`, then don't split mixed forms.
- **coffee** – compile coffee kernel instead of loopy kernel

Returns a tuple of tuples of (index, integral type, subdomain id, coordinates, coefficients, needs_orientations, `Kernels`).

`needs_orientations` indicates whether the form requires cell orientation information (for correctly pulling back to reference elements on embedded manifolds).

The coordinates are extracted from the domain of the integral (a *Mesh()*)

firedrake.tsfc_interface.**extract_numbered_coefficients**(*expr*, *numbers*)

Return expression coefficients specified by a numbering.

Parameters

- **expr** – A UFL expression.
- **numbers** – Iterable of indices used for selecting the correct coefficients from `expr`.

Returns

A list of UFL coefficients.

firedrake.tsfc_interface.**gather_integer_subdomain_ids**(*knls*)

Gather a dict of all integer subdomain IDs per integral type.

This is needed to correctly interpret the `"otherwise"` subdomain ID.

Parameters

knls – Iterable of *SplitKernel* objects.

4.42 firedrake.ufl_expr module

class firedrake.ufl_expr.**Argument**(*function_space*, *number*, *part=None*)

Bases: `Argument`

Representation of the argument to a form.

Parameters

- **function_space** – the *FunctionSpace* the argument corresponds to.
- **number** – the number of the argument being constructed.
- **part** – optional index (mostly ignored).

Note: an *Argument* with a number of `0` is used as a *TestFunction()*, with a number of `1` it is used as a *TrialFunction()*.

cell_node_map

exterior_facet_node_map

function_space()

interior_facet_node_map

make_dat()

reconstruct(*function_space=None*, *number=None*, *part=None*)

firedrake.ufl_expr.**CellSize**(*mesh*)

A symbolic representation of the cell size of a mesh.

Parameters
mesh – the mesh for which to calculate the cell size.

firedrake.ufl_expr.**FacetNormal**(*mesh*)

A symbolic representation of the facet normal on a cell in a mesh.

Parameters
mesh – the mesh over which the normal should be represented.

firedrake.ufl_expr.**TestFunction**(*function_space*, *part=None*)

Build a test function on the specified function space.

Parameters

- **function_space** – the *FunctionSpace* to build the test function on.
- **part** – optional index (mostly ignored).

firedrake.ufl_expr.**TestFunctions**(*function_space*)

Return a tuple of test functions on the specified function space.

Parameters
function_space – the *FunctionSpace* to build the test functions on.

This returns `len(function_space)` test functions, which, if the function space is a *MixedFunctionSpace*, are indexed appropriately.

`firedrake.ufl_expr.`**`TrialFunction`**(*function_space*, *part=None*)

Build a trial function on the specified function space.

Parameters

- **function_space** – the *FunctionSpace* to build the trial function on.
- **part** – optional index (mostly ignored).

`firedrake.ufl_expr.`**`TrialFunctions`**(*function_space*)

Return a tuple of trial functions on the specified function space.

Parameters

function_space – the *FunctionSpace* to build the trial functions on.

This returns `len(function_space)` trial functions, which, if the function space is a *MixedFunctionSpace*, are indexed appropriately.

`firedrake.ufl_expr.`**`action`**(*form*, *coefficient*)

Compute the action of a form on a coefficient.

Parameters

- **form** – A UFL form, or a Slate tensor.
- **coefficient** – The *Function* to act on.

Returns

a symbolic expression for the action.

`firedrake.ufl_expr.`**`adjoint`**(*form*, *reordered_arguments=None*)

Compute the adjoint of a form.

Parameters

- **form** – A UFL form, or a Slate tensor.
- **reordered_arguments** – arguments to use when creating the adjoint. Ignored if form is a Slate tensor.

If the form is a slate tensor, this just returns its transpose. Otherwise, given a bilinear form, compute the adjoint form by changing the ordering (number) of the test and trial functions.

By default, new Argument objects will be created with opposite ordering. However, if the adjoint form is to be added to other forms later, their arguments must match. In that case, the user must provide a tuple reordered_arguments=(u2,v2).

`firedrake.ufl_expr.`**`derivative`**(*form*, *u*, *du=None*, *coefficient_derivatives=None*)

Compute the derivative of a form.

Given a form, this computes its linearization with respect to the provided *Function*. The resulting form has one additional *Argument* in the same finite element space as the Function.

Parameters

- **form** – a `Form` to compute the derivative of.

- **u** – a *Function* to compute the derivative with respect to.
- **du** – an optional *Argument* to use as the replacement in the new form (constructed automatically if not provided).
- **coefficient_derivatives** – an optional `dict` to provide the derivative of a coefficient function.

Raises

ValueError – If any of the coefficients in `form` were obtained from `u.split()`. UFL doesn't notice that these are related to `u` and so therefore the derivative is wrong (instead one should have written `split(u)`).

See also `ufl.derivative()`.

4.43 firedrake.utility_meshes module

`firedrake.utility_meshes.`**`BoxMesh`**(*nx*, *ny*, *nz*, *Lx*, *Ly*, *Lz*, *hexahedral=False*, *reorder=None*, *distribution_parameters=None*, *diagonal='default'*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a mesh of a 3D box.

Parameters

- **nx** – The number of cells in the x direction
- **ny** – The number of cells in the y direction
- **nz** – The number of cells in the z direction
- **Lx** – The extent in the x direction
- **Ly** – The extent in the y direction
- **Lz** – The extent in the z direction
- **hexahedral** – (optional), creates hexahedral mesh, defaults to False
- **diagonal** – Two ways of cutting hexadra, should be cut into 6 tetrahedra (`"default"`), or 5 tetrahedra thus less biased (`"crossed"`)
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).

The boundary surfaces are numbered as follows:

- 1: plane x == 0
- 2: plane x == Lx
- 3: plane y == 0
- 4: plane y == Ly
- 5: plane z == 0

- 6: plane z == Lz

`firedrake.utility_meshes.`**`CircleManifoldMesh`**(*ncells*, *radius=1*, *degree=1*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generated a 1D mesh of the circle, immersed in 2D.

Parameters

- **ncells** – number of cells the circle should be divided into (min 3)
- **radius** – (optional) radius of the circle to approximate (defaults to 1).
- **degree** – polynomial degree of coordinate space (defaults to 1: cells are straight line segments)
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

`firedrake.utility_meshes.`**`CubeMesh`**(*nx*, *ny*, *nz*, *L*, *hexahedral=False*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a mesh of a cube

Parameters

- **nx** – The number of cells in the x direction
- **ny** – The number of cells in the y direction
- **nz** – The number of cells in the z direction
- **L** – The extent in the x, y and z directions
- **hexahedral** – (optional), creates hexahedral mesh, defaults to False
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

The boundary surfaces are numbered as follows:

- 1: plane x == 0
- 2: plane x == L
- 3: plane y == 0
- 4: plane y == L
- 5: plane z == 0
- 6: plane z == L

`firedrake.utility_meshes.`**`CubedSphereMesh`**(*radius*, *refinement_level=0*, *degree=1*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate an cubed approximation to the surface of the sphere.

Parameters

- **radius** – The radius of the sphere to approximate.
- **refinement_level** – optional number of refinements (0 is a cube).
- **degree** – polynomial degree of coordinate space (defaults to 1: bilinear quads)
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

`firedrake.utility_meshes.`**`CylinderMesh`**(*nr*, *nl*, *radius=1*, *depth=1*, *longitudinal_direction='z'*, *quadrilateral=False*, *reorder=None*, *distribution_parameters=None*, *diagonal=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generates a cylinder mesh.

Parameters

- **nr** – number of cells the cylinder circumference should be divided into (min 3)
- **nl** – number of cells along the longitudinal axis of the cylinder

- **radius** – (optional) radius of the cylinder to approximate (default 1).
- **depth** – (optional) depth of the cylinder to approximate (default 1).
- **longitudinal_direction** – (option) direction for the longitudinal axis of the cylinder.
- **quadrilateral** – (optional), creates quadrilateral mesh, defaults to False
- **diagonal** – (optional), one of `"crossed"`, `"left"`, `"right"`. `"left"` is the default. Not valid for quad meshes.
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

The boundary edges in this mesh are numbered as follows:

- 1: plane l == 0 (bottom)
- 2: plane l == depth (top)

`firedrake.utility_meshes.`**`IcosahedralSphereMesh`**(*radius*, *refinement_level=0*, *degree=1*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate an icosahedral approximation to the surface of the sphere.

Parameters

- **radius** – The radius of the sphere to approximate. For a radius R the edge length of the underlying icosahedron will be.

$$a = \frac{R}{\sin(2\pi/5)}$$

- **refinement_level** – optional number of refinements (0 is an icosahedron).
- **degree** – polynomial degree of coordinate space (defaults to 1: flat triangles)
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.

- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

firedrake.utility_meshes.**IntervalMesh**(*ncells*, *length_or_left*, *right=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a uniform mesh of an interval.

Parameters

- **ncells** – The number of the cells over the interval.
- **length_or_left** – The length of the interval (if `right` is not provided) or else the left hand boundary point.
- **right** – (optional) position of the right boundary point (in which case `length_or_left` should be the left boundary point).
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

The left hand boundary point has boundary marker 1, while the right hand point has marker 2.

firedrake.utility_meshes.**OctahedralSphereMesh**(*radius*, *refinement_level=0*, *degree=1*, *hemisphere='both'*, *z0=0.8*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate an octahedral approximation to the surface of the sphere.

Parameters

- **radius** – The radius of the sphere to approximate.
- **refinement_level** – optional number of refinements (0 is an octahedron).
- **degree** – polynomial degree of coordinate space (defaults to 1: flat triangles)
- **hemisphere** – One of “both” (default), “north”, or “south”

- **z0** – for abs(z/R)>z0, blend from a mesh where the higher-order non-vertex nodes are on lines of latitude to a mesh where these nodes are just pushed out radially from the equivalent P1 mesh. (defaults to z0=0.8).
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

firedrake.utility_meshes.**PeriodicBoxMesh**(*nx*, *ny*, *nz*, *Lx*, *Ly*, *Lz*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a periodic mesh of a 3D box.

Parameters

- **nx** – The number of cells in the x direction
- **ny** – The number of cells in the y direction
- **nz** – The number of cells in the z direction
- **Lx** – The extent in the x direction
- **Ly** – The extent in the y direction
- **Lz** – The extent in the z direction
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

firedrake.utility_meshes.**PeriodicIntervalMesh**(*ncells*, *length*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a periodic mesh of an interval.

Parameters

- **ncells** – The number of cells over the interval.
- **length** – The length the interval.
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

`firedrake.utility_meshes.`**`PeriodicRectangleMesh`**(*nx*, *ny*, *Lx*, *Ly*, *direction='both'*, *quadrilateral=False*, *reorder=None*, *distribution_parameters=None*, *diagonal=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a periodic rectangular mesh

Parameters

- **nx** – The number of cells in the x direction
- **ny** – The number of cells in the y direction
- **Lx** – The extent in the x direction
- **Ly** – The extent in the y direction
- **direction** – The direction of the periodicity, one of `"both"`, `"x"` or `"y"`.
- **quadrilateral** – (optional), creates quadrilateral mesh, defaults to False
- **reorder** – (optional), should the mesh be reordered
- **diagonal** – (optional), one of `"crossed"`, `"left"`, `"right"`. `"left"` is the default. Not valid for quad meshes. Only used for direction `"x"` or direction `"y"`.
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

If direction == “x” the boundary edges in this mesh are numbered as follows:

- 1: plane y == 0

- 2: plane y == Ly

If direction == "y" the boundary edges are:

- 1: plane x == 0
- 2: plane x == Lx

`firedrake.utility_meshes.`**`PeriodicSquareMesh`**(*nx*, *ny*, *L*, *direction='both'*, *quadrilateral=False*, *reorder=None*, *distribution_parameters=None*, *diagonal=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a periodic square mesh

Parameters

- **nx** – The number of cells in the x direction
- **ny** – The number of cells in the y direction
- **L** – The extent in the x and y directions
- **direction** – The direction of the periodicity, one of `"both"`, `"x"` or `"y"`.
- **quadrilateral** – (optional), creates quadrilateral mesh, defaults to False
- **reorder** – (optional), should the mesh be reordered
- **diagonal** – (optional), one of `"crossed"`, `"left"`, `"right"`. `"left"` is the default. Not valid for quad meshes.
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

If direction == "x" the boundary edges in this mesh are numbered as follows:

- 1: plane y == 0
- 2: plane y == L

If direction == "y" the boundary edges are:

- 1: plane x == 0
- 2: plane x == L

firedrake.utility_meshes.**PeriodicUnitCubeMesh**(*nx*, *ny*, *nz*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a periodic mesh of a unit cube

Parameters

- **nx** – The number of cells in the x direction
- **ny** – The number of cells in the y direction
- **nz** – The number of cells in the z direction
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

firedrake.utility_meshes.**PeriodicUnitIntervalMesh**(*ncells*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a periodic mesh of the unit interval

Parameters

- **ncells** – The number of cells in the interval.
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

firedrake.utility_meshes.**PeriodicUnitSquareMesh**(*nx*, *ny*, *direction='both'*, *reorder=None*, *quadrilateral=False*, *distribution_parameters=None*, *diagonal=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a periodic unit square mesh

Parameters

- **nx** – The number of cells in the x direction
- **ny** – The number of cells in the y direction
- **direction** – The direction of the periodicity, one of `"both"`, `"x"` or `"y"`.
- **quadrilateral** – (optional), creates quadrilateral mesh, defaults to False
- **reorder** – (optional), should the mesh be reordered
- **diagonal** – (optional), one of `"crossed"`, `"left"`, `"right"`. `"left"` is the default. Not valid for quad meshes.
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

If direction == "x" the boundary edges in this mesh are numbered as follows:

- 1: plane y == 0
- 2: plane y == 1

If direction == "y" the boundary edges are:

- 1: plane x == 0
- 2: plane x == 1

`firedrake.utility_meshes.`**`RectangleMesh`**(*nx*, *ny*, *Lx*, *Ly*, *quadrilateral=False*, *reorder=None*, *diagonal='left'*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a rectangular mesh

Parameters

- **nx** – The number of cells in the x direction
- **ny** – The number of cells in the y direction
- **Lx** – The extent in the x direction
- **Ly** – The extent in the y direction
- **quadrilateral** – (optional), creates quadrilateral mesh, defaults to False

- **reorder** – (optional), should the mesh be reordered
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **diagonal** – For triangular meshes, should the diagonal got from bottom left to top right (`"right"`), or top left to bottom right (`"left"`), or put in both diagonals (`"crossed"`).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

The boundary edges in this mesh are numbered as follows:

- 1: plane x == 0
- 2: plane x == Lx
- 3: plane y == 0
- 4: plane y == Ly

`firedrake.utility_meshes.`**`SquareMesh`**(*nx*, *ny*, *L*, *reorder=None*, *quadrilateral=False*, *diagonal='left'*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a square mesh

Parameters

- **nx** – The number of cells in the x direction
- **ny** – The number of cells in the y direction
- **L** – The extent in the x and y directions
- **quadrilateral** – (optional), creates quadrilateral mesh, defaults to False
- **reorder** – (optional), should the mesh be reordered
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

The boundary edges in this mesh are numbered as follows:

- 1: plane x == 0

- 2: plane x == L
- 3: plane y == 0
- 4: plane y == L

`firedrake.utility_meshes.`**`TensorRectangleMesh`**(*xcoords*, *ycoords*, *quadrilateral=False*, *reorder=None*, *diagonal='left'*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a rectangular mesh

Parameters

- **`xcoords`** – mesh points for the x direction
- **`ycoords`** – mesh points for the y direction
- **`quadrilateral`** – (optional), creates quadrilateral mesh, defaults to False
- **`reorder`** – (optional), should the mesh be reordered
- **`comm`** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **`diagonal`** – For triangular meshes, should the diagonal got from bottom left to top right (`"right"`), or top left to bottom right (`"left"`), or put in both diagonals (`"crossed"`).

The boundary edges in this mesh are numbered as follows:

- 1: plane x == xcoords[0]
- 2: plane x == xcoords[-1]
- 3: plane y == ycoords[0]
- 4: plane y == ycoords[-1]

`firedrake.utility_meshes.`**`TorusMesh`**(*nR*, *nr*, *R*, *r*, *quadrilateral=False*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a toroidal mesh

Parameters

- **`nR`** – The number of cells in the major direction (min 3)
- **`nr`** – The number of cells in the minor direction (min 3)
- **`R`** – The major radius
- **`r`** – The minor radius
- **`quadrilateral`** – (optional), creates quadrilateral mesh, defaults to False

- **reorder** – (optional), should the mesh be reordered
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

firedrake.utility_meshes.**UnitBallMesh**(*refinement_level=0*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a mesh of the unit ball in 3D

Parameters

- **refinement_level** – optional number of refinements (0 is an octahedron)
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional MPI communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

firedrake.utility_meshes.**UnitCubeMesh**(*nx*, *ny*, *nz*, *hexahedral=False*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a mesh of a unit cube

Parameters

- **nx** – The number of cells in the x direction
- **ny** – The number of cells in the y direction
- **nz** – The number of cells in the z direction
- **hexahedral** – (optional), creates hexahedral mesh, defaults to False
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).

- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

The boundary surfaces are numbered as follows:

- 1: plane x == 0
- 2: plane x == 1
- 3: plane y == 0
- 4: plane y == 1
- 5: plane z == 0
- 6: plane z == 1

firedrake.utility_meshes.**UnitCubedSphereMesh**(*refinement_level=0*, *degree=1*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a cubed approximation to the unit sphere.

Parameters

- **refinement_level** – optional number of refinements (0 is a cube).
- **degree** – polynomial degree of coordinate space (defaults to 1: bilinear quads)
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

firedrake.utility_meshes.**UnitDiskMesh**(*refinement_level=0*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a mesh of the unit disk in 2D

Parameters

- **refinement_level** – optional number of refinements (0 is a diamond)
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

firedrake.utility_meshes.**UnitIcosahedralSphereMesh**(*refinement_level=0*, *degree=1*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate an icosahedral approximation to the unit sphere.

Parameters

- **refinement_level** – optional number of refinements (0 is an icosahedron).
- **degree** – polynomial degree of coordinate space (defaults to 1: flat triangles)
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

firedrake.utility_meshes.**UnitIntervalMesh**(*ncells*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a uniform mesh of the interval [0,1].

Parameters

- **ncells** – The number of the cells over the interval.
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).

- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

The left hand ($x = 0$) boundary point has boundary marker 1, while the right hand ($x = 1$) point has marker 2.

firedrake.utility_meshes.**UnitOctahedralSphereMesh**(*refinement_level=0*, *degree=1*, *hemisphere='both'*, *z0=0.8*, *reorder=None*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate an octahedral approximation to the unit sphere.

Parameters

- **refinement_level** – optional number of refinements (0 is an octahedron).
- **degree** – polynomial degree of coordinate space (defaults to 1: flat triangles)
- **hemisphere** – One of “both” (default), “north”, or “south”
- **z0** – for abs(z)>z0, blend from a mesh where the higher-order non-vertex nodes are on lines of latitude to a mesh where these nodes are just pushed out radially from the equivalent P1 mesh. (defaults to z0=0.8).
- **reorder** – (optional), should the mesh be reordered?
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

firedrake.utility_meshes.**UnitSquareMesh**(*nx*, *ny*, *reorder=None*, *diagonal='left'*, *quadrilateral=False*, *distribution_parameters=None*, *comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a unit square mesh

Parameters

- **nx** – The number of cells in the x direction
- **ny** – The number of cells in the y direction
- **quadrilateral** – (optional), creates quadrilateral mesh, defaults to False
- **reorder** – (optional), should the mesh be reordered
- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

The boundary edges in this mesh are numbered as follows:

- 1: plane x == 0
- 2: plane x == 1
- 3: plane y == 0
- 4: plane y == 1

firedrake.utility_meshes.**UnitTetrahedronMesh**(*comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a mesh of the reference tetrahedron.

Parameters

- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.
- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

firedrake.utility_meshes.**UnitTriangleMesh**(*comm=<mpi4py.MPI.Intracomm object>*, *name='firedrake_default'*, *distribution_name=None*, *permutation_name=None*)

Generate a mesh of the reference triangle

Parameters

- **comm** – Optional communicator to build the mesh on (defaults to COMM_WORLD).
- **name** – Optional name of the mesh.

- **distribution_name** – the name of parallel distribution used when checkpointing; if *None*, the name is automatically generated.
- **permutation_name** – the name of entity permutation (reordering) used when checkpointing; if *None*, the name is automatically generated.

4.44 firedrake.utils module

`firedrake.utils.`**`known_pyop2_safe`**(*f*)

Decorator to mark a function as being PyOP2 type-safe.

This switches the current PyOP2 type checking mode to the value given by the parameter "type_check_safe_par_loops", and restores it after the function completes.

`firedrake.utils.`**`split_by`**(*condition*, *items*)

Split an iterable in two according to some condition.

Parameters

- **condition** – Callable applied to each item in `items`, returning `True` or `False`.
- **items** – Iterable to split apart.

Returns

A 2-tuple of the form `(yess, nos)`, where `yess` is a tuple containing the entries of `items` where `condition` is `True` and `nos` is a tuple of those where `condition` is `False`.

`firedrake.utils.`**`tuplify`**(*item*)

Convert an object into a hashable equivalent.

This is particularly useful for caching dictionaries of parameters such as *form_compiler_parameters* from `firedrake.assemble.assemble()`.

Parameters

item – The object to attempt to 'tuplify'.

Returns

The object interpreted as a tuple. For hashable objects this is simply a 1-tuple containing *item*. For dictionaries the function is called recursively on the values of the dict. For example, *{"a": 5, "b": 8}* returns *(("a", (5,)), ("b", (8,)))*.

`firedrake.utils.`**`unique_name`**(*name*, *nameset*)

Return name if name is not in nameset, or a deterministic uniquified name if name is in nameset. The new name is inserted into nameset to prevent further name clashes.

4.45 firedrake.variational_solver module

class firedrake.variational_solver.**LinearVariationalProblem**(*a*, *L*, *u*, *bcs=None*, *aP=None*, *form_compiler_parameters=None*, *constant_jacobian=False*)

Bases: *NonlinearVariationalProblem*

Linear variational problem a(u, v) = L(v).

Parameters

- **a** – the bilinear form
- **L** – the linear form
- **u** – the *Function* to which the solution will be assigned
- **bcs** – the boundary conditions (optional)
- **aP** – an optional operator to assemble to precondition the system (if not provided a preconditioner may be computed from a)
- **form_compiler_parameters** (*dict*) – parameters to pass to the form compiler (optional)
- **constant_jacobian** – (optional) flag indicating that the Jacobian is constant (i.e. does not depend on varying fields). If your Jacobian does not change, set this flag to True.

class firedrake.variational_solver.**LinearVariationalSolver**(*problem*, ***, *solver_parameters=None*, *options_prefix=None*, *nullspace=None*, *transpose_nullspace=None*, *near_nullspace=None*, *appctx=None*, *pre_jacobian_callback=None*, *post_jacobian_callback=None*, *pre_function_callback=None*, *post_function_callback=None*)

Bases: *NonlinearVariationalSolver*

Solves a *LinearVariationalProblem*.

Parameters

- **problem** – A *LinearVariationalProblem* to solve.
- **solver_parameters** – Solver parameters to pass to PETSc. This should be a dict mapping PETSc options to values.
- **nullspace** – an optional *VectorSpaceBasis* (or *MixedVectorSpaceBasis*) spanning the null space of the operator.
- **transpose_nullspace** – as for the nullspace, but used to make the right hand side consistent.

- **options_prefix** – an optional prefix used to distinguish PETSc options. If not provided a unique prefix will be created. Use this option if you want to pass options to the solver from the command line in addition to through the `solver_parameters` dict.
- **appctx** – A dictionary containing application context that is passed to the preconditioner if matrix-free.
- **pre_jacobian_callback** – A user-defined function that will be called immediately before Jacobian assembly. This can be used, for example, to update a coefficient function that has a complicated dependence on the unknown solution.
- **post_jacobian_callback** – As above, but called after the Jacobian has been assembled.
- **pre_function_callback** – As above, but called immediately before residual assembly.
- **post_function_callback** – As above, but called immediately after residual assembly.

See also *NonlinearVariationalSolver* for nonlinear problems.

Parameters

- **problem** – A *NonlinearVariationalProblem* to solve.
- **nullspace** – an optional *VectorSpaceBasis* (or *MixedVectorSpaceBasis*) spanning the null space of the operator.
- **transpose_nullspace** – as for the nullspace, but used to make the right hand side consistent.
- **near_nullspace** – as for the nullspace, but used to specify the near nullspace (for multigrid solvers).
- **solver_parameters** – Solver parameters to pass to PETSc. This should be a dict mapping PETSc options to values.
- **appctx** – A dictionary containing application context that is passed to the preconditioner if matrix-free.
- **options_prefix** – an optional prefix used to distinguish PETSc options. If not provided a unique prefix will be created. Use this option if you want to pass options to the solver from the command line in addition to through the `solver_parameters` dict.
- **pre_jacobian_callback** – A user-defined function that will be called immediately before Jacobian assembly. This can be used, for example, to update a coefficient function that has a complicated dependence on the unknown solution.
- **post_jacobian_callback** – As above, but called after the Jacobian has been assembled.
- **pre_function_callback** – As above, but called immediately before residual assembly.

- **post_function_callback** – As above, but called immediately after residual assembly.

Example usage of the `solver_parameters` option: to set the nonlinear solver type to just use a linear solver, use

```
{'snes_type': 'ksponly'}
```

PETSc flag options (where the presence of the option means something) should be specified with `None`. For example:

```
{'snes_monitor': None}
```

To use the `pre_jacobian_callback` or `pre_function_callback` functionality, the user-defined function must accept the current solution as a petsc4py Vec. Example usage is given below:

```
def update_diffusivity(current_solution):
    with cursol.dat.vec_wo as v:
        current_solution.copy(v)
    solve(trial*test*dx == dot(grad(cursol), grad(test))*dx, diffusivity)

solver = NonlinearVariationalSolver(problem,
                                    pre_jacobian_callback=update_
↪diffusivity)
```

DEFAULT_KSP_PARAMETERS = {'ksp_rtol': 1e-07, 'ksp_type': 'preonly', 'mat_mumps_icntl_14': 200, 'mat_type': 'aij', 'pc_factor_mat_solver_type': 'mumps', 'pc_type': 'lu'}

DEFAULT_SNES_PARAMETERS = {'snes_type': 'ksponly'}

invalidate_jacobian()

Forces the matrix to be reassembled next time it is required.

class firedrake.variational_solver.**NonlinearVariationalProblem**(*F*, *u*, *bcs=None*, *J=None*, *Jp=None*, *form_compiler_parameters=None* *is_linear=False*)

Bases: *NonlinearVariationalProblemMixin*

Nonlinear variational problem F(u; v) = 0.

Parameters

- **F** – the nonlinear form
- **u** – the *Function* to solve for
- **bcs** – the boundary conditions (optional)
- **J** – the Jacobian J = dF/du (optional)
- **Jp** – a form used for preconditioning the linear system, optional, if not supplied then the Jacobian itself will be used.

- **form_compiler_parameters** (*dict*) – parameters to pass to the form compiler (optional)

Is_linear
internally used to check if all domain/bc forms are given either in 'A == b' style or in 'F == 0' style.

dirichlet_bcs()

dm

class firedrake.variational_solver.**NonlinearVariationalSolver**(*problem*, ***, *solver_parameters=None*, *options_prefix=None*, *nullspace=None*, *transpose_nullspace=None*, *near_nullspace=None*, *appctx=None*, *pre_jacobian_callback=None*, *post_jacobian_callback=None*, *pre_function_callback=None*, *post_function_callback=None*)

Bases: *OptionsManager*, *NonlinearVariationalSolverMixin*

Solves a *NonlinearVariationalProblem*.

Parameters

- **problem** – A *NonlinearVariationalProblem* to solve.
- **nullspace** – an optional *VectorSpaceBasis* (or *MixedVectorSpaceBasis*) spanning the null space of the operator.
- **transpose_nullspace** – as for the nullspace, but used to make the right hand side consistent.
- **near_nullspace** – as for the nullspace, but used to specify the near nullspace (for multigrid solvers).
- **solver_parameters** – Solver parameters to pass to PETSc. This should be a dict mapping PETSc options to values.
- **appctx** – A dictionary containing application context that is passed to the preconditioner if matrix-free.
- **options_prefix** – an optional prefix used to distinguish PETSc options. If not provided a unique prefix will be created. Use this option if you want to pass options to the solver from the command line in addition to through the `solver_parameters` dict.
- **pre_jacobian_callback** – A user-defined function that will be called immediately before Jacobian assembly. This can be used, for example, to update a coefficient function that has a complicated dependence on the unknown solution.

- **post_jacobian_callback** – As above, but called after the Jacobian has been assembled.
- **pre_function_callback** – As above, but called immediately before residual assembly.
- **post_function_callback** – As above, but called immediately after residual assembly.

Example usage of the `solver_parameters` option: to set the nonlinear solver type to just use a linear solver, use

```
{'snes_type': 'ksponly'}
```

PETSc flag options (where the presence of the option means something) should be specified with `None`. For example:

```
{'snes_monitor': None}
```

To use the `pre_jacobian_callback` or `pre_function_callback` functionality, the user-defined function must accept the current solution as a petsc4py Vec. Example usage is given below:

```
def update_diffusivity(current_solution):
    with cursol.dat.vec_wo as v:
        current_solution.copy(v)
    solve(trial*test*dx == dot(grad(cursol), grad(test))*dx, diffusivity)

solver = NonlinearVariationalSolver(problem,
                                    pre_jacobian_callback=update_
↪diffusivity)
```

DEFAULT_KSP_PARAMETERS = {'ksp_rtol': 1e-05, 'ksp_type': 'preonly', 'mat_mumps_icntl_14': 200, 'mat_type': 'aij', 'pc_factor_mat_solver_type': 'mumps', 'pc_type': 'lu'}

DEFAULT_SNES_PARAMETERS = {'snes_linesearch_type': 'basic', 'snes_type': 'newtonls'}

set_transfer_manager(*manager*)

Set the object that manages transfer between grid levels. Typically a *TransferManager* object.

Parameters

manager – Transfer manager, should conform to the TransferManager interface.

Raises

ValueError – if called after the transfer manager is setup.

solve(*bounds=None*)

Solve the variational problem.

Parameters

bounds – Optional bounds on the solution (lower, upper). `lower` and `upper` must both be *Function*s. or *Vector*s.

Note: If bounds are provided the `snes_type` must be set to `vinewtonssls` or `vinewtonrsls`.

4.46 firedrake.vector module

class `firedrake.vector.`**`Vector`**(*x*)

Bases: `object`

Build a *Vector* that wraps a `pyop2.Dat` for Dolfin compatibilty.

Parameters
x – an *`Function`* to wrap or a *`Vector`* to copy. The former shares data, the latter copies data.

`apply`(*action*)

Finalise vector assembly. This is not actually required in Firedrake but is provided for Dolfin compatibility.

`array`()

Return a copy of the process local data as a numpy array

`axpy`(*a*, *x*)

Add a*x to self.

Parameters

- **a** – a scalar
- **x** – a *`Vector`* or *`Function`*

`copy`()

Return a copy of this vector.

`dat`

`gather`(*global_indices=None*)

Gather a *`Vector`* to all processes

Parameters
global_indices – the globally numbered indices to gather (should be the same on all processes). If *None*, gather the entire *`Vector`*.

`get_local`()

Return a copy of the process local data as a numpy array

`inner`(*other*)

Return the l2-inner product of self with other

`local_range`()

Return the global indices of the start and end of the local part of this vector.

`local_size`()

Return the size of the process local data (without ghost points)

max()

Return the maximum entry in the vector.

set_local(*values*)

Set process local values

Parameters
values – a numpy array of values of length *Vector.local_size()*

size()

Return the global size of the data

sum()

Return global sum of vector entries.

firedrake.vector.**as_backend_type**(*tensor*)

Compatibility operation for Dolfin's backend switching operations. This is for Dolfin compatibility only. There is no reason for Firedrake users to ever call this.

4.47 firedrake.version module

firedrake.version.**check**()

4.48 Module contents

CHAPTER FIVE

FUNDING

Firedrake is supported by Imperial College London, the Grantham Institute for Climate Change, the Engineering and Physical Sciences Research Council and the Natural Environment research council.

EPSRC grant EP/M011054/1:
A new simulation and optimisation platform for marine technology.

NERC fellowship NE/K008951/1:
Abstracting the environment: automating geoscientific simulation.

NERC grant NE/K006789/1:
Gung Ho Phase 2 - developing a new UK weather and climate model.

EPSRC grant EP/L000407/1:
Platform: underpinning technologies for finite element simulation.

EPSRC Grant EP/K008730/1:
PAMELA: a Panoramic Approach to the Many-CorE LAnd-sape - from end-user to end-device: a holistic game-changing approach.

Previous support was provided by:

EPSRC Grant EP/I00677X/1:
Multi-layered abstractions for PDEs.

NERC studentship NE/G523512/1:
Multi-layered abstractions for PDEs.

BIBLIOGRAPHY

[BMH+16] Gheorghe-Teodor Bercea, Andrew T. T. McRae, David A. Ham, Lawrence Mitchell, Florian Rathgeber, Luigi Nardi, Fabio Luporini, and Paul H. J. Kelly. A structure-exploiting numbering algorithm for finite elements on extruded meshes, and its performance evaluation in firedrake. *Geoscientific Model Development*, 9(10):3803–3815, 2016. URL: http://arxiv.org/abs/1604.05937, arXiv:1604.05937, doi:10.5194/gmd-9-3803-2016.

[GMHC20] Thomas H. Gibson, Lawrence Mitchell, David A. Ham, and Colin J. Cotter. Slate: extending Firedrake's domain-specific abstraction to hybridized solvers for geoscience and beyond. 2020. URL: https://arxiv.org/abs/1802.00303, arXiv:1802.00303, doi:10.5194/gmd-13-735-2020.

[HMLH18] M. Homolya, L. Mitchell, F. Luporini, and D. Ham. Tsfc: a structure-preserving form compiler. *SIAM Journal on Scientific Computing*, 40(3):C401–C428, 2018. URL: https://doi.org/10.1137/17M1130642, doi:10.1137/17M1130642.

[HH16] Miklós Homolya and David A. Ham. A parallel edge orientation algorithm for quadrilateral meshes. *SIAM Journal on Scientific Computing*, 38(5):S48–S61, 2016. URL: http://arxiv.org/abs/1505.03357, arXiv:1505.03357, doi:10.1137/15M1021325.

[HKH17] Miklós Homolya, Robert C. Kirby, and David A. Ham. Exposing and exploiting structure: optimal code generation for high-order finite element methods. 2017. URL: http://arxiv.org/abs/1711.02473, arXiv:1711.02473.

[KM18] Robert C. Kirby and Lawrence Mitchell. Solver composition across the PDE/linear algebra barrier. *SIAM Journal on Scientific Computing*, 40(1):C76–C98, 2018. URL: http://arxiv.org/abs/1706.01346, arXiv:1706.01346, doi:10.1137/17M1133208.

[LHK17] Fabio Luporini, David A. Ham, and Paul H. J. Kelly. An algorithm for the optimization of finite element integration loops. *ACM Transactions on Mathematical Software*, 44(1):3:1–3:26, 2017. arXiv:1604.05872, doi:10.1145/3054944.

[LVR+15] Fabio Luporini, Ana Lucia Varbanescu, Florian Rathgeber, Gheorghe-Teodor Bercea, J. Ramanujam, David A. Ham, and Paul H. J. Kelly. Cross-loop optimization of arithmetic intensity for finite element local assembly. *ACM Transactions on Architecture and Code Optimization*, 11(4):57:1–57:25, 2015. URL: http://doi.acm.org/10.1145/2687415, doi:10.1145/2687415.

[MBM+16] Andrew T. T. McRae, Gheorghe-Teodor Bercea, Lawrence Mitchell, David A. Ham, and Colin J. Cotter. Automated generation and symbolic manipulation of tensor product finite elements. *SIAM Journal on Scientific Comput-*

ing, 38(5):S25–S47, 2016. URL: http://arxiv.org/abs/1411.2940, arXiv:1411.2940, doi:10.1137/15M1021167.

[RHM+16] Florian Rathgeber, David A. Ham, Lawrence Mitchell, Michael Lange, Fabio Luporini, Andrew T. T. Mcrae, Gheorghe-Teodor Bercea, Graham R. Markall, and Paul H. J. Kelly. Firedrake: automating the finite element method by composing abstractions. *ACM Trans. Math. Softw.*, 43(3):24:1–24:27, 2016. URL: http://arxiv.org/abs/1501.01809, arXiv:1501.01809, doi:10.1145/2998441.

[BGGMuller21] Jack Betteridge, Thomas H. Gibson, Ivan G. Graham, and Eike H. Müller. Multigrid preconditioners for the hybridised discontinuous Galerkin discretisation of the shallow water equations. *Journal of Computational Physics*, 426:109948, 2021. URL: https://www.sciencedirect.com/science/article/pii/S0021999120307221, doi:10.1016/j.jcp.2020.109948.

[BF21] Pablo D. Brubeck and Patrick E. Farrell. A scalable and robust vertex-star relaxation for high-order FEM. 2021. URL: https://arxiv.org/abs/2107.14758, arXiv:2107.14758.

[FKMW21] Patrick E. Farrell, Matthew G. Knepley, Lawrence Mitchell, and Florian Wechsung. PCPATCH: software for the topological construction of multigrid relaxation methods. *ACM Transactions on Mathematical Software*, 47(25):1–22, 2021. URL: https://arxiv.org/abs/1912.08516, arXiv:1912.08516, doi:10.1145/3445791.

[GT09] Jayadeep Gopalakrishnan and Shuguang Tan. A convergent multigrid cycle for the hybridized mixed method. *Numerical Linear Algebra with Applications*, 16(9):689–714, sep 2009. URL: https://doi.org/10.1002/nla.636, doi:10.1002/nla.636.

[LeV96] Randall J. LeVeque. High-Resolution Conservative Algorithms for Advection in Incompressible Flow. *SIAM Journal on Numerical Analysis*, 33(2):627–665, 1996. doi:10.1137/0733033.

[SO88] Chi-Wang Shu and Stanley Osher. Efficient Implementation of Essentially Non-oscillatory Shock-Capturing Schemes. *Journal of Computational Physics*, 77(2):439–471, 1988. doi:10.1016/0021-9991(88)90177-5.

[BK16] O. Bokhove and A. Kalogirou. *Lectures on the Theory of Water Waves*, chapter Variational water wave modelling: from continuum to experiment. LMS Lecture Note Series. Cambridge University Press, 2016. URL: http://www1.maths.leeds.ac.uk/~matak/documents/lms-cup2015.pdf.

[DJ89] P.G. Drazin and R.S. Johnson. *Solitons: an Introduction*. Cambridge University Press, 1989.

[EHW06] C. Lubich E. Hairer and G. Wanner. *Geometric numerical integration*. Springer, 2006.

[GBK17] F. Gidel, O. Bokhove, and A. Kalogirou. Variational modelling of extreme waves through oblique interaction of solitary waves: application to Mach reflection. *Nonlinear Processes in Geophysics*, 24:43–60, 2017. doi:10.5194/npg-24-43-2017.

[QG-Got05] Sigal Gottlieb. On high order strong stability preserving Runge–Kutta and multi step time discretizations. *Journal of Scientific Computing*, 25(1):105–128, 2005. doi:10.1007/s10915-004-4635-5.

[QG-Ped92] Joseph Pedlosky. *Geophysical Fluid Dynamics*. Springer study edition. Springer New York, 1992. ISBN 9780387963877.

[QG-Val06] Geoffrey K. Vallis. *Atmospheric and Oceanic Fluid Dynamics*. Cambridge University Press, Cambridge, U.K., 2006.

[QGeval-Ped92] Joseph Pedlosky. *Geophysical Fluid Dynamics*. Springer study edition. Springer New York, 1992. ISBN 9780387963877.

[QGeval-Val06] Geoffrey K. Vallis. *Atmospheric and Oceanic Fluid Dynamics*. Cambridge University Press, Cambridge, U.K., 2006.

[Mun50] Walter H. Munk. On the wind-driven ocean circulation. *Journal of Meteorology*, 7:79–93, 1950. doi:10.1175/1520-0469(1950)007<0080:OTWDOC>2.0.CO;2.

[Ped92] Joseph Pedlosky. *Geophysical Fluid Dynamics*. Springer study edition. Springer New York, 1992. ISBN 9780387963877.

[Sto48] Henry Stommel. The westward intensifciation of wind driven ocean currents. *Trans. Am. Geophys. Union*, 29:202–206, 1948. doi:10.1007/s10915-004-4635-5.

[Val06] Geoffrey K. Vallis. *Atmospheric and Oceanic Fluid Dynamics*. Cambridge University Press, Cambridge, U.K., 2006.

[BGL05] Michele Benzi, Gene H. Golub, and Jörg Liesen. Numerical solution of saddle point problems. *Acta Numerica*, 14:1–137, 5 2005. doi:10.1017/S0962492904000212.

[ESW14] Howard Elman, David Silvester, and Andy Wathen. *Finite elements and fast iterative solvers*. Oxford University Press, second edition edition, 2014.

[HX07] Ralf Hiptmair and Jinchao Xu. Nodal auxiliary space preconditioning in H(curl) and H(div) spaces. *SIAM Journal on Numerical Analysis*, 45(6):2483–2509, 2007. doi:10.1137/060660588.

[Kir10] Robert C. Kirby. From functional analysis to iterative methods. *SIAM Review*, 52(2):269–293, 2010. doi:10.1137/070706914.

[MGW00] Malcolm F. Murphy, Gene H. Golub, and Andrew J. Wathen. A note on preconditioning for indefinite linear systems. *SIAM Journal on Scientific Computing*, 21(6):1969–1972, 2000. doi:10.1137/S1064827599355153.

[CH93] Roberto Camassa and Darryl D. Holm. An integrable shallow water equation with peaked solitons. *Physical Review Letters*, 71(11):1661, 1993.

[Ise09] Arieh Iserles. *A first course in the numerical analysis of differential equations*. Number 44 in Cambridge Texts in Applied Mathematics. Cambridge University Press, 2009.

[Mat10] Takayasu Matuso. A Hamiltonian-conserving Galerkin scheme for the Camassa-Holm equation. *Journal of Computational and Applied Mathematics*, 234(4):1258–1266, 2010.

[Awa14] Gerard Awanou. Quadratic mixed finite element approximations of the Monge–Ampère equation in 2D. *Calcolo*, pages 1–16, 2014.

[LP13] Omar Lakkis and Tristan Pryer. A finite element method for nonlinear elliptic problems. *SIAM Journal on Scientific Computing*, 35(4):A2025–A2045, 2013.

[SBK16] Tomasz Salwa, Onno Bokhove, and Mark A. Kelmanson. Variational modelling of wave-structure interactions for offshore wind turbines. *Extended paper for Int. Conf. on Ocean, Offshore and Arctic Eng., OMAE2016, Busan, South-Korea*, June 2016. URL: http://proceedings.asmedigitalcollection.asme.org/proceeding.aspx?articleID=2570974.

[SBK17] Tomasz Salwa, Onno Bokhove, and Mark A. Kelmanson. Variational modelling of wave–structure interactions with an offshore wind-turbine mast. *Journal of Engineering Mathematics*, Sep 2017. doi:10.1007/s10665-017-9936-4.

[CJKMVV99] MJS Chin-Joe-Kong, Wim A Mulder, and M Van Veldhuizen. Higher-order triangular and tetrahedral finite elements with mass lumping for solving the wave equation. *Journal of Engineering Mathematics*, 35(4):405–426, 1999. doi:https://doi.org/10.1023/A:1004420829610.

[GMvdV18] Sjoerd Geevers, Wim A Mulder, and Jaap JW van der Vegt. New higher-order mass-lumped tetrahedral elements for wave propagation modelling. *SIAM journal on scientific computing*, 40(5):A2830–A2857, 2018. doi:https://doi.org/10.1137/18M1175549.

PYTHON MODULE INDEX

INDEX