

Solving PDEs with Firedrake: preliminaries

Patrick E. Farrell



University of Oxford

February 2024

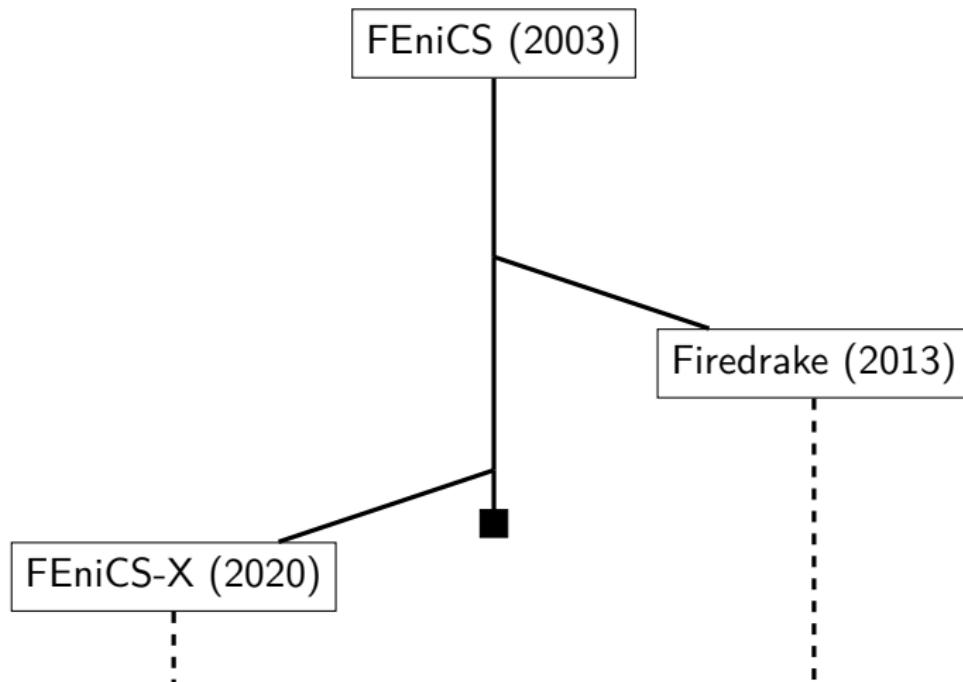
Firedrake is a powerful library for implementing FEM discretisations.

- ▶ Python → efficient C code
- ▶ Began in 2013
- ▶ Thousands of downloads/month
- ▶ Developers in UK, US, AU
- ▶ Licensed under the GNU LGPL

<https://firedrakeproject.org/>



Family tree



Software dependencies

To do the exercises for this course, you need to access Firedrake somehow.

Software dependencies

To do the exercises for this course, you need to access Firedrake somehow.

Your options:

- ▶ Install on Linux/OSX via `firedrake-install` script;

Software dependencies

To do the exercises for this course, you need to access Firedrake somehow.

Your options:

- ▶ Install on Linux/OSX via `firedrake-install` script;
- ▶ Install on Windows via Windows Subsystem for Linux and `firedrake-install` script;

Software dependencies

To do the exercises for this course, you need to access Firedrake somehow.

Your options:

- ▶ Install on Linux/OSX via `firedrake-install` script;
- ▶ Install on Windows via Windows Subsystem for Linux and `firedrake-install` script;
- ▶ Load a Docker image;

Software dependencies

To do the exercises for this course, you need to access Firedrake somehow.

Your options:

- ▶ Install on Linux/OSX via `firedrake-install` script;
- ▶ Install on Windows via Windows Subsystem for Linux and `firedrake-install` script;
- ▶ Load a Docker image;
- ▶ Jupyter notebooks on Google Colab.

Software dependencies

To do the exercises for this course, you need to access Firedrake somehow.

Your options:

- ▶ Install on Linux/OSX via `firedrake-install` script;
- ▶ Install on Windows via Windows Subsystem for Linux and `firedrake-install` script;
- ▶ Load a Docker image;
- ▶ Jupyter notebooks on Google Colab.

If you're not sure, use Google Colab.

To use Google Colab:

- ▶ Visit <https://colab.research.google.com>;

To use Google Colab:

- ▶ Visit <https://colab.research.google.com>;
- ▶ Sign in with your Google account, if necessary;

To use Google Colab:

- ▶ Visit <https://colab.research.google.com>;
- ▶ Sign in with your Google account, if necessary;
- ▶ Visit <https://fem-on-colab.github.io/packages.html>;

To use Google Colab:

- ▶ Visit <https://colab.research.google.com>;
- ▶ Sign in with your Google account, if necessary;
- ▶ Visit <https://fem-on-colab.github.io/packages.html>;
- ▶ Copy & paste the Netgen snippet to the first cell of your notebook;

To use Google Colab:

- ▶ Visit <https://colab.research.google.com>;
- ▶ Sign in with your Google account, if necessary;
- ▶ Visit <https://fem-on-colab.github.io/packages.html>;
- ▶ Copy & paste the Netgen snippet to the first cell of your notebook;
- ▶ Copy & paste the Firedrake snippet to the second cell of your notebook.



+ Code

```
try:
    import netgen
except ImportError:
    !wget "https://fem-on-colab.github.io/releases/ngsolve-install-real.sh" -O "/tmp/ngsolve-install.sh" && bash "/tmp/ngsolve-install.sh"
    import netgen

[ ] try:
    import firedrake
except ImportError:
    !wget "https://fem-on-colab.github.io/releases/firedrake-install-real.sh" -O "/tmp/firedrake-install.sh" && bash "/tmp/firedrake-install.sh"
    import firedrake

[ ] from firedrake import *
from netgen.occ import *

cube = Box(Pnt(0,0,0), Pnt(1,1,1))
sphere = Sphere(Pnt(0.5, 0.5, 0.5), 0.7)

cyl1 = Cylinder(Pnt(-0.5,0.5,0.5), X, r=0.3, h=2.)
cyl2 = Cylinder(Pnt(0.5,-0.5,0.5), Y, r=0.3, h=2.)
cyl3 = Cylinder(Pnt(0.5,0.5,-0.5), Z, r=0.3, h=2.)

shape = (cube * sphere) - (cyl1 + cyl2 + cyl3)
geo = OCCGeometry(shape, dim=3)

ngmesh = geo.GenerateMesh(maxh=0.1)
mesh = Mesh(ngmesh)
File("output/csg3d.pvd").write(mesh)
```

If you install on your own machine, use

```
firedrake-install --netgen --slepc --install irksome --install fascd
```

The other software we will use is ParaView, for visualisation.

The other software we will use is ParaView, for visualisation.

Please download it from <https://www.paraview.org/download>.

Solving PDEs with Firedrake: the Poisson equation

Patrick E. Farrell



University of Oxford

February 2024

A first solver: Poisson

Consider Poisson's equation with Dirichlet boundary conditions:

$$\begin{aligned}-\Delta u &= f \quad \text{in } \Omega, \\ u &= 0 \quad \text{on } \partial\Omega.\end{aligned}$$

Poisson's equation is ubiquitous in physical applications, and often arises as a subproblem of more complex solvers.

From PDE to variational problem

We multiply the PDE by a test function v and integrate over Ω :

$$-\int_{\Omega} (\Delta u)v \, dx = \int_{\Omega} fv \, dx.$$

From PDE to variational problem

We multiply the PDE by a test function v and integrate over Ω :

$$-\int_{\Omega} (\Delta u)v \, dx = \int_{\Omega} fv \, dx.$$

Then integrate by parts and set $v = 0$ on the Dirichlet boundary:

$$-\int_{\Omega} (\Delta u)v \, dx = \int_{\Omega} \nabla u \cdot \nabla v \, dx - \underbrace{\int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds}_{=0}.$$

From PDE to variational problem

We multiply the PDE by a test function v and integrate over Ω :

$$-\int_{\Omega} (\Delta u)v \, dx = \int_{\Omega} fv \, dx.$$

Then integrate by parts and set $v = 0$ on the Dirichlet boundary:

$$-\int_{\Omega} (\Delta u)v \, dx = \int_{\Omega} \nabla u \cdot \nabla v \, dx - \underbrace{\int_{\partial\Omega} \frac{\partial u}{\partial n} v \, ds}_{=0}.$$

In weak form, the equation is: find $u \in H_0^1(\Omega)$ such that

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx = \int_{\Omega} fv \, dx$$

for all $v \in H_0^1(\Omega)$.

From continuous to discrete

We build finite dimensional subspace of $V = H_0^1(\Omega)$:

$$V_h \subset V$$

with a mesh of Ω and finite elements of degree p on each cell.

From continuous to discrete

We build finite dimensional subspace of $V = H_0^1(\Omega)$:

$$V_h \subset V$$

with a mesh of Ω and finite elements of degree p on each cell.

Galerkin projection: find $u_h \in V_h$ such that

$$\int_{\Omega} \nabla u_h \cdot \nabla v_h \, dx = \int_{\Omega} f v_h \, dx$$

for all $v_h \in V_h$.

From continuous to discrete

We build finite dimensional subspace of $V = H_0^1(\Omega)$:

$$V_h \subset V$$

with a mesh of Ω and finite elements of degree p on each cell.

Galerkin projection: find $u_h \in V_h$ such that

$$\int_{\Omega} \nabla u_h \cdot \nabla v_h \, dx = \int_{\Omega} f v_h \, dx$$

for all $v_h \in V_h$.

On choosing a basis for V_h , this gives a linear system to solve.

Error analysis

From standard theory, we know that for smooth $u \in C^\infty(\Omega)$,

$$\|u - u_h\|_{H^1} \leq Ch^p,$$

where

- ▶ C is a constant,
- ▶ h is the maximum cell diameter in the mesh, and
- ▶ p is the polynomial degree.

A test problem

We construct a test problem for which we can easily check the answer. We first choose the exact solution to be

$$u_{\text{ex}}(x, y) = \sin(3x) \exp(x + y)$$

A test problem

We construct a test problem for which we can easily check the answer. We first choose the exact solution to be

$$u_{\text{ex}}(x, y) = \sin(3x) \exp(x + y)$$

We insert this into Poisson's equation:

$$f = -\Delta u_{\text{ex}} = -\operatorname{div} \operatorname{grad} u_{\text{ex}}$$

and use u_{ex} as Dirichlet data on the boundary of Ω .

A test problem

We construct a test problem for which we can easily check the answer. We first choose the exact solution to be

$$u_{\text{ex}}(x, y) = \sin(3x) \exp(x + y)$$

We insert this into Poisson's equation:

$$f = -\Delta u_{\text{ex}} = -\operatorname{div} \operatorname{grad} u_{\text{ex}}$$

and use u_{ex} as Dirichlet data on the boundary of Ω .

This technique is called the *method of manufactured solutions* (MMS).

Poisson solver in Firedrake: implementation

```
from firedrake import *

mesh = UnitSquareMesh(10, 10, quadrilateral=True)

(x, y) = SpatialCoordinate(mesh)
u_ex = sin(3*x) * exp(x + y)
f = -div(grad(u_ex))

V = FunctionSpace(mesh, "Lagrange", 1)

u = Function(V, name="Solution")
v = TestFunction(V)

bc = DirichletBC(V, u_ex, "on_boundary")

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

solve(F == 0, u, bc)
File("output/poisson.pvd").write(u)
```

Step by step: the first line

The first line of our Firedrake programs will usually begin with

```
from firedrake import *
```

which imports everything Firedrake offers into our namespace.

Step by step: creating a mesh

Next, we create a mesh of our domain $\Omega = (0, 1)^2$:

```
mesh = UnitSquareMesh(10, 10, quadrilateral=True)
```

defines a quadrilateral mesh with 10 elements along each edge.

Step by step: creating a mesh

Next, we create a mesh of our domain $\Omega = (0, 1)^2$:

```
mesh = UnitSquareMesh(10, 10, quadrilateral=True)
```

defines a quadrilateral mesh with 10 elements along each edge.

Other useful classes for creating meshes include `UnitIntervalMesh`, `UnitDiskMesh`, `UnitCubeMesh`, `UnitCubedSphereMesh`, `RectangleMesh`, `BoxMesh`, `PeriodicBoxMesh`, `AnnulusMesh`. See `firedrake.utility_meshes._all_`.

Step by step: creating a mesh

Next, we create a mesh of our domain $\Omega = (0, 1)^2$:

```
mesh = UnitSquareMesh(10, 10, quadrilateral=True)
```

defines a quadrilateral mesh with 10 elements along each edge.

Other useful classes for creating meshes include `UnitIntervalMesh`, `UnitDiskMesh`, `UnitCubeMesh`, `UnitCubedSphereMesh`, `RectangleMesh`, `BoxMesh`, `PeriodicBoxMesh`, `AnnulusMesh`. See `firedrake.utility_meshes._all_`.

Complex geometries can be built in dedicated mesh generation tools:

```
mesh = Mesh("domain.msh")
```

where `domain.msh` is in Gmsh format.

Step by step: creating a function space

The following line creates our finite element space on Ω :

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

The second argument specifies the *family* of element, while the third argument is the *polynomial degree* p .

Step by step: creating a function space

The following line creates our finite element space on Ω :

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

The second argument specifies the *family* of element, while the third argument is the *polynomial degree* p .

Some available families include

- ▶ "Discontinuous Lagrange"
- ▶ "Brezzi-Douglas-Marini"
- ▶ "Raviart-Thomas"
- ▶ "Crouzeix-Raviart"
- ▶ "Nedelec 1st kind
- ▶ H(curl)"
- ▶ "Argyris"
- ▶ "Hermite"
- ▶ "Arnold-Winther"
- ▶ "Hellan-Herrmann-Johnson"
- ▶ "Serendipity"

Step by step: defining expressions

Next, we define a symbolic expression for the true solution:

```
(x, y) = SpatialCoordinate(mesh)
u_ex = sin(3*x) * exp(x + y)
```

Step by step: defining expressions

Next, we define a symbolic expression for the true solution:

```
(x, y) = SpatialCoordinate(mesh)
u_ex = sin(3*x) * exp(x + y)
```

We then use it to calculate the source term:

```
f = -div(grad(u_ex))
```

Step by step: defining a boundary condition

The following code defines a Dirichlet boundary condition:

```
bc = DirichletBC(V, u_ex, "on_boundary")
```

This boundary condition states that a function in the function space defined by V should be equal to u_{ex} on the interface associated with "on_boundary" (which is all of $\partial\Omega$).

Note that the above line does not yet apply the boundary condition to all functions in the function space. (It gets enforced strongly during `solve`.)

Step by step: more about defining domains

For finer control, you can do

```
DirichletBC(V, u_ex, (1, 2, 3, 4))
```

where in the UnitSquareMesh docstring we learn

""" The boundary edges in this mesh are numbered as follows:

- * 1: plane $x == 0$
- * 2: plane $x == 1$
- * 3: plane $y == 0$
- * 4: plane $y == 1$ """

Step by step: more about defining domains

For finer control, you can do

```
DirichletBC(V, u_ex, (1, 2, 3, 4))
```

where in the UnitSquareMesh docstring we learn

""" The boundary edges in this mesh are numbered as follows:

- * 1: plane $x == 0$
- * 2: plane $x == 1$
- * 3: plane $y == 0$
- * 4: plane $y == 1$ """

When you make your own meshes, you colour the boundaries yourself.

Step by step: solution and test function

Variational problems are defined in terms of *solution* and *test* functions:

```
u = Function(V, name="Solution")
v = TestFunction(V)
```

The `name` argument is used e.g. in visualisation.

Step by step: defining the variational problem

We now have all the objects we need in order to specify the variational statement of the problem.

Step by step: defining the variational problem

We now have all the objects we need in order to specify the variational statement of the problem.

Recall that our problem is: find $u_h \in V_h$ such that

$$\int_{\Omega} \nabla u_h \cdot \nabla v_h \, dx - \int_{\Omega} f v_h \, dx = 0$$

for all $v_h \in V_h$.

Step by step: defining the variational problem

We now have all the objects we need in order to specify the variational statement of the problem.

Recall that our problem is: find $u_h \in V_h$ such that

$$\int_{\Omega} \nabla u_h \cdot \nabla v_h \, dx - \int_{\Omega} f v_h \, dx = 0$$

for all $v_h \in V_h$.

In code this becomes

```
F = inner(grad(u), grad(v))*dx - f*v*dx
```

Here `dx` is a type of class `Measure` that means "integrate over the whole volume". There are other measures: for example, `ds` means "integrate over exterior facets".

Step by step: solving variational problems

Once a variational problem has been defined, it may be solved by calling the `solve` function:

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

Nice!

Step by step: visualisation

For postprocessing in paraview, store the solution in PVD format:

```
File("poisson.pvd").write(u)
```

Step by step: visualisation

For postprocessing in paraview, store the solution in PVD format:

```
File("poisson.pvd").write(u)
```

This is *lossy*: VTK doesn't understand general finite element spaces. For lossless checkpointing, do

```
with CheckpointFile("checkpoint.h5", "w") as cfile:  
    cfile.save_mesh(mesh)  
    cfile.save_function(u)
```

Poisson solver in Firedrake: implementation

```
from firedrake import *

mesh = UnitSquareMesh(10, 10, quadrilateral=True)

(x, y) = SpatialCoordinate(mesh)
u_ex = sin(3*x) * exp(x + y)
f = -div(grad(u_ex))

V = FunctionSpace(mesh, "Lagrange", 1)

u = Function(V, name="Solution")
v = TestFunction(V)

bc = DirichletBC(V, u_ex, "on_boundary")

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

solve(F == 0, u, bc)
File("output/poisson.pvd").write(u)
```

Challenge!

QI.1. Run the code and look at the solution in paraview.

QI.2. Compute the norm of the error with

```
print(f"||u - u_ex||_H1: {norm(u - u_ex, 'H1')}")
```

QI.3. Refine the mesh with

```
base = UnitSquareMesh(10, 10, quadrilateral=True)
mh = MeshHierarchy(base, 1) # 1 refinement
mesh = mh[-1] # take the finest mesh in the hierarchy
```

How does the error change as you keep refining the mesh?

Section 2

Geometric multigrid

Solvers

By default, Firedrake solves linear variational problems with a sparse LU factorisation via MUMPS.

Solvers

By default, Firedrake solves linear variational problems with a sparse LU factorisation via MUMPS.

This is robust, but scales badly for larger problems.

Solvers

By default, Firedrake solves linear variational problems with a sparse LU factorisation via MUMPS.

This is robust, but scales badly for larger problems.

With the right solver algorithm, you can solve *much* bigger problems. But the right choice is problem-specific.

Solvers

By default, Firedrake solves linear variational problems with a sparse LU factorisation via MUMPS.

This is robust, but scales badly for larger problems.

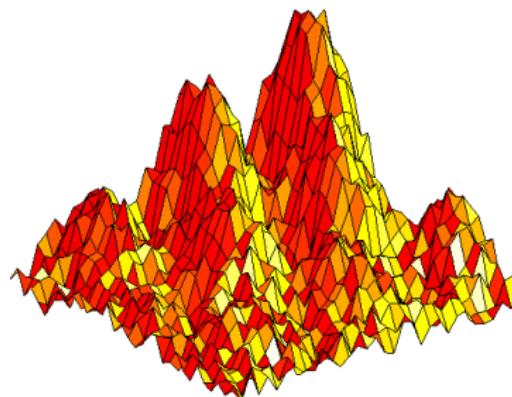
With the right solver algorithm, you can solve *much* bigger problems. But the right choice is problem-specific.

For this coercive, symmetric problem, an excellent choice is conjugate gradients preconditioned by geometric multigrid.

Basic idea of multigrid

Multigrid algorithm

- ▶ Begin with an initial guess.

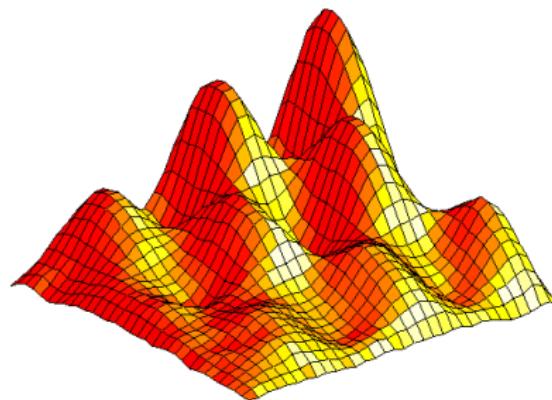


Error of initial guess.

Basic idea of multigrid

Multigrid algorithm

- ▶ Begin with an initial guess.
- ▶ Apply a *relaxation method* to smooth the error.

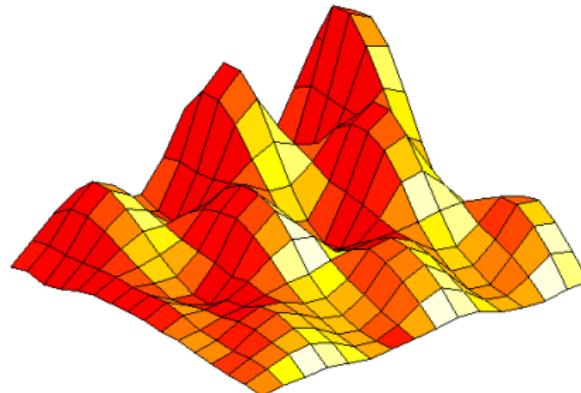


Error after relaxation.

Basic idea of multigrid

Multigrid algorithm

- ▶ Begin with an initial guess.
- ▶ Apply a *relaxation method* to smooth the error.
- ▶ Approximate the smooth error on a *coarse space*.

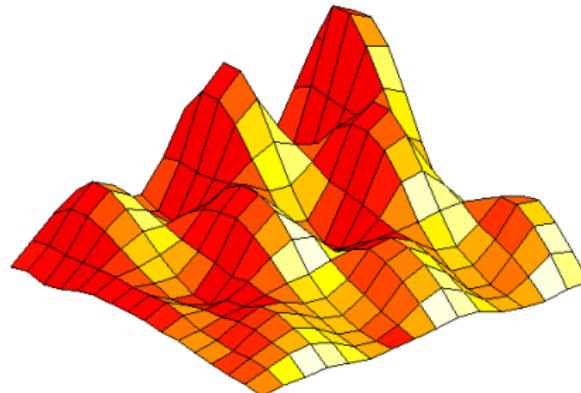


Error approximated on coarse grid.

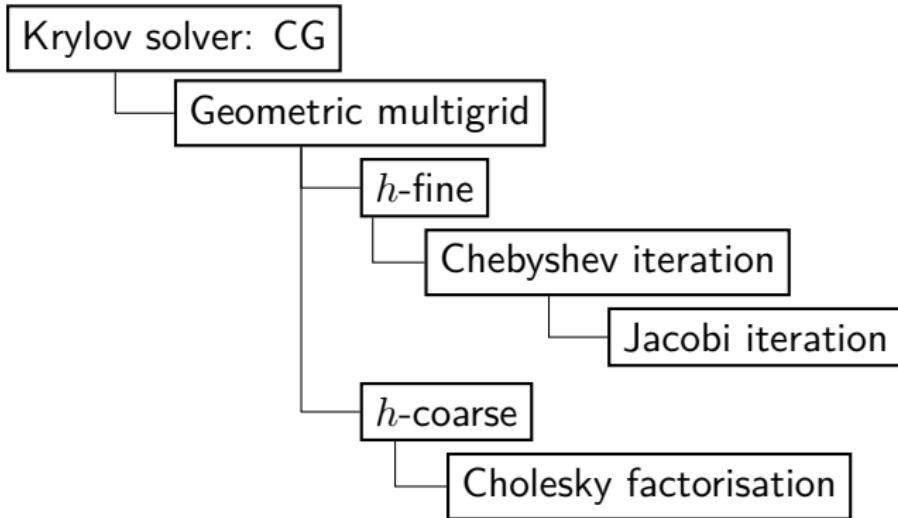
Basic idea of multigrid

Multigrid algorithm

- ▶ Begin with an initial guess.
- ▶ Apply a *relaxation method* to smooth the error.
- ▶ Approximate the smooth error on a *coarse space*.
- ▶ *Prolong* the error approximation to the fine grid and subtract.



Error approximated on coarse grid.



Solver diagram for typical CG + MG solver.

Step by step: specifying solvers

To use geometric multigrid, we again employ a *mesh hierarchy*:

```
base = UnitSquareMesh(10, 10, quadrilateral=True)
mh = MeshHierarchy(base, 2) # 2 refinements
mesh = mh[-1] # take the finest mesh in the hierarchy
# Build problem on mesh ...
```

Step by step: specifying solvers

To use geometric multigrid, we again employ a *mesh hierarchy*:

```
base = UnitSquareMesh(10, 10, quadrilateral=True)
mh = MeshHierarchy(base, 2) # 2 refinements
mesh = mh[-1] # take the finest mesh in the hierarchy
# Build problem on mesh ...
```

We can then tell Firedrake to use CG + MG with

```
sp = {
    "ksp_type": "cg",      # use conjugate gradients
    "ksp_monitor": None,   # print info about iteration
    "ksp_rtol": 1.0e-10,   # residual relative tolerance
    "pc_type": "mg",       # use geometric multigrid
}
solve(F == 0, u, bc, solver_parameters=sp)
```

How this works

$$Ax = b$$



How this works

$$Ax = b$$



1. Firedrake makes a context object that stores all symbolic information about the problem, and associates it with A .

How this works

$$Ax = b$$



1. Firedrake makes a context object that stores all symbolic information about the problem, and associates it with A .
2. Firedrake asks PETSc to use its geometric multigrid preconditioner, PCMG.

How this works

$$Ax = b$$



1. Firedrake makes a context object that stores all symbolic information about the problem, and associates it with A .
2. Firedrake asks PETSc to use its geometric multigrid preconditioner, PCMG.
3. PCMG calls a function on the context object to coarsen the problem.

How this works

$$Ax = b$$



1. Firedrake makes a context object that stores all symbolic information about the problem, and associates it with A .
2. Firedrake asks PETSc to use its geometric multigrid preconditioner, PCMG.
3. PCMG calls a function on the context object to coarsen the problem.
4. Firedrake gets the next coarser mesh from the MeshHierarchy, interpolates any coefficients, and redefines a new variational problem.

How this works

$$Ax = b$$



1. Firedrake makes a context object that stores all symbolic information about the problem, and associates it with A .
2. Firedrake asks PETSc to use its geometric multigrid preconditioner, PCMG.
3. PCMG calls a function on the context object to coarsen the problem.
4. Firedrake gets the next coarser mesh from the MeshHierarchy, interpolates any coefficients, and redefines a new variational problem.
5. Firedrake returns the assembly of this coarser problem.

How this works

$$Ax = b$$



1. Firedrake makes a context object that stores all symbolic information about the problem, and associates it with A .
2. Firedrake asks PETSc to use its geometric multigrid preconditioner, PCMG.
3. PCMG calls a function on the context object to coarsen the problem.
4. Firedrake gets the next coarser mesh from the MeshHierarchy, interpolates any coefficients, and redefines a new variational problem.
5. Firedrake returns the assembly of this coarser problem.
6. PCMG then manages the multigrid cycle, calling Firedrake to transfer information between grids.

Challenge!

QI.4. Change the code to use geometric multigrid. How do the conjugate gradient iterations vary as you take more refinements?

QI.5. Now that we have a good solver algorithm, we can go to 3D.
For example, use

```
base = UnitCubeMesh(5, 5, 5, hexahedral=True)
```

and change the exact solution to be 3D also. If you have installed Firedrake you can (probably) run in parallel with

```
mpiexec -n 8 python myscript.py
```

QI.6. Visualise the 3D solution in paraview.

Section 3

Higher order

Higher order: approximation power

Recall that the standard error theory tells us that for smooth solutions,

$$\|u - u_h\|_{H^1} \leq Ch^p.$$

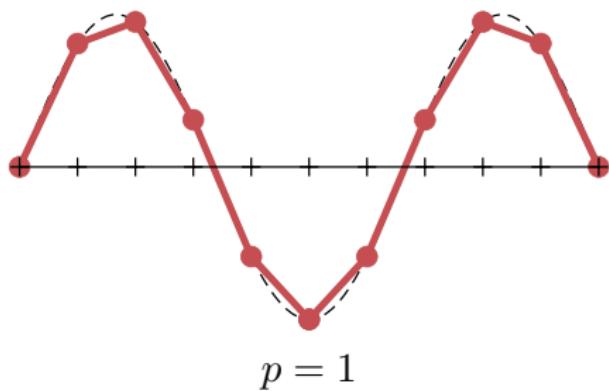
For smooth solutions, we would much rather increase p than decrease h .

Higher order: approximation power

Recall that the standard error theory tells us that for smooth solutions,

$$\|u - u_h\|_{H^1} \leq Ch^p.$$

For smooth solutions, we would much rather increase p than decrease h .

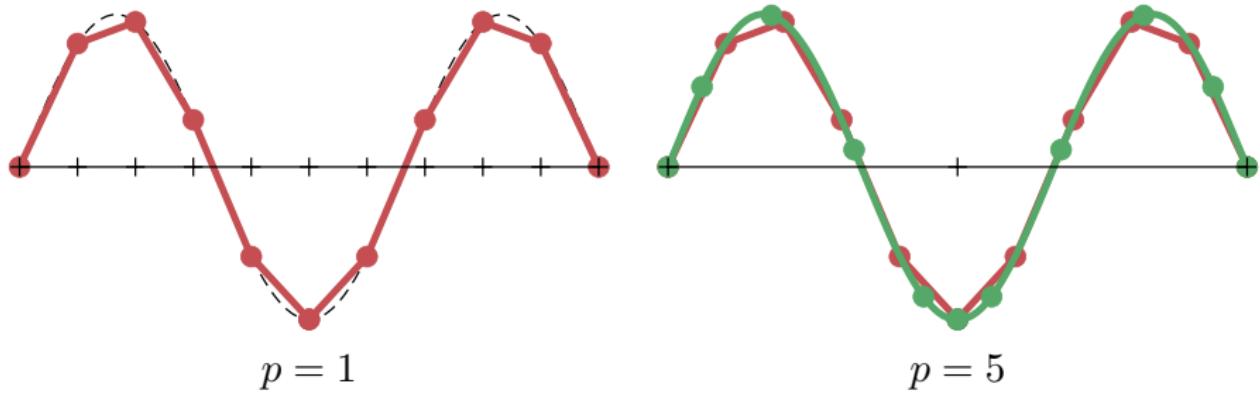


Higher order: approximation power

Recall that the standard error theory tells us that for smooth solutions,

$$\|u - u_h\|_{H^1} \leq Ch^p.$$

For smooth solutions, we would much rather increase p than decrease h .



Going to high order: practicalities

There are many points to consider when going high order.

Going to high order: practicalities

There are many points to consider when going high order.

One of the main ones is solvers. Assembling a matrix takes $\mathcal{O}(p^{2d+1})$ operations.

Going to high order: practicalities

There are many points to consider when going high order.

One of the main ones is solvers. Assembling a matrix takes $\mathcal{O}(p^{2d+1})$ operations.

By contrast, on tensor product cells, you can compute matrix-vector products with $\mathcal{O}(p^{d+1})$ operations.

Going to high order: practicalities

There are many points to consider when going high order.

One of the main ones is solvers. Assembling a matrix takes $\mathcal{O}(p^{2d+1})$ operations.

By contrast, on tensor product cells, you can compute matrix-vector products with $\mathcal{O}(p^{d+1})$ operations.

This strongly motivates matrix-free solvers!

Going to high order: practicalities

There are many points to consider when going high order.

One of the main ones is solvers. Assembling a matrix takes $\mathcal{O}(p^{2d+1})$ operations.

By contrast, on tensor product cells, you can compute matrix-vector products with $\mathcal{O}(p^{d+1})$ operations.

This strongly motivates matrix-free solvers!

We also want to make sure that our solver convergence is p -robust.

p -multigrid offers an excellent solver at high order.



Luca Pavarino

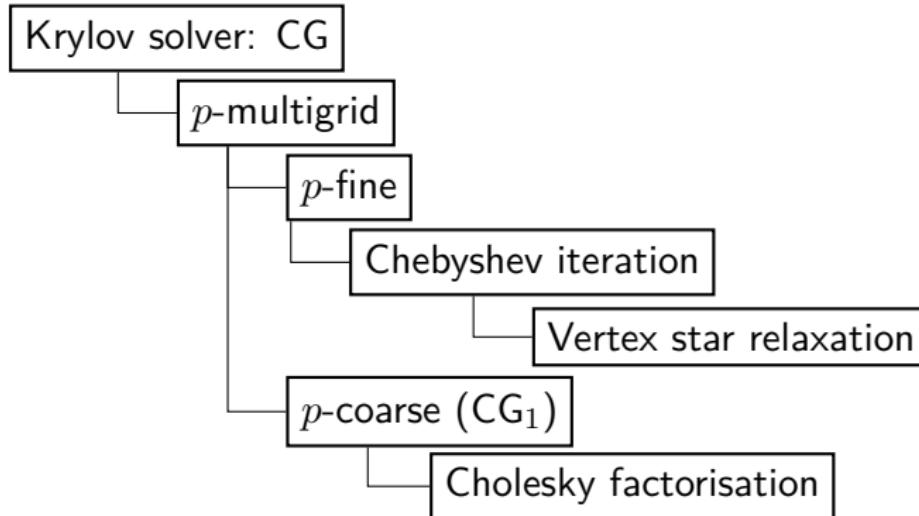


Pablo Brubeck

p -multigrid offers an excellent solver at high order.



Luca Pavarino

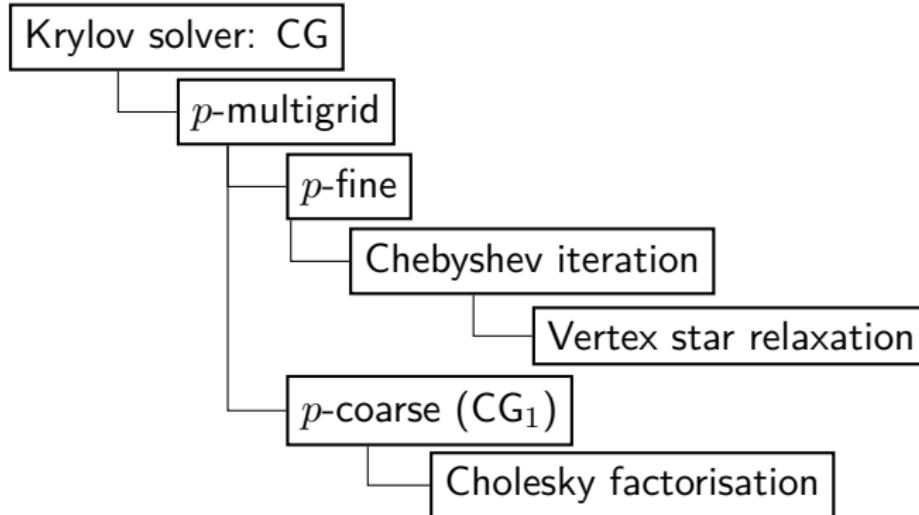


Solver diagram for p -robust CG + p -MG solver.



Pablo Brubeck

p -multigrid offers an excellent solver at high order.



Solver diagram for p -robust CG + p -MG solver.

Luca Pavarino proved this solver was p -robust. Pablo Brubeck figured out how to implement it with $\mathcal{O}(p^{d+1})$ operations.



Luca Pavarino



Pablo Brubeck

```
sp = {
    "mat_type": "matfree",
    "ksp_type": "cg",
    "ksp_monitor": None,
    "ksp_rtol": 1.0e-10,
    "pc_type": "python",
    "pc_python_type": "firedrake.P1PC",
    "pmg_mg_coarse": {
        "pc_type": "python",
        "pc_python_type": "firedrake.AssembledPC",
        "assembled_pc_type": "cholesky",
    },
    "pmg_mg_levels": {
        "ksp_max_it": 1,
        "ksp_type": "chebyshev",
        "pc_type": "python",
        "pc_python_type": "firedrake.FDMPC",
        "fdm": {
            "pc_type": "python",
            "pc_python_type": "firedrake.ASMExtrudedStarPC",
            "pc_star_mat_ordering_type": "metisnd",
            "pc_star_sub_sub_pc_type": "cholesky",
        }
    }
}
```

Challenge!

QI.7. Change the code to use a high order discretisation.

Use the same 2D exact solution. Use the mesh

```
mesh = UnitSquareMesh(5, 5, quadrilateral=True)
```

QI.8. What p is required to achieve a H^1 error of less than 10^{-10} ? How many degrees of freedom does this employ? Use

```
dofs = V.dim()
```

to extract the number of degrees of freedom of a FunctionSpace V .

QI.9. Estimate what mesh resolution would be required to achieve the same error with $p = 1$. How many degrees of freedom would that take? (Do not attempt to compute with this resolution!)

Solving PDEs with Firedrake: meshes and meshing

Patrick E. Farrell



University of Oxford

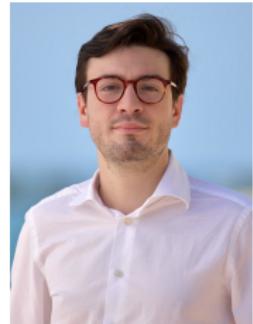
February 2024

There are three main ways of making meshes for Firedrake:

1. Use built-in classes (`BoxMesh`, `RectangleMesh`, etc.);
2. Use a dedicated external mesh generator (`gmsh` etc.) and read in from a file;
3. Use Netgen, an integrated package for constructive solid geometry.



Joachim Schöberl

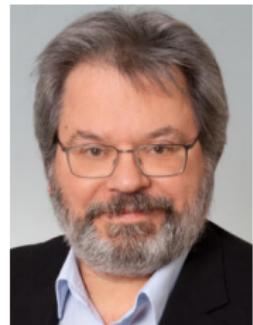


Umberto Zerbinati

There are three main ways of making meshes for Firedrake:

1. Use built-in classes (`BoxMesh`, `RectangleMesh`, etc.);
2. Use a dedicated external mesh generator (`gmsh` etc.) and read in from a file;
3. Use Netgen, an integrated package for constructive solid geometry.

As we will see, using Netgen has many advantages.



Joachim Schöberl



Umberto Zerbinati

There are three main ways of making meshes for Firedrake:

1. Use built-in classes (`BoxMesh`, `RectangleMesh`, etc.);
2. Use a dedicated external mesh generator (`gmsh` etc.) and read in from a file;
3. Use Netgen, an integrated package for constructive solid geometry.

As we will see, using Netgen has many advantages.

Netgen is developed by Joachim Schöberl. Its Firedrake integration is developed by Umberto Zerbinati.



Joachim Schöberl



Umberto Zerbinati

Constructive solid geometry

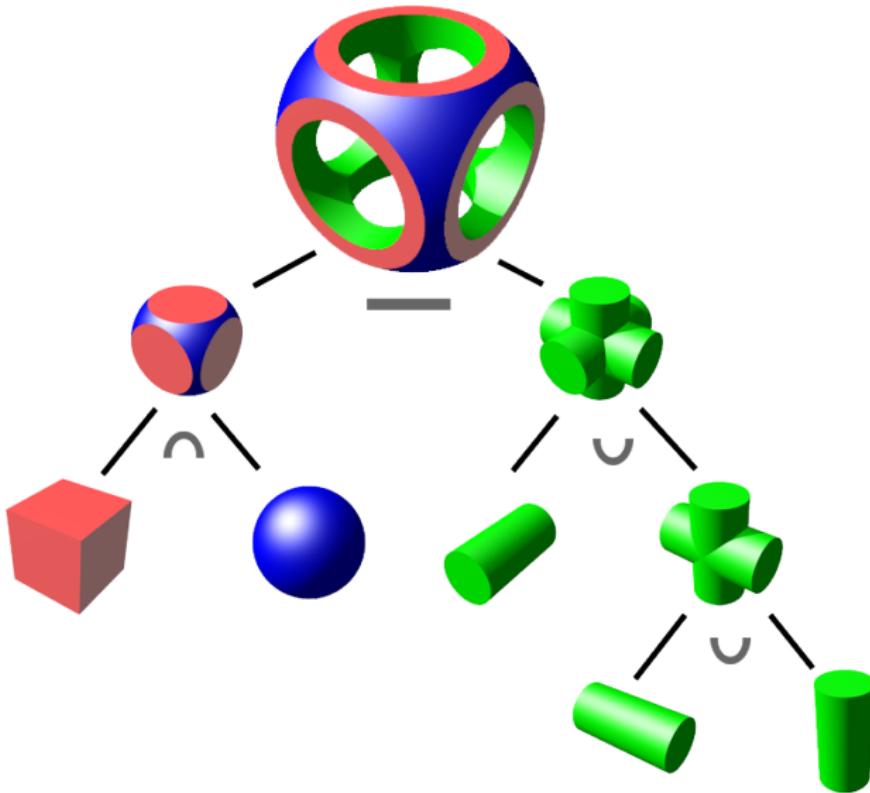
Constructive solid geometry expresses a domain with two ingredients:

Geometric primitives:

- ▶ rectangle/box
- ▶ circle/sphere
- ▶ cylinder/cone
- ▶ polygon/polyhedron
- ▶ ...

Operations on those primitives:

- ▶ union
- ▶ intersection
- ▶ set difference
- ▶ rotation
- ▶ scaling
- ▶ translation



Constructive solid geometry. Credit: Zollie/Wikipedia

Implementation in Netgen/Firedrake

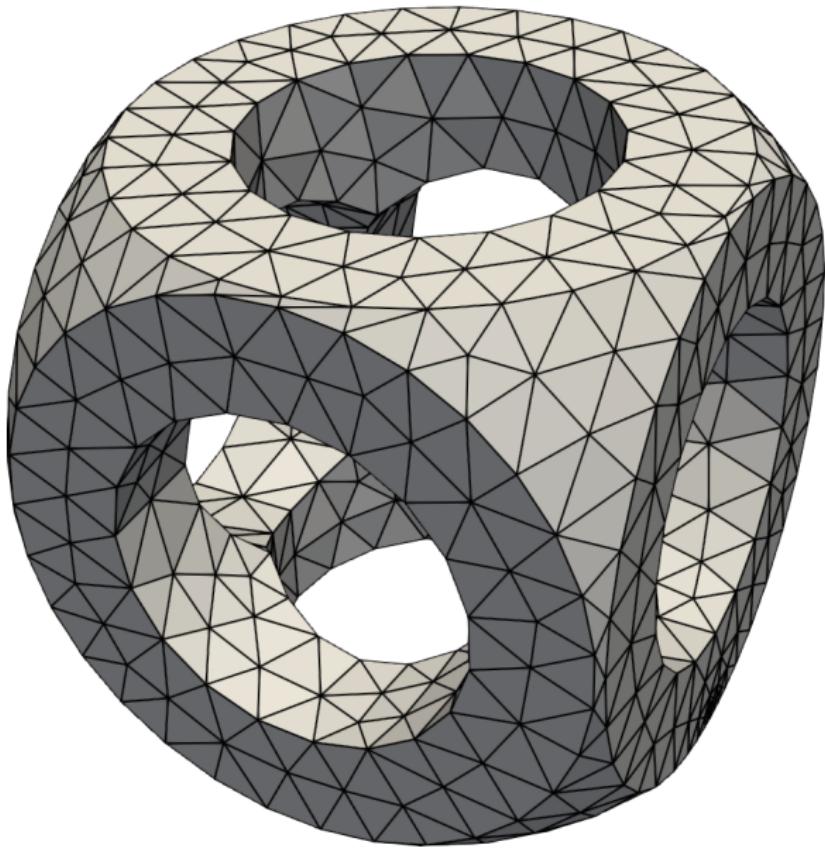
```
from firedrake import *
from netgen.occ import *

cube = Box(Pnt(0,0,0), Pnt(1,1,1))
sphere = Sphere(Pnt(0.5, 0.5, 0.5), 0.7)

cyl1 = Cylinder(Pnt(-0.5,0.5,0.5), X, r=0.3, h=2.)
cyl2 = Cylinder(Pnt(0.5,-0.5,0.5), Y, r=0.3, h=2.)
cyl3 = Cylinder(Pnt(0.5,0.5,-0.5), Z, r=0.3, h=2.)

shape = (cube * sphere) - (cyl1 + cyl2 + cyl3)
geo = OCCGeometry(shape, dim=3)

ngmesh = geo.GenerateMesh(maxh=0.1)
mesh = Mesh(ngmesh)
File("output/csg3d.pvd").write(mesh)
```



L-shaped domain

We can do this in 2D also.

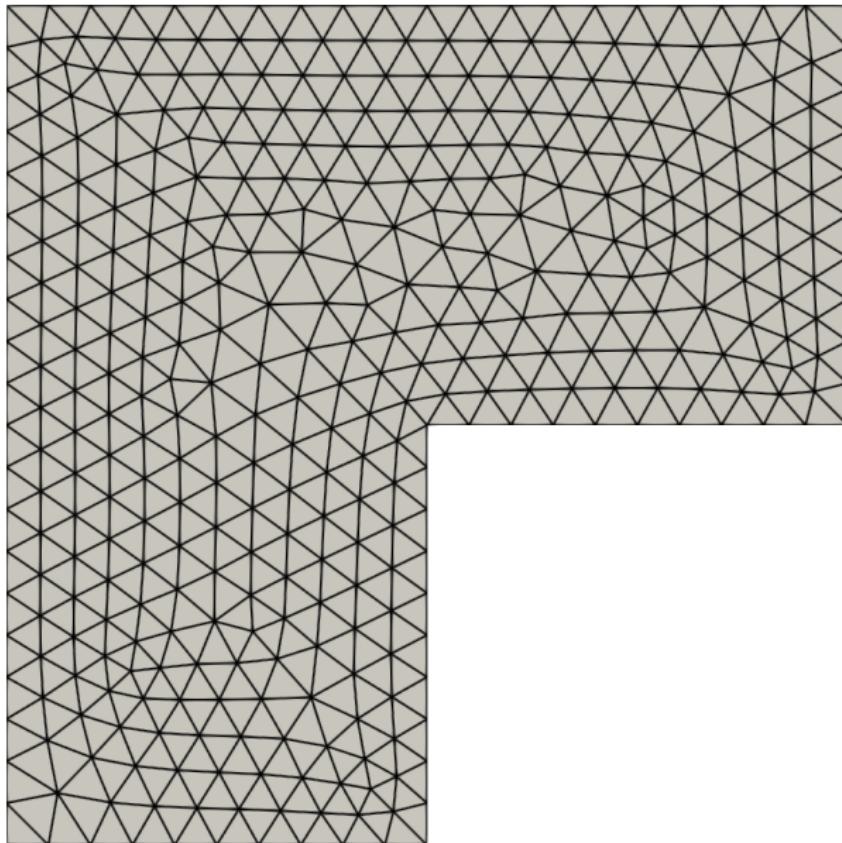
```
from firedrake import *
from netgen.occ import *

# Make 2D rectangle from (0, 0) to (1, 2)
rect1 = WorkPlane(Axes((0,0,0), n=Z, h=X)).Rectangle(1,2).Face()

# Make 2D rectangle from (0, 1) to (2, 2)
rect2 = WorkPlane(Axes((0,1,0), n=Z, h=X)).Rectangle(2,1).Face()
L = rect1 + rect2

geo = OCCGeometry(L, dim=2)
ngmesh = geo.GenerateMesh(maxh=0.1)
mesh = Mesh(ngmesh)

File("output/lshaped.pvd").write(mesh)
```



Challenge!

QII.1. Solve the Poisson equation on the L-shaped domain, with homogeneous Dirichlet conditions and $f = 1$. Represent this with

```
f = Constant(1)
```

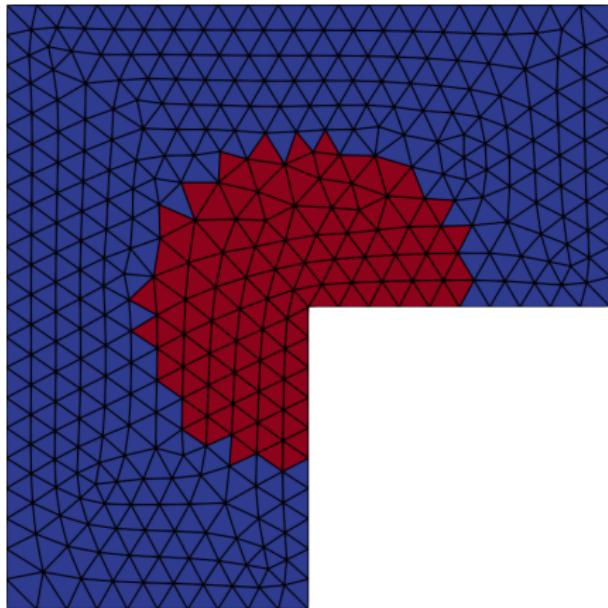
QII.2. What is the key mathematical difference between solving the Poisson equation on an L-shaped domain, compared to solving on a unit square?

Section 2

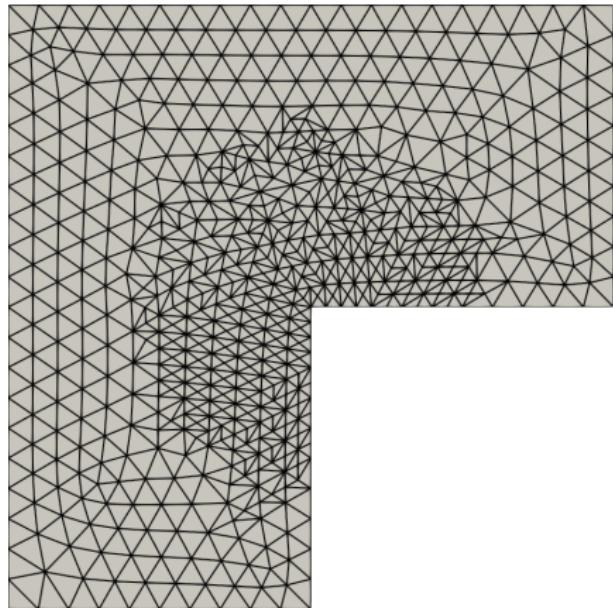
Adaptive mesh refinement

When the mesh is built with Netgen, Firedrake can do adaptive refinement.

```
# Simple criterion for refinement:  
# is the cell close to the singular corner at (1, 1)?  
(x, y) = SpatialCoordinate(mesh)  
r_squared = (x - 1)**2 + (y - 1)**2  
  
# conditional(condition, true_value, false_value)  
should_refine = conditional(lt(r_squared, 0.3), 1, 0)  
  
# Make a function that stores one number for each cell  
DG0 = FunctionSpace(mesh, "DG", 0)  
markers = Function(DG0)  
markers.interpolate(should_refine)  
  
refined_mesh = mesh.refine_marked_elements(markers)
```



Markers



Refined mesh

We can use an *a posteriori error estimator* to guide refinement.

We can use an *a posteriori* error estimator to guide refinement.

For the Poisson equation

$$-\nabla^2 u = f \text{ in } \Omega, \quad u = 0 \text{ on } \partial\Omega,$$

a standard Babuška–Rheinboldt residual error estimator is

$$\eta_K^2 = h_K^2 \int_K |f + \nabla^2 u_h|^2 \, dx + \frac{h_K}{2} \int_{\partial K \setminus \partial\Omega} [\![\nabla u_h \cdot n]\!]^2 \, ds,$$

where

- ▶ K is a mesh cell,
- ▶ h_K is its diameter,
- ▶ n is the outward-facing unit normal on ∂K , and
- ▶ $[\![\cdot]\!]$ is the jump operator.

To compute this, we will solve a variational problem:

find $\eta \in \text{DG}_0(\text{mesh})$ such that

$$\begin{aligned} \int_{\Omega} \eta^2 w \, dx &= \int_{\Omega} \sum_K \int_K h_K^2 (f + \operatorname{div} \operatorname{grad} u_h)^2 \, dx w \, dx \\ &\quad + \int_{\Omega} \sum_K \int_{\partial K \setminus \partial \Omega} \frac{h_K}{2} [\![\nabla u \cdot n]\!]^2 \, ds w \, dx, \end{aligned}$$

for all $w \in \text{DG}_0(\text{mesh})$.

To compute this, we will solve a variational problem:

find $\eta \in \text{DG}_0(\text{mesh})$ such that

$$\begin{aligned} \int_{\Omega} \eta^2 w \, dx &= \int_{\Omega} \sum_K \int_K h_K^2 (f + \operatorname{div} \operatorname{grad} u_h)^2 \, dx w \, dx \\ &\quad + \int_{\Omega} \sum_K \int_{\partial K \setminus \partial \Omega} \frac{h_K}{2} [\![\nabla u \cdot n]\!]^2 \, ds w \, dx, \end{aligned}$$

for all $w \in \text{DG}_0(\text{mesh})$.

This is just a trick to get Firedrake to compute the error estimator with compiled code, rather than looping over the cells ourselves.

To compute this, we will solve a variational problem:

find $\eta \in \text{DG}_0(\text{mesh})$ such that

$$\int_{\Omega} \eta^2 w \, dx = \int_{\Omega} \sum_K \int_K h_K^2 (f + \operatorname{div} \operatorname{grad} u_h)^2 \, dx w \, dx \\ + \int_{\Omega} \sum_K \int_{\partial K \setminus \partial \Omega} \frac{h_K}{2} [\![\nabla u \cdot n]\!]^2 \, ds w \, dx,$$

for all $w \in \text{DG}_0(\text{mesh})$.

This is just a trick to get Firedrake to compute the error estimator with compiled code, rather than looping over the cells ourselves.

We can simplify the expression by swapping the integrals and using

$$\int_K \text{const} \, dx = \text{vol}(K) \text{const.}$$

```

def estimate_error(mesh, uh):
    W = FunctionSpace(mesh, "DG", 0)
    eta_sq = Function(W)
    w = TestFunction(W)
    f = Constant(1)
    h = CellDiameter(mesh) # symbols for mesh quantities
    n = FacetNormal(mesh)
    v = CellVolume(mesh)

    # Compute error indicator cellwise
    G = (
        inner(eta_sq / v, w)*dx
        - inner(h**2 * (f + div(grad(uh)))**2, w) * dx
        - inner(h('+')/2 * jump(grad(uh), n)**2, w('+')) * dS
    )

    # Each cell is an independent 1x1 solve, so Jacobi is exact
    sp = {"mat_type": "matfree", "ksp_type": "richardson", "pc_type": "jacobi"}
    solve(G == 0, eta_sq, solver_parameters=sp)
    eta = Function(W)
    eta.interpolate(sqrt(eta_sq)) # the above computed eta^2

    with eta.dat.vec_ro as eta_:
        # compute estimate for error in energy norm
        error_est = sqrt(eta_.dot(eta_))
    return (eta, error_est)

```

With the error indicator in hand, we need to decide how to refine.



Willy Dörfler

With the error indicator in hand, we need to decide how to refine.

The best known strategy is *Dörfler marking*, which provably converges at the optimal rate.



Willy Dörfler

With the error indicator in hand, we need to decide how to refine.

The best known strategy is *Dörfler marking*, which provably converges at the optimal rate.

We will implement a slightly simpler strategy, where a cell K is marked for refinement if



Willy Dörfler

$$\eta_K \geq \theta \max_L \eta_L,$$

where $\theta \in [0, 1]$. This requires less parallel communication.

With the error indicator in hand, we need to decide how to refine.

The best known strategy is *Dörfler marking*, which provably converges at the optimal rate.

We will implement a slightly simpler strategy, where a cell K is marked for refinement if



Willy Dörfler

$$\eta_K \geq \theta \max_L \eta_L,$$

where $\theta \in [0, 1]$. This requires less parallel communication.

This has no proof but offers similar performance in practice.

```
def adapt(mesh, eta):
    W = FunctionSpace(mesh, "DG", 0)
    markers = Function(W)

    # We decide to refine an element if its error indicator
    # is within a fraction of the maximum cellwise error indicator

    # Access storage underlying our Function
    # (a PETSc Vec) to get maximum value of eta
    with eta.dat.vec_ro as eta_:
        eta_max = eta_.max()[1]

    theta = 0.5
    should_refine = conditional(gt(eta, theta*eta_max), 1, 0)
    markers.interpolate(should_refine)

    refined_mesh = mesh.refine_marked_elements(markers)
    return refined_mesh
```

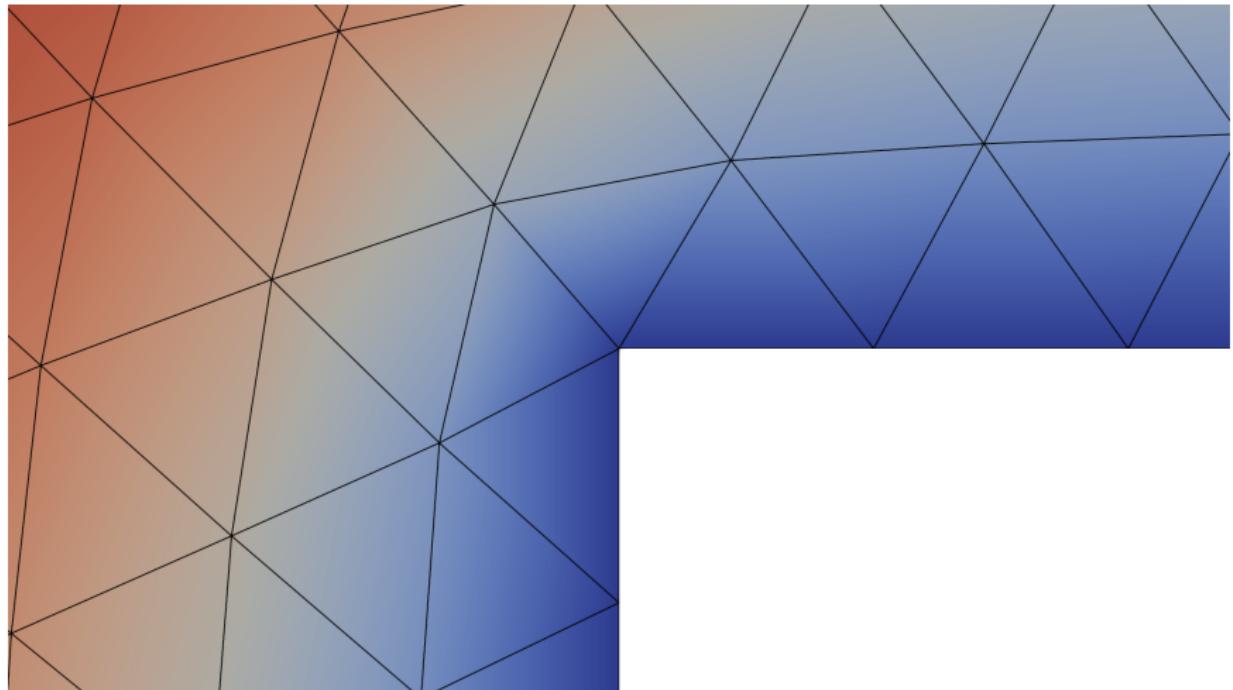
We can now write our main loop to drive the adaptive process.

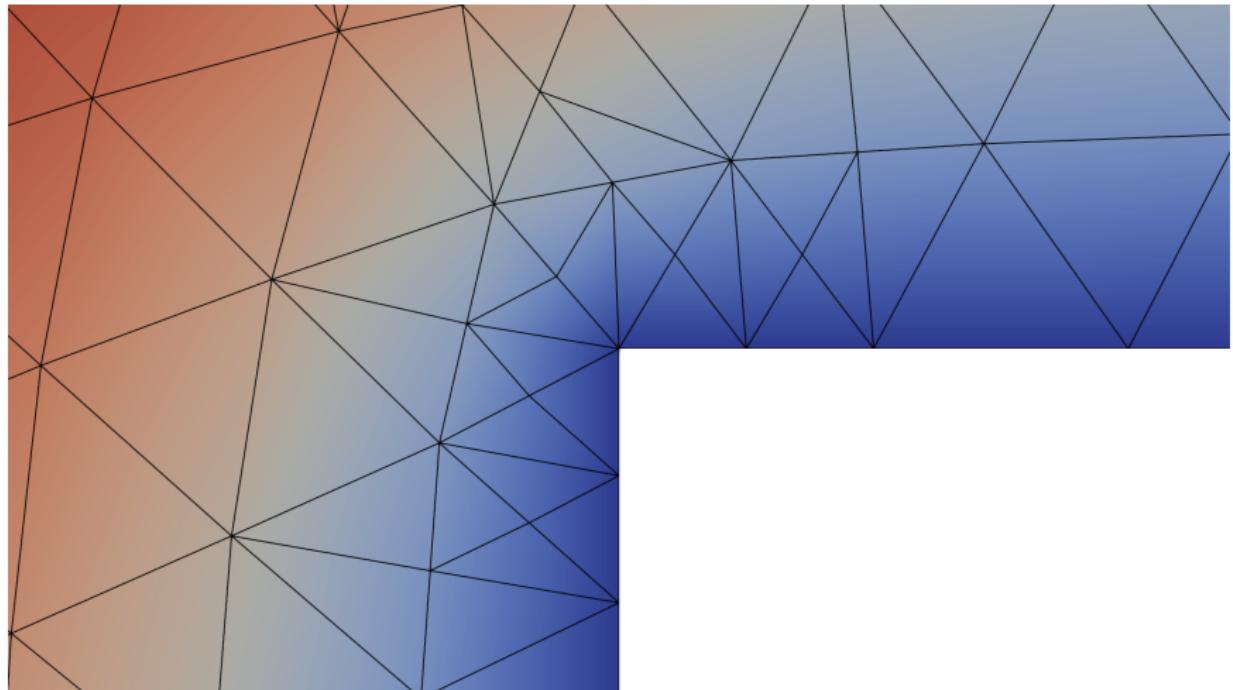
```
max_iterations = 10
error_estimators = []
dofs = []
for i in range(max_iterations):
    print(f"Solving on level {i}")

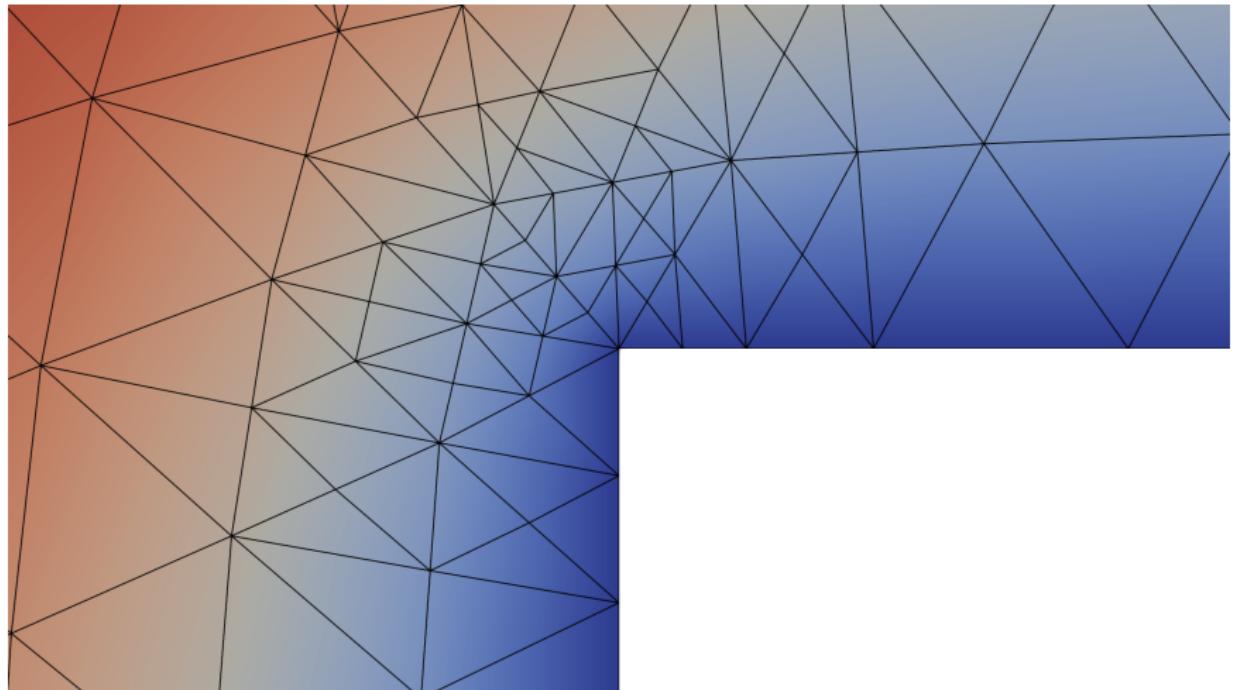
    uh = solve_poisson(mesh)
    File(f"output/adaptive_loop_{i}.pvd").write(uh)

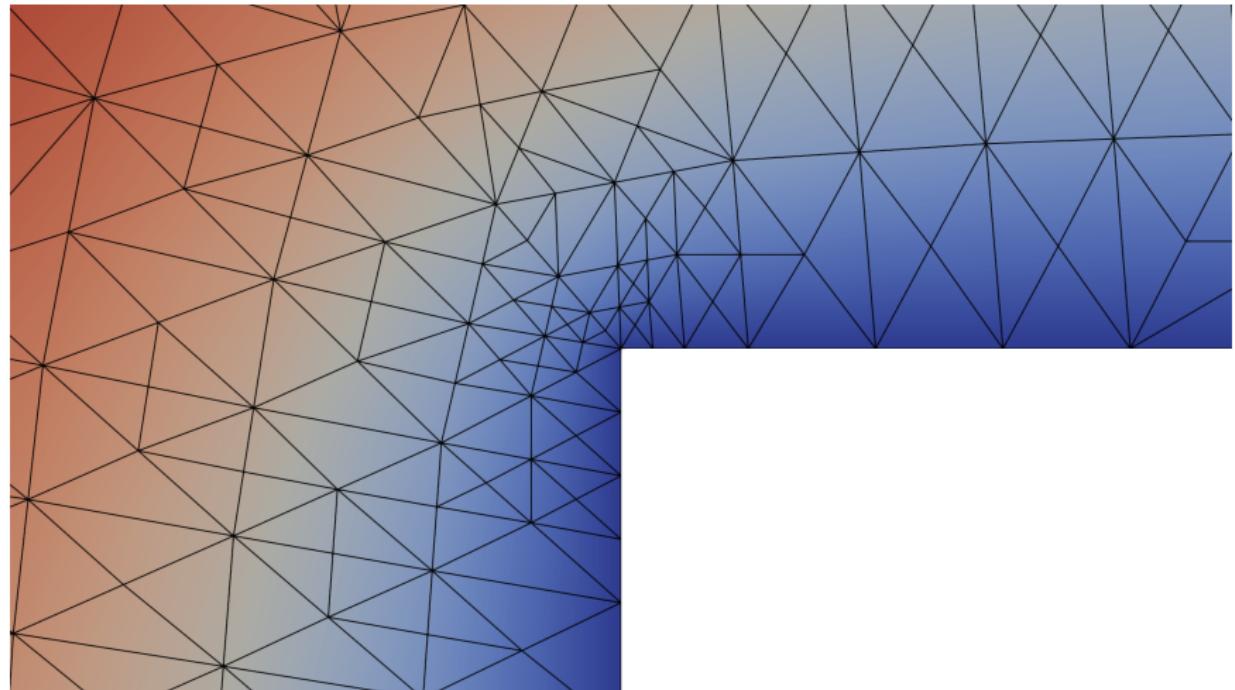
    (eta, error_est) = estimate_error(mesh, uh)
    print(f" ||u - u_h|| {error_est}")
    error_estimators.append(error_est)
    dofs.append(uh.function_space().dim())

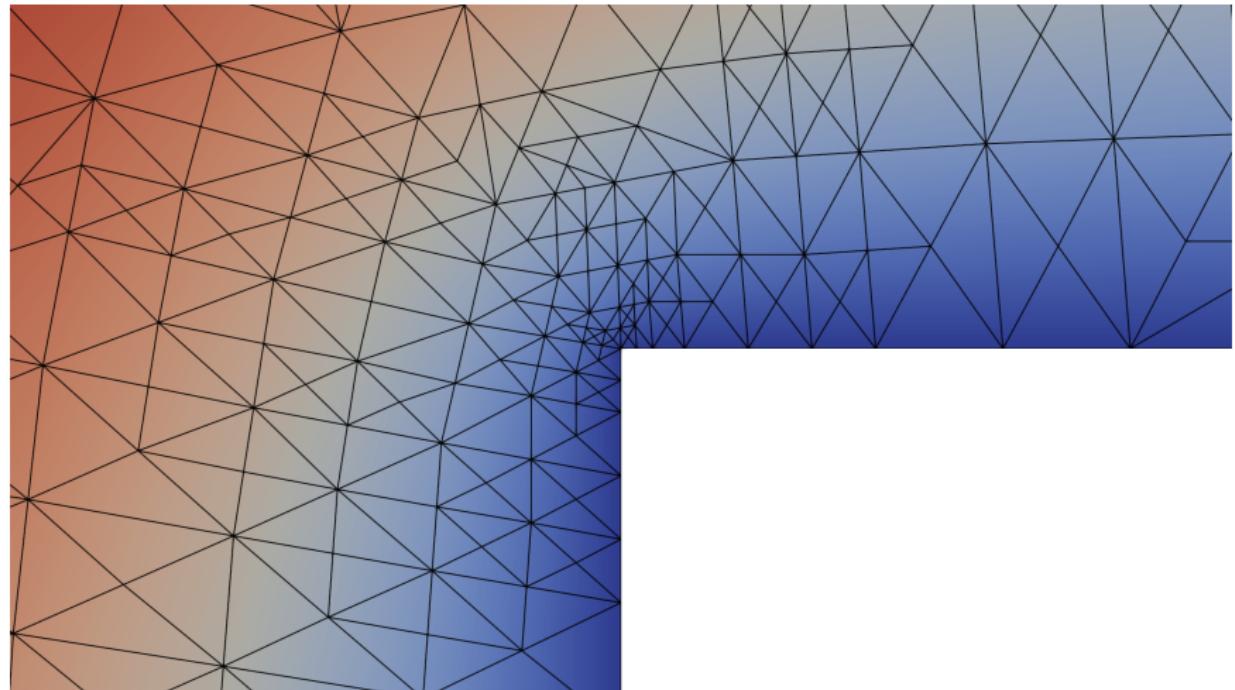
    mesh = adapt(mesh, eta)
```

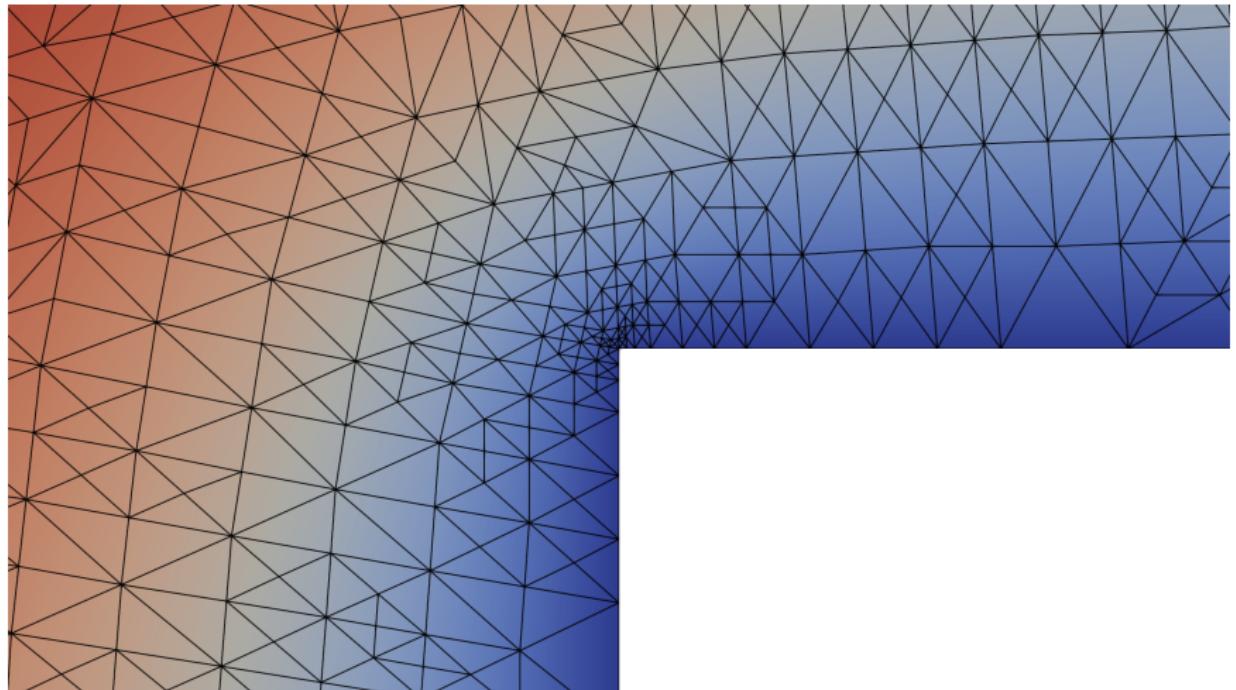


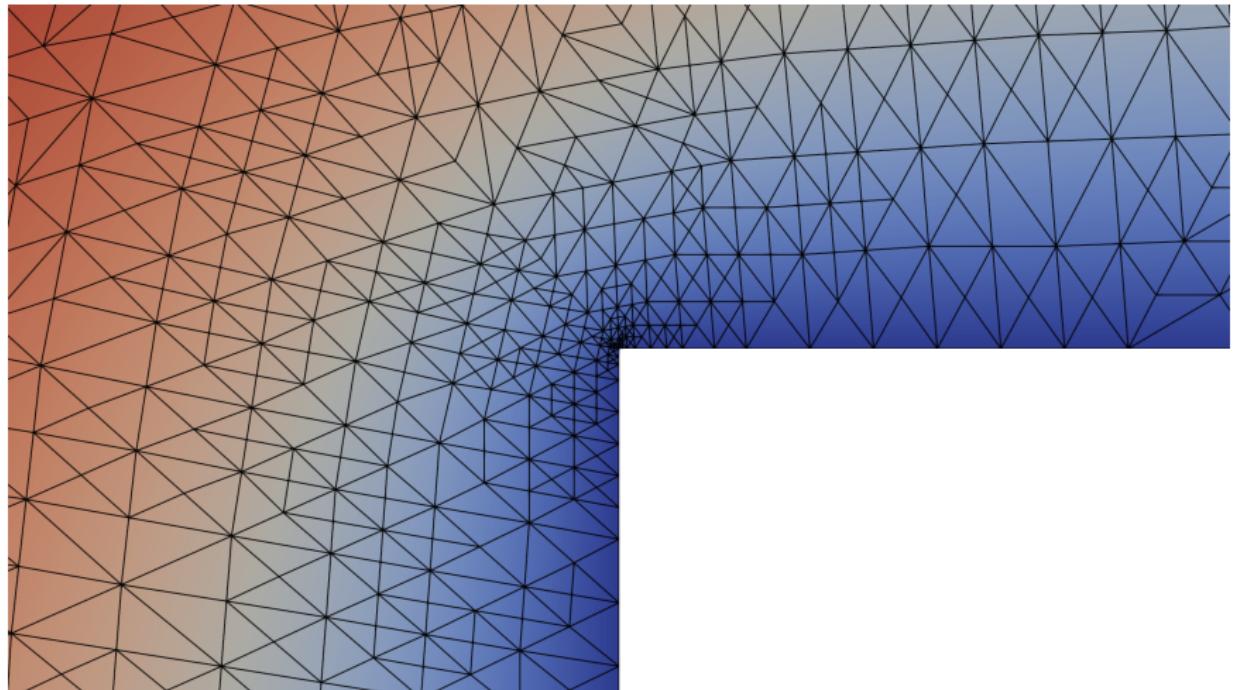


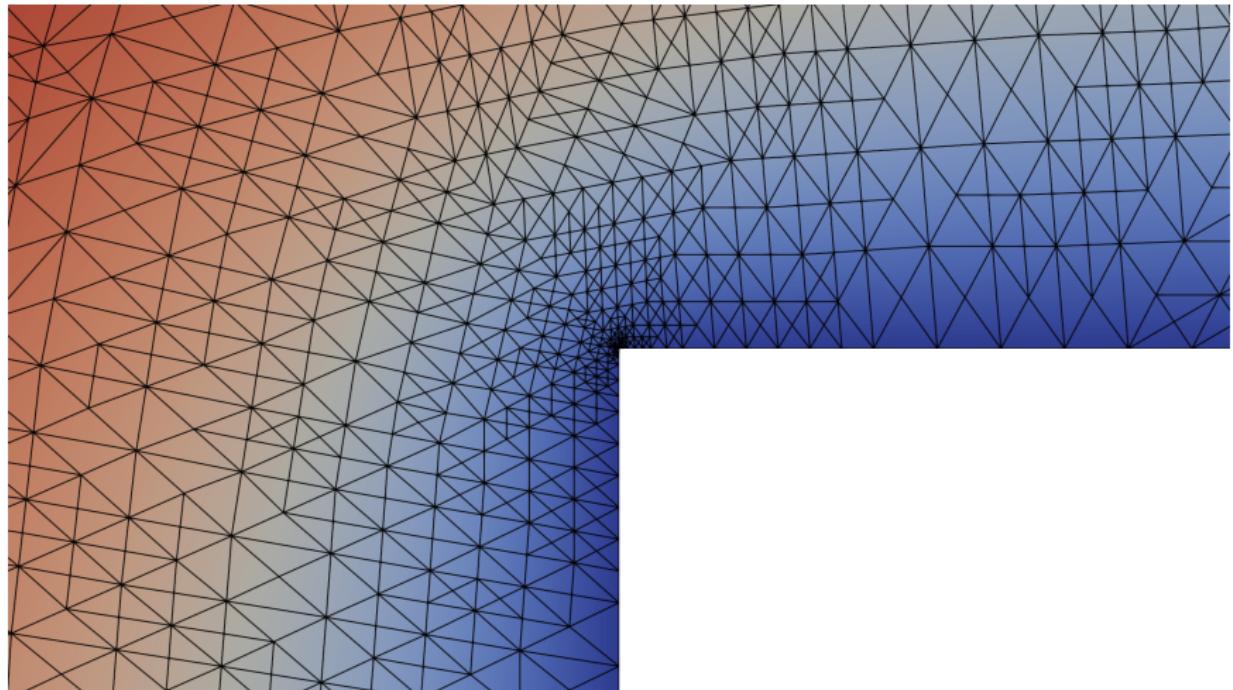


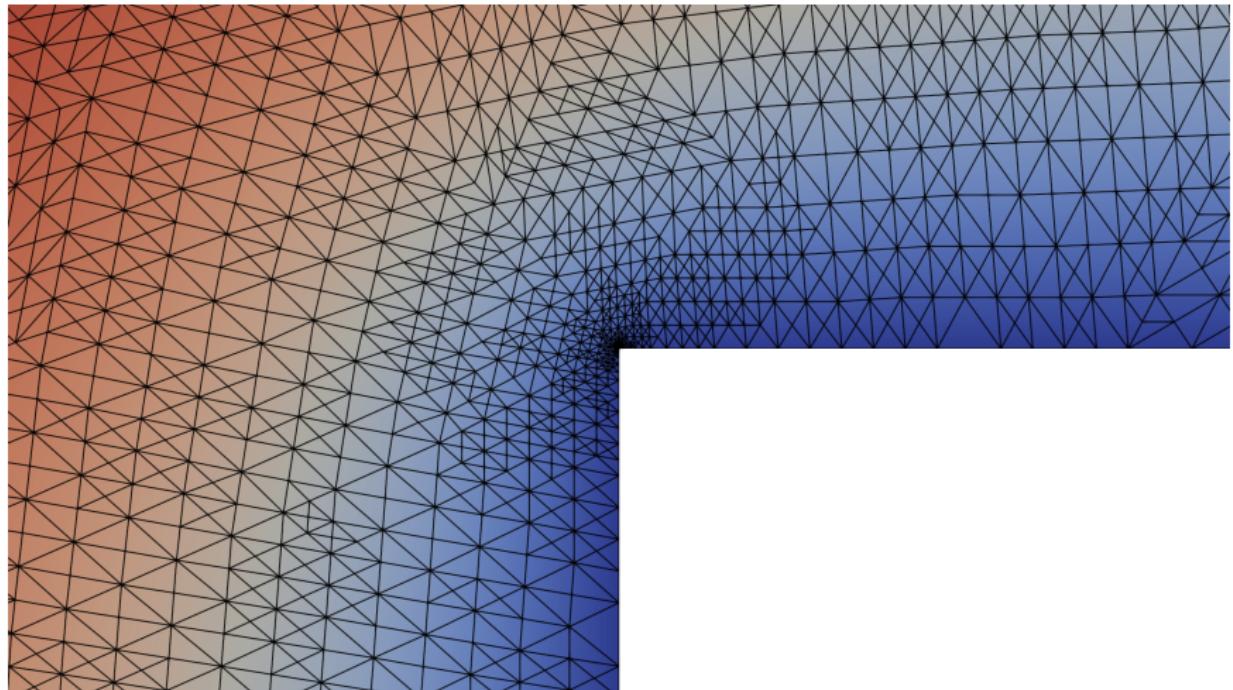


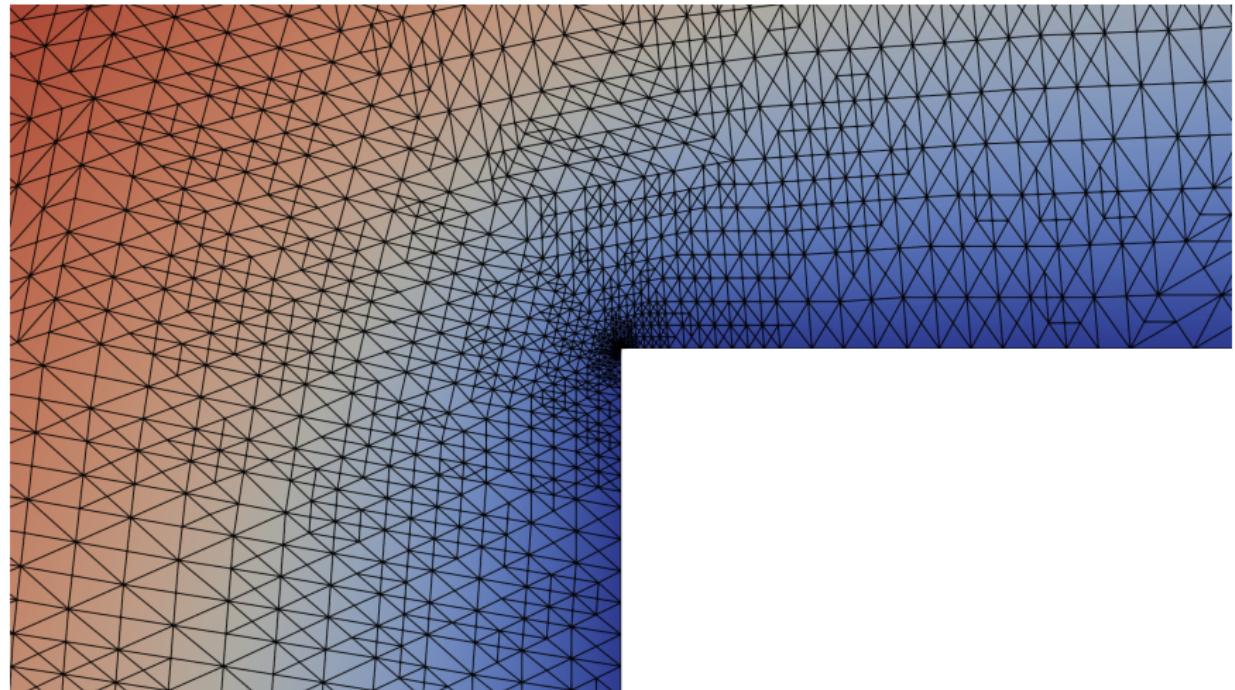










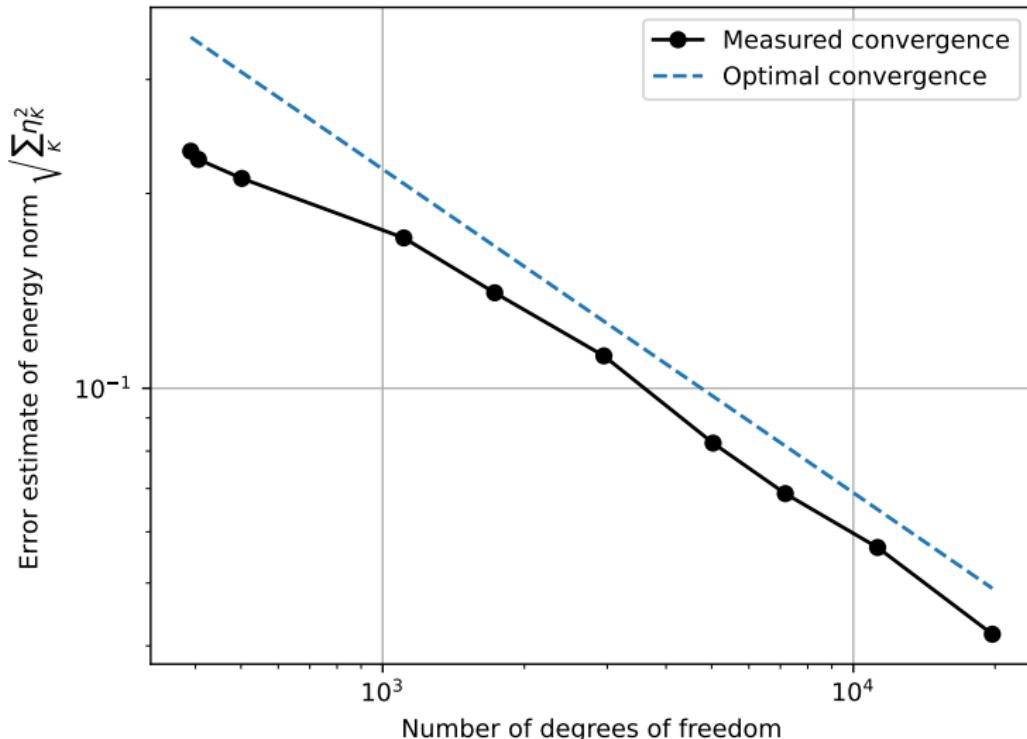


We can also check that

$$\text{error estimate} \propto (\# \text{ degrees of freedom})^{-1/2}.$$

We can also check that

$$\text{error estimate} \propto (\# \text{ degrees of freedom})^{-1/2}.$$



Challenge!

QII.3. Implement the `solve_poisson` function to solve the Poisson equation (with $f = 1$). Use this to drive the adaptive refinement code. Verify that

$$\text{error estimate} \propto (\# \text{ degrees of freedom})^{-1/2}.$$

Section 3

Netgen mesh hierarchies

We can build hierarchies of refined meshes from a Netgen mesh with

```
from ngsPETSc import NetgenHierarchy  
mh = NetgenHierarchy(ngmesh, 2) # 2 refinements
```

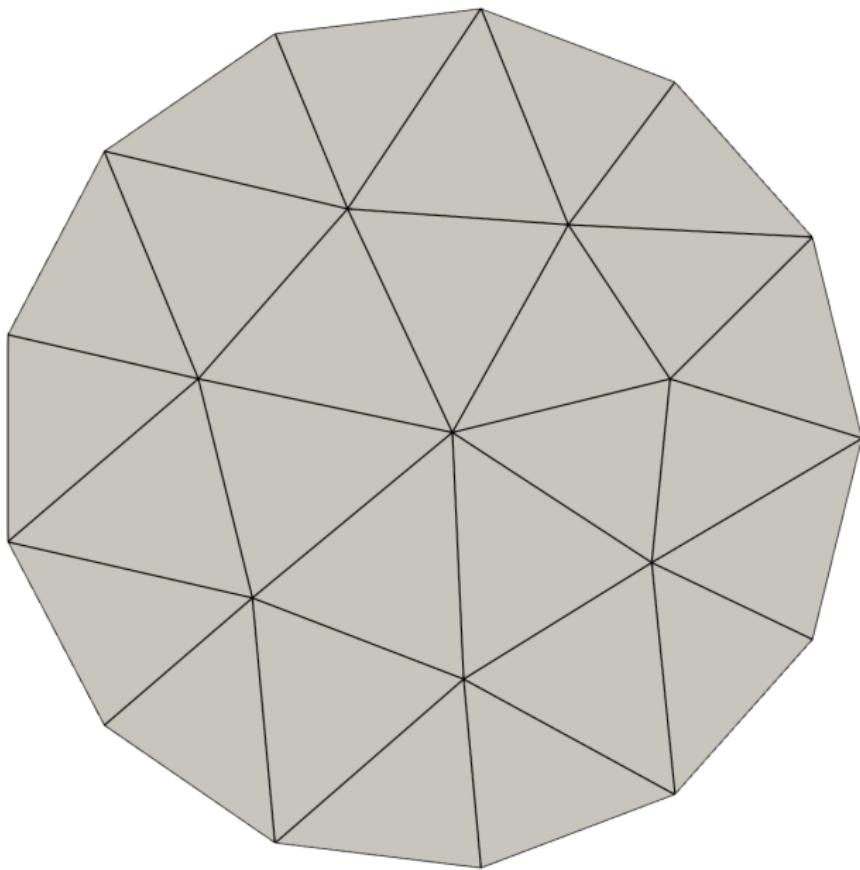
The refined meshes will conform to the underlying geometry!

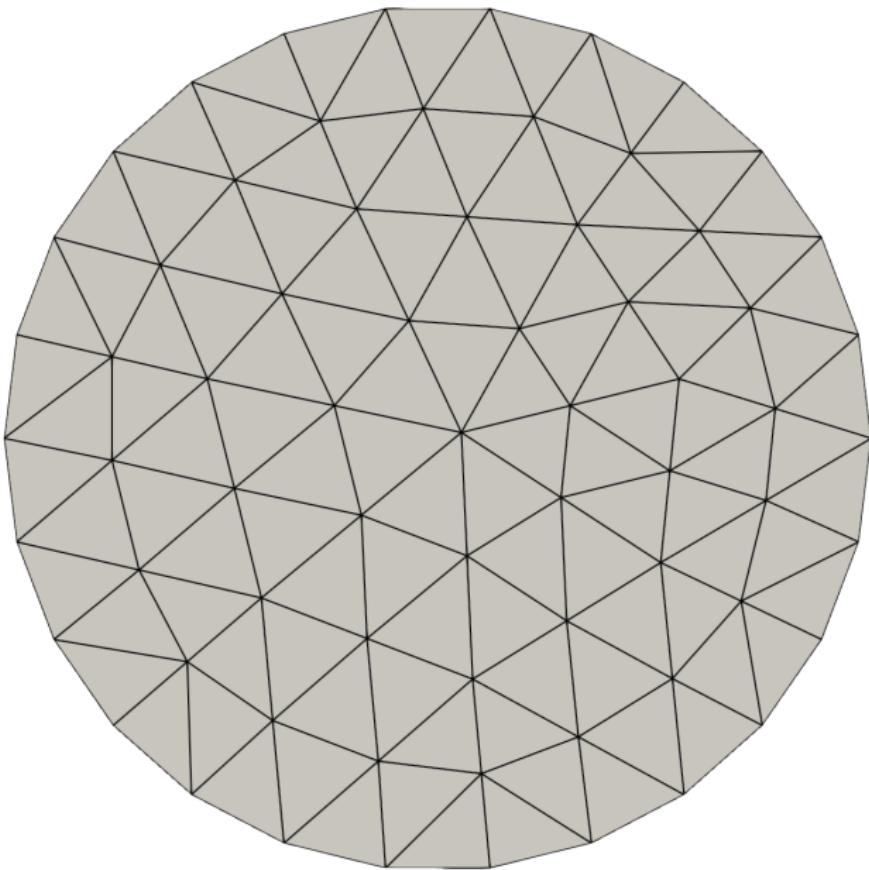
```
from firedrake import *
from netgen.occ import *
from ngsPETSc import NetgenHierarchy

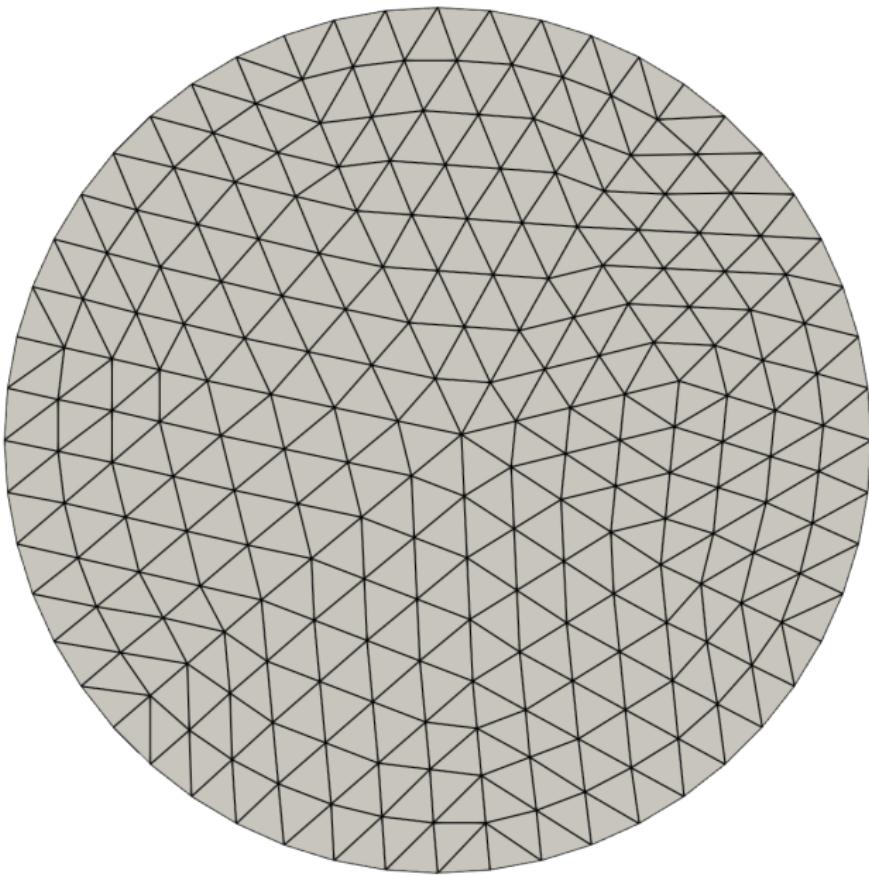
# Make 2D disk
disk = WorkPlane(Axes((0,0,0), n=Z, h=X)).Circle(0, 0, 1).Face()

geo = OCCGeometry(disk, dim=2)
ngmesh = geo.GenerateMesh(maxh=1)
mh = NetgenHierarchy(ngmesh, 2)

for (level, mesh) in enumerate(mh):
    File(f"output/mesh-{level}.pvd").write(mesh)
```







Firedrake supports non-nested grid transfers, so we can use this in a non-nested geometric multigrid scheme.

Firedrake supports non-nested grid transfers, so we can use this in a non-nested geometric multigrid scheme.

We will solve a problem where the exact solution is

$$u_{\text{ex}} = 1 - (x^2 + y^2)^3$$

so that we know

$$u = 0 \text{ on } \partial\Omega.$$

```
errors = []
for (level, mesh) in enumerate(mh):
    (x, y) = SpatialCoordinate(mesh)
    u_exact = 1 - (x**2 + y**2)**3

    V = FunctionSpace(mesh, "CG", 3) # use CG3
    u = Function(V, name="Solution")
    v = TestFunction(V)

    f = -div(grad(u_exact))
    F = (
        inner(grad(u), grad(v))*dx
        - inner(f, v)*dx
    )
    bc = DirichletBC(V, 0, "on_boundary")

    sp = {"ksp_type": "cg", "ksp_rtol": 1.0e-12,
          "ksp_monitor": None, "pc_type": "mg"} if level > 0 else None
    solve(F == 0, u, bc, solver_parameters=sp)

    print(f"||u - uh||: {norm(u - u_exact, 'H1')}")
```

Our multigrid scheme converges beautifully, but ...

$\|u - uh\|: 0.1544499890623805$

$\|u - uh\|: 0.055408215711143474$

$\|u - uh\|: 0.01966664317224813$

Convergence orders: [1.47896795 1.49434918]



Gil Strang



George Fix

Our multigrid scheme converges beautifully, but ...

```
||u - uh||: 0.1544499890623805
||u - uh||: 0.055408215711143474
||u - uh||: 0.01966664317224813
Convergence orders: [1.47896795 1.49434918]
```



Gil Strang

With CG₃, we should be dividing the error by 8 (2^3) each refinement. Instead, we are seeing convergence order ≈ 1.5 . Why?



George Fix

Our multigrid scheme converges beautifully, but ...

```
||u - uh||: 0.1544499890623805
||u - uh||: 0.055408215711143474
||u - uh||: 0.01966664317224813
Convergence orders: [1.47896795 1.49434918]
```



Gil Strang

With CG₃, we should be dividing the error by 8 (2^3) each refinement. Instead, we are seeing convergence order ≈ 1.5 . Why?

It is because our *geometry errors* are dominating. We are solving on an approximate domain

$$\Omega_h \approx \Omega$$

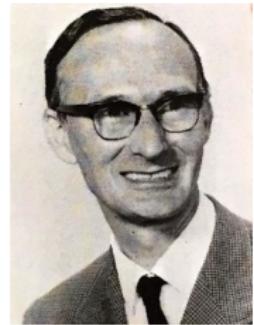
where Ω_h is constructed with piecewise linear elements.



George Fix

Instead, we can *curve* the sides of our cells:

```
mh = NetgenHierarchy(ngmesh, 2, order=3)
```



Bruce Irons

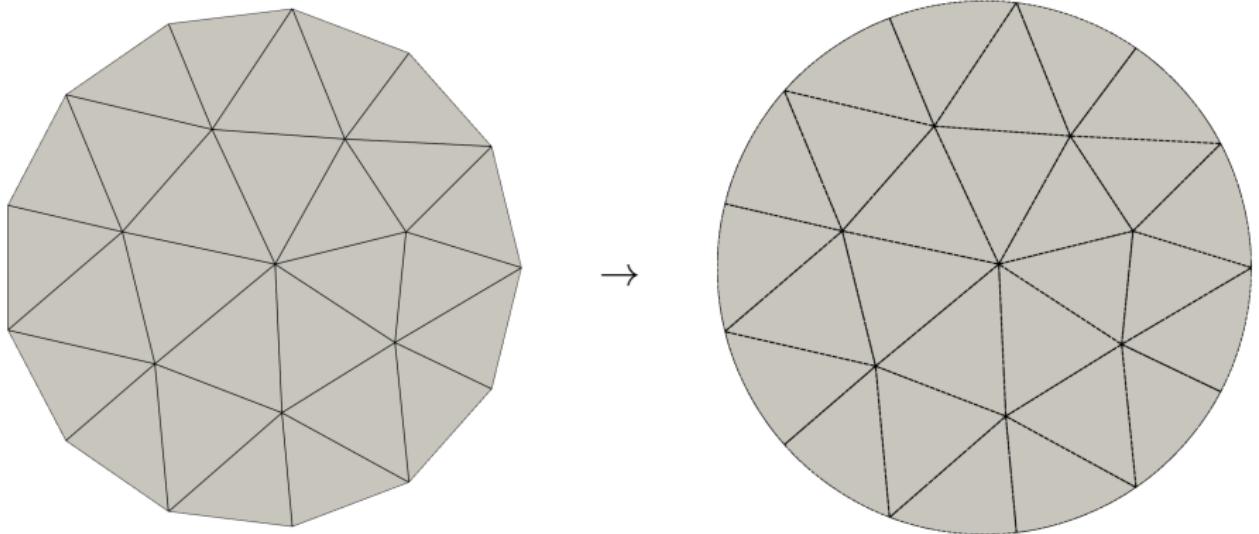
Instead, we can *curve* the sides of our cells:

```
mh = NetgenHierarchy(ngmesh, 2, order=3)
```

Using the same order for the geometry and the solution is called using “isoparametric elements”, a pun on the isoperimetric inequality.



Bruce Irons



$\|u - uh\|: 0.01068562949469617$
 $\|u - uh\|: 0.001309945838830347$
 $\|u - uh\|: 0.00015680841910396096$
Convergence orders: [3.02809283 3.06243224]

Challenge!

QII.4. Solve the Poisson problem on the disk, with exact solution

$$u_{\text{ex}} = 1 - (x^2 + y^2)^3.$$

Use CG₃ for both the solution and geometry.

Use conjugate gradients with geometric multigrid to solve the problem.

Verify that the expected order of convergence is achieved.

Solving PDEs with Firedrake: boundary conditions and nonlinearity

Patrick E. Farrell



University of Oxford

February 2024

Two kinds of boundary conditions

Two kinds of boundary conditions:

Essential boundary conditions

Enforced in the construction of the trial space, e.g.

$$V = \{u \in H^1(\Omega) : u|_{\partial\Omega} = g\}.$$

Natural boundary conditions

Implied by the variational form of the problem.

A boundary condition can switch from one type to another with a different formulation. For the Poisson equation, if you solve for u , then

$$u = g \text{ on } \partial\Omega$$

is an essential boundary condition.

A boundary condition can switch from one type to another with a different formulation. For the Poisson equation, if you solve for u , then

$$u = g \text{ on } \partial\Omega$$

is an essential boundary condition.

But with a mixed formulation for Poisson (solving for u and ∇u), you can enforce it naturally, because you get a boundary integral involving u .

A boundary condition can switch from one type to another with a different formulation. For the Poisson equation, if you solve for u , then

$$u = g \text{ on } \partial\Omega$$

is an essential boundary condition.

But with a mixed formulation for Poisson (solving for u and ∇u), you can enforce it naturally, because you get a boundary integral involving u .

Ease of enforcing boundary conditions is one of the big factors influencing the choice of variables to solve for, and the choice of variational formulation.

Poisson with Robin boundary conditions

Consider Poisson's equation with a Robin boundary condition:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ \alpha u + \nabla u \cdot n &= g \quad \text{on } \partial\Omega \end{aligned}$$

Poisson with Robin boundary conditions

Consider Poisson's equation with a Robin boundary condition:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ \alpha u + \nabla u \cdot n &= g \quad \text{on } \partial\Omega \end{aligned}$$

Integrating by parts, we find

$$\begin{aligned} \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} (\nabla u \cdot n)v \, ds &= \int_{\Omega} fv \, dx \\ \implies \int_{\Omega} \nabla u \cdot \nabla v \, dx + \int_{\partial\Omega} \alpha uv \, ds &= \int_{\Omega} fv \, dx + \int_{\partial\Omega} gv \, ds. \end{aligned}$$

Integrating over surfaces

The surface integral term can be implemented with the `ds` measure:

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx + \int_{\partial\Omega} \alpha uv \, ds - \int_{\Omega} fv \, dx - \int_{\partial\Omega} gv \, ds = 0$$

```
F = (
    inner(grad(u), grad(v))*dx
    + inner(alpha*u, v)*ds
    - inner(f, v)*dx
    - inner(g, v)*ds
)
```

Imposing mixed boundary conditions

Consider Poisson's equation with mixed boundary conditions:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ \alpha u + \nabla u \cdot n &= g \quad \text{on } \partial\Omega|_{x=0} \\ u &= h \quad \text{on } \partial\Omega|_{x>0} \end{aligned}$$

To specify the boundaries for the different conditions, we use the surface colouring of the mesh.

Imposing mixed boundary conditions

Consider Poisson's equation with mixed boundary conditions:

$$\begin{aligned} -\Delta u &= f \quad \text{in } \Omega \\ \alpha u + \nabla u \cdot n &= g \quad \text{on } \partial\Omega|_{x=0} \\ u &= h \quad \text{on } \partial\Omega|_{x>0} \end{aligned}$$

To specify the boundaries for the different conditions, we use the surface colouring of the mesh.

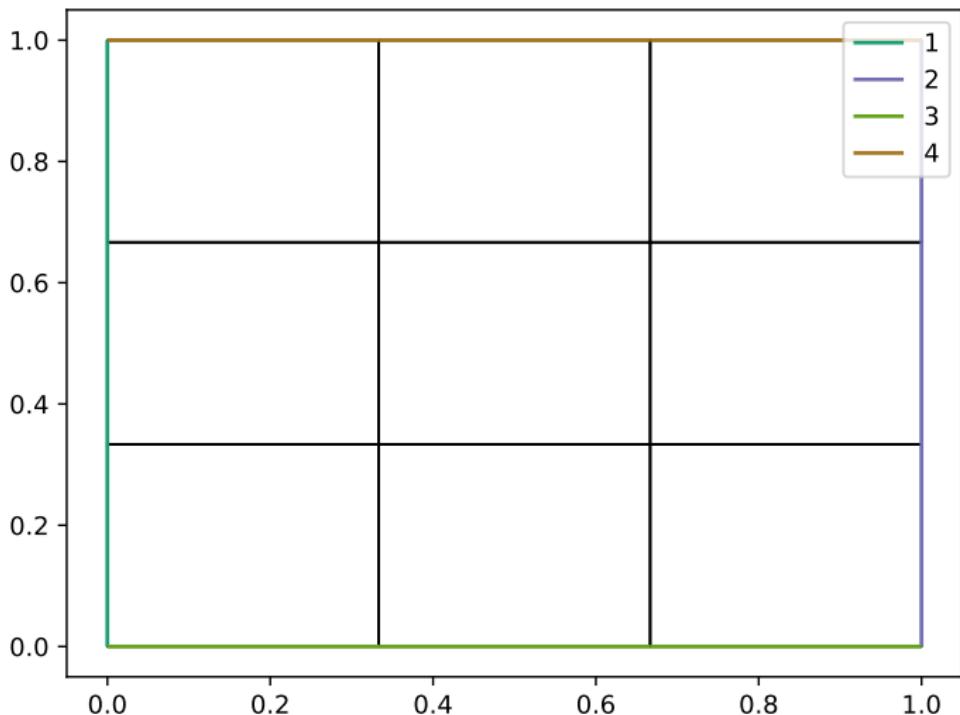
On a UnitSquareMesh, the left boundary is colour 1. So we can do

```
bc = DirichletBC(V, h, 1)
```

If you're not sure how the mesh is coloured, you can plot it (in 2D).

```
import matplotlib.pyplot as plt
from firedrake import *
from firedrake.pyplot import *

mesh = UnitSquareMesh(3, 3, quadrilateral=True)
triplot(mesh)
plt.gca().legend()
plt.show()  # not necessary on Colab
```



Integrating over subdomains

To integrate over part of the boundary, we pass the surface colour to the Measure.

```
# Integrate over facets with colours 2, 3, 4
F = (
    ...
    + inner(alpha*u, v)*ds((2, 3, 4))
)
```

Integrating over subdomains

To integrate over part of the boundary, we pass the surface colour to the Measure.

```
# Integrate over facets with colours 2, 3, 4
F = (
    ...
    + inner(alpha*u, v)*ds((2, 3, 4))
)
```

Note that if you impose a DirichletBC on a surface with colour j , then the TestFunction is automatically set to zero on that boundary.

Challenge!

QIII.1. On $\Omega = (0, 1)^2$, solve the problem

$$-\Delta u = 1 \quad \text{in } \Omega,$$

$$u + \nabla u \cdot n = 1 \quad \text{on } \partial\Omega|_{x=0},$$

$$u = 1 \quad \text{on } \partial\Omega|_{x>0}.$$

Section 2

Colouring meshes in Netgen

When building meshes with Netgen, you can give labels to objects.

```
from firedrake import *
from netgen.occ import *
from ngsPETSc import NetgenHierarchy

# Netgen uses string labels
cube = Box(Pnt(0,0,0), Pnt(1,1,1)).bc("cube")
sphere = Sphere(Pnt(1, 1, 1), 0.5).bc("sphere")

# We need to Glue them together to keep
# distinct surface labels
geo = OCCGeometry(Glue([cube, sphere]))
ngmesh = geo.GenerateMesh(maxh=0.5)

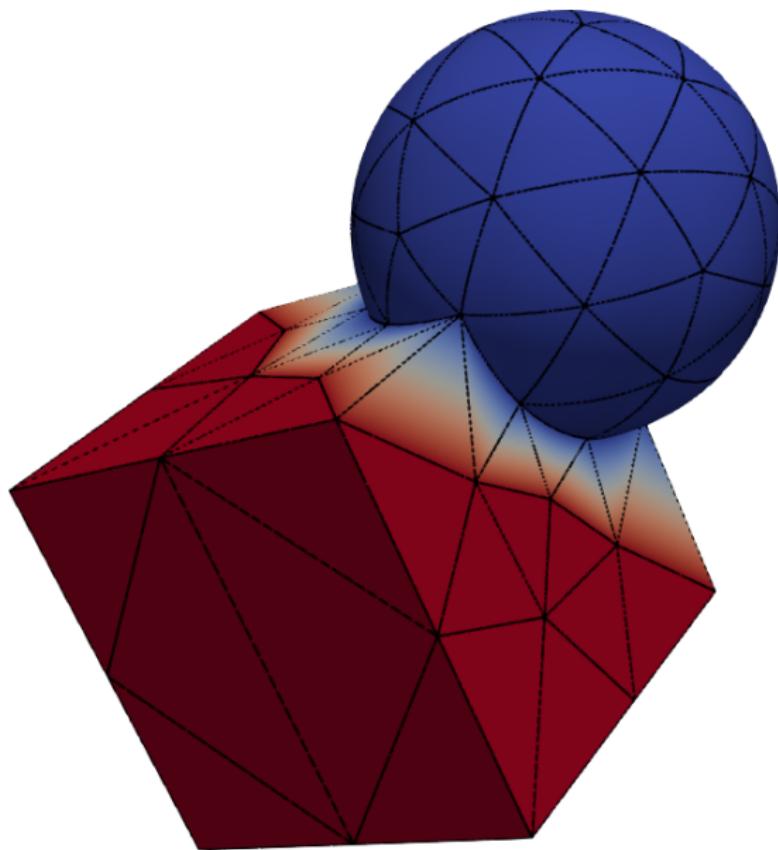
# Firedrake uses integer labels, so we need to convert.
cube_labels = [i + 1 for (i, name) in
               enumerate(ngmesh.GetRegionNames(codim=1)) if name == "cube"]
sphere_labels = [i + 1 for (i, name) in
                 enumerate(ngmesh.GetRegionNames(codim=1)) if name == "sphere"]
```

```
# Make a high-order mesh of the union of cube and sphere
mh = NetgenHierarchy(ngmesh, 0, order=3)
mesh = mh[-1]

V = FunctionSpace(mesh, "CG", 1)
u = Function(V, name="Solution")
v = TestFunction(V)

# OK, this data is not  $H^{1/2}$ , but
# we use it for illustration anyway
bcs = [DirichletBC(V, +1, cube_labels),
        DirichletBC(V, -1, sphere_labels)]

F = inner(grad(u), grad(v))*dx
solve(F == 0, u, bcs)
```



You can also label edges (2D) or faces (3D) separately:

```
cube = Box(Pnt(0,0,0), Pnt(1,1,1)).bc("cube")
cube.faces.Min(X).name = "left"
cube.faces.Min(Y).name = "front"
cube.faces.Max(Z).name = "top"
```

In 2D, use `shape.edges` instead of `shape.faces`.

Section 3

Nonlinear boundary conditions

Stefan–Boltzmann radiation conditions

The Poisson equation models stationary heat distribution. If the body loses energy via black-body radiation, we get a *Stefan–Boltzmann boundary condition*:

$$\begin{aligned} -\Delta u &= f && \text{in } \Omega, \\ \nabla u \cdot n &= \beta(c^4 - u^4) && \text{on } \partial\Omega, \end{aligned}$$

where $c > 0$ is the temperature of the surrounding medium and $\beta > 0$ is a coefficient.

This equation is *semilinear*.

Classification of PDEs

Linear PDEs

The coefficients do not depend on the solution.

Classification of PDEs

Linear PDEs

The coefficients do not depend on the solution.

Semilinear PDEs

The coefficients of the highest-order derivatives do not depend on the solution.

Classification of PDEs

Linear PDEs

The coefficients do not depend on the solution.

Semilinear PDEs

The coefficients of the highest-order derivatives do not depend on the solution.

Quasilinear PDEs

The coefficients of the highest-order derivatives depend on lower-order derivatives of the solution.

Classification of PDEs

Linear PDEs

The coefficients do not depend on the solution.

Semilinear PDEs

The coefficients of the highest-order derivatives do not depend on the solution.

Quasilinear PDEs

The coefficients of the highest-order derivatives depend on lower-order derivatives of the solution.

Fully nonlinear PDEs

All coefficients can depend on all derivatives of the solution.

Poisson with Stefan–Boltzmann

Integrating by parts, we find

$$\begin{aligned} \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} (\nabla u \cdot n)v \, ds &= \int_{\Omega} fv \, dx \\ \Rightarrow \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} \beta(c^4 - u^4)v \, ds &= \int_{\Omega} fv \, dx \end{aligned}$$

Poisson with Stefan–Boltzmann

Integrating by parts, we find

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} (\nabla u \cdot n)v \, ds = \int_{\Omega} fv \, dx$$

$$\Rightarrow \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\partial\Omega} \beta(c^4 - u^4)v \, ds = \int_{\Omega} fv \, dx$$

```
F = (
    inner(grad(u), grad(v))*dx
    - inner(beta*(c**4 - u**4), v)*ds
    - inner(f, v)*dx
)
```

We will employ an algorithm for dealing with the nonlinearity, the *Newton–Kantorovich* iteration.

We will employ an algorithm for dealing with the nonlinearity, the *Newton–Kantorovich* iteration.

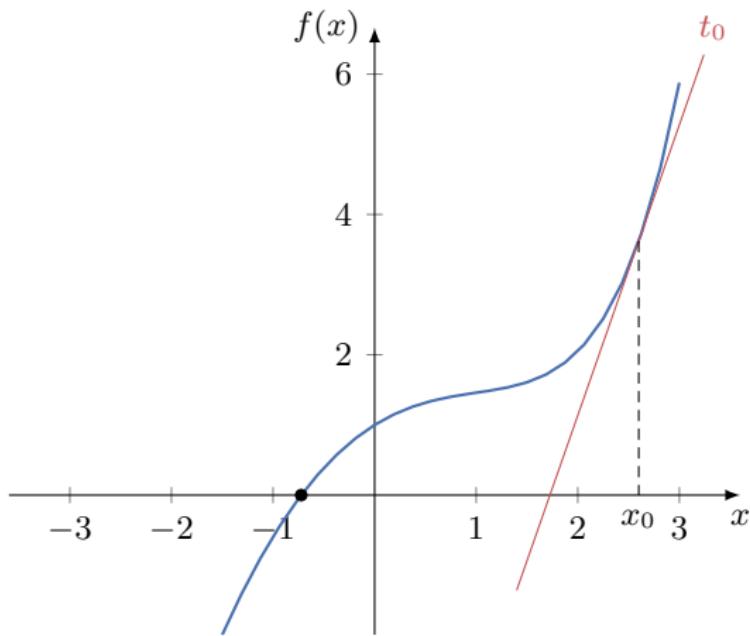
This will compute the solution of the nonlinear problem by solving a sequence of linear PDE problems.

We will employ an algorithm for dealing with the nonlinearity, the *Newton–Kantorovich* iteration.

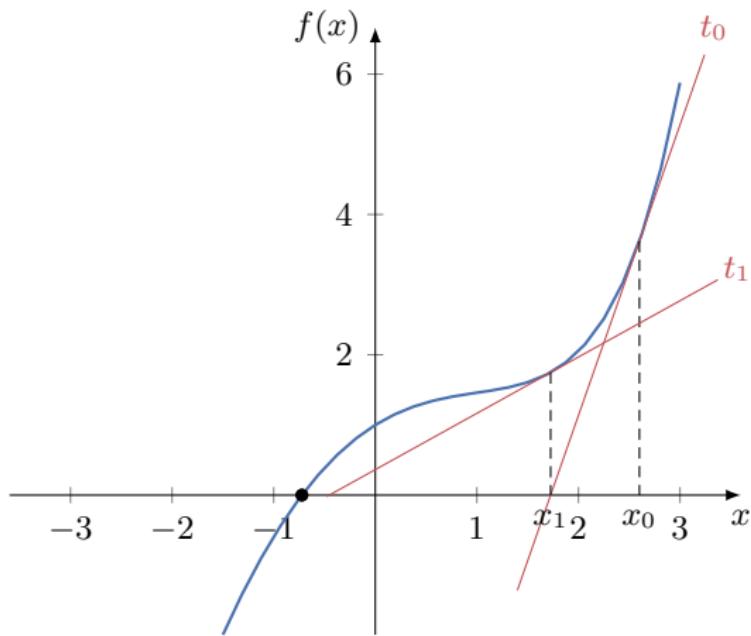
This will compute the solution of the nonlinear problem by solving a sequence of linear PDE problems.

First, let's recall Newton's method in \mathbb{R} and \mathbb{R}^N .

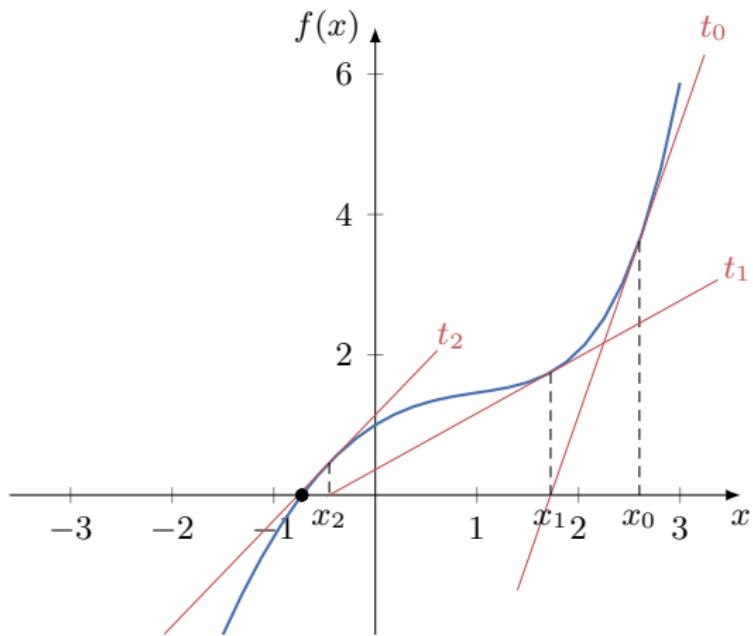
Essential idea: solve a succession of *linearised* rootfinding problems.



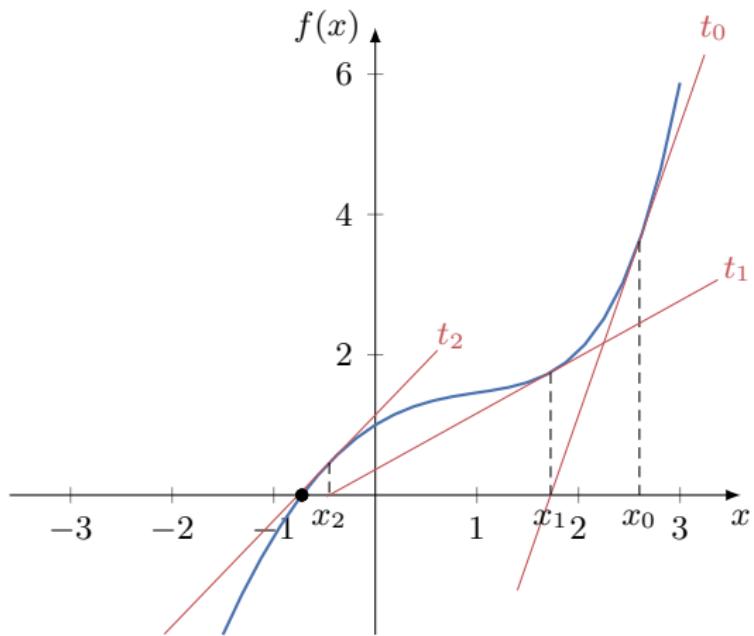
Essential idea: solve a succession of *linearised* rootfinding problems.



Essential idea: solve a succession of *linearised* rootfinding problems.



Essential idea: solve a succession of *linearised* rootfinding problems.



solve $f'(x_k)\delta x_k = -f(x_k)$; update $x_{k+1} = x_k + \delta x_k$.

Termination

The algorithm terminates if $f(x_k) = 0$, as desired.

Termination

The algorithm terminates if $f(x_k) = 0$, as desired.

Invertibility

We require $f'(x_k)$ to be invertible at every iteration.

Termination

The algorithm terminates if $f(x_k) = 0$, as desired.

Invertibility

We require $f'(x_k)$ to be invertible at every iteration.

Poor global convergence

The initial guess matters. With poor initial guesses, Newton's method may diverge to infinity, or get stuck in a cycle.

Termination

The algorithm terminates if $f(x_k) = 0$, as desired.

Invertibility

We require $f'(x_k)$ to be invertible at every iteration.

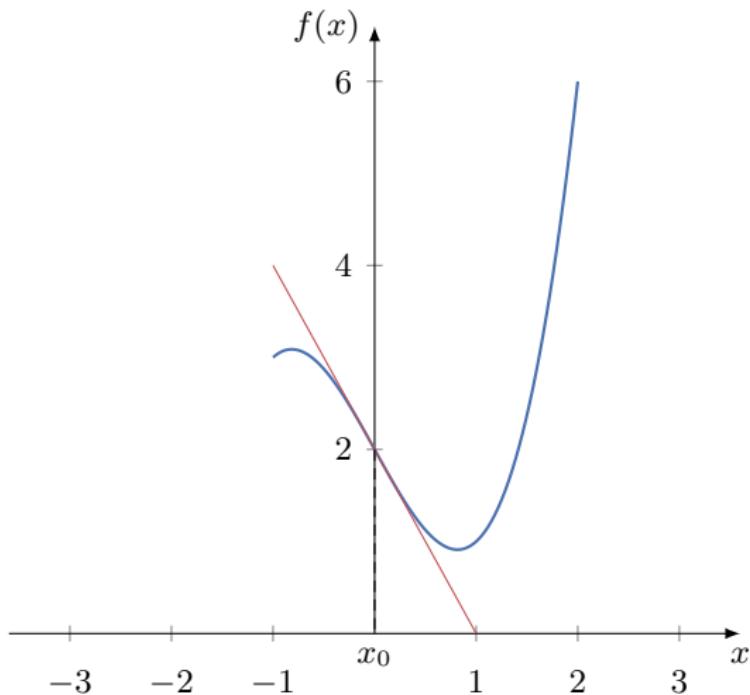
Poor global convergence

The initial guess matters. With poor initial guesses, Newton's method may diverge to infinity, or get stuck in a cycle.

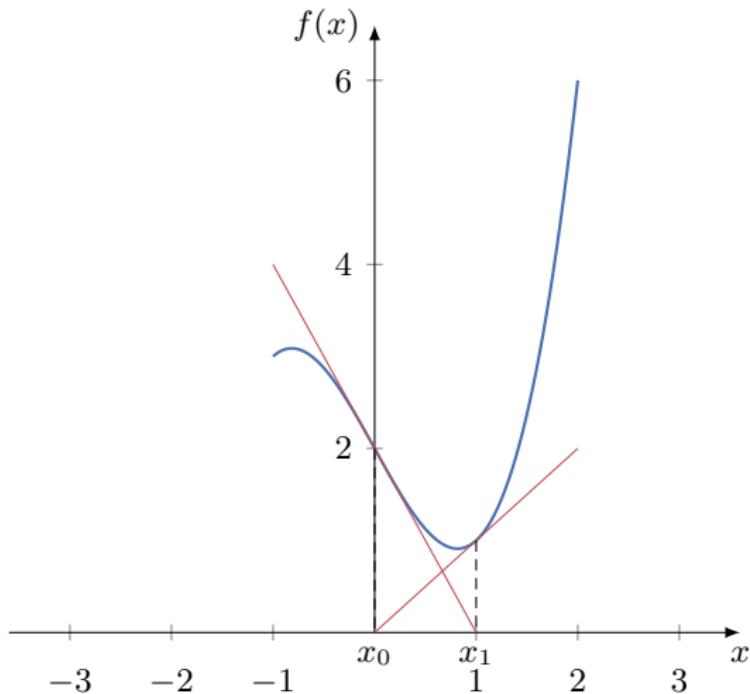
Good local convergence

If f is smooth, the solution is isolated, and the guess close, Newton converges quadratically.

Consider $f(x) = x^3 - 2x + 2$ with $x_0 = 0$.



Consider $f(x) = x^3 - 2x + 2$ with $x_0 = 0$.



This geometric reasoning is hard to generalise to higher dimensions. Let's look at a derivation that *does* extend.

This geometric reasoning is hard to generalise to higher dimensions. Let's look at a derivation that *does* extend.

Consider the Taylor expansion of $f : \mathbb{R} \rightarrow \mathbb{R}$ around x_k :

$$f(x_k + \delta x_k) = f(x_k) + f'(x_k)\delta x_k + \mathcal{O}(\delta x_k^2).$$

This geometric reasoning is hard to generalise to higher dimensions. Let's look at a derivation that *does* extend.

Consider the Taylor expansion of $f : \mathbb{R} \rightarrow \mathbb{R}$ around x_k :

$$f(x_k + \delta x_k) = f(x_k) + f'(x_k)\delta x_k + \mathcal{O}(\delta x_k^2).$$

Linearise the model by ignoring higher-order terms:

$$f(x_k + \delta x) \approx f(x_k) + f'(x_k)\delta x_k$$

and find δx such that $f(x_k + \delta x) \approx 0$:

$$0 = f(x_k) + f'(x_k)\delta x_k.$$

This does extend to an $F \in C^1(\mathbb{R}^N; \mathbb{R}^N)$. Newton's method is to

solve $DF(x_k)\delta x_k = -F(x_k)$; update $x_{k+1} = x_k + \delta x_k$,

where DF is the Jacobian (Fréchet derivative) of F .

This does extend to an $F \in C^1(\mathbb{R}^N; \mathbb{R}^N)$. Newton's method is to

solve $DF(x_k)\delta x_k = -F(x_k)$; update $x_{k+1} = x_k + \delta x_k$,

where DF is the Jacobian (Fréchet derivative) of F .

All the previous remarks apply, plus

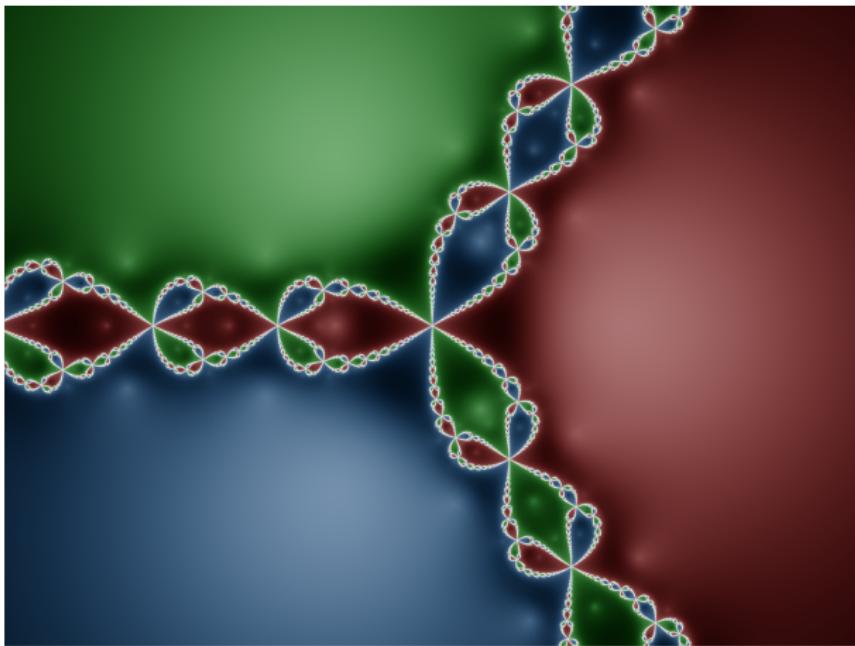
Affine covariance

Given any nonsingular $A \in \mathbb{R}^{N \times N}$, Newton's method applied to AF yields *the exact same sequence of iterates* as applied to F , starting from the same initial guess.

We can visualise the erratic global convergence with a *Newton fractal*.

$$f : \mathbb{C} \rightarrow \mathbb{C}$$

$$f(z) = z^3 - 1.$$



The generalisation of Newton's method to Banach spaces is called the *Newton–Kantorovich* algorithm.



Leonid Kantorovich

The generalisation of Newton's method to Banach spaces is called the *Newton–Kantorovich* algorithm.

Kantorovich's theorem (1948) is a triumph of both PDE analysis and numerical analysis. It *does not assume the existence of a solution*: given certain conditions on the residual and initial guess, it *proves* the existence and local uniqueness of a solution.



Leonid Kantorovich

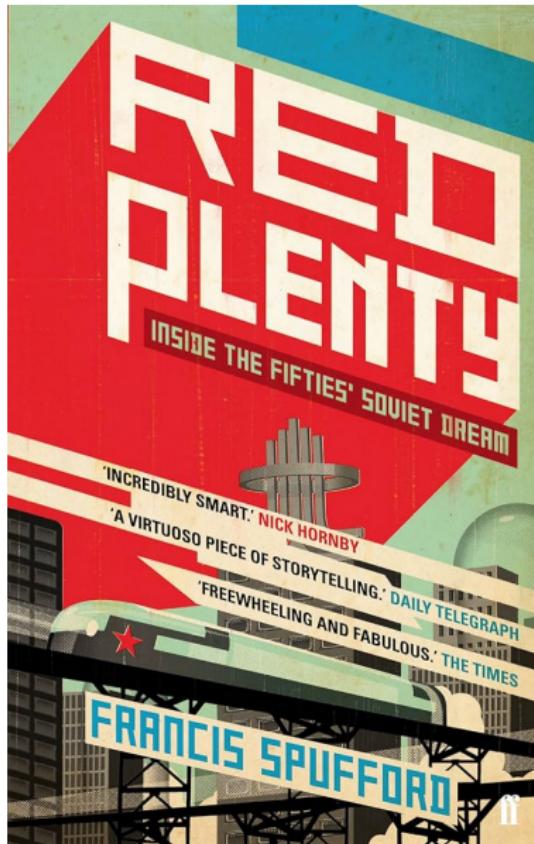
The generalisation of Newton's method to Banach spaces is called the *Newton–Kantorovich* algorithm.

Kantorovich's theorem (1948) is a triumph of both PDE analysis and numerical analysis. It *does not assume the existence of a solution*: given certain conditions on the residual and initial guess, it *proves* the existence and local uniqueness of a solution.



Leonid Kantorovich

With a good initial guess, and great cleverness, it is possible to devise *computer-assisted proofs* of the existence of solutions to infinite-dimensional nonlinear problems.



Newton–Kantorovich method

The main algorithm for solving nonlinear equations:

Newton–Kantorovich algorithm

- ▶ Apply boundary conditions to u .
- ▶ While not converged:
 - ▶ Solve: find $\delta u \in V_0$ such that

$$F'(u; v, \delta u) = -F(u; v) \quad \forall v \in \hat{V}$$

- ▶ Set $u = u + \delta u$.

Properties of Newton–Kantorovich

Some facts about Newton–Kantorovich:

Computational challenge

Main cost: solving the linearised system.

Properties of Newton–Kantorovich

Some facts about Newton–Kantorovich:

Computational challenge

Main cost: solving the linearised system.

Fast convergence

Converges quadratically if close to a regular solution.

Properties of Newton–Kantorovich

Some facts about Newton–Kantorovich:

Computational challenge

Main cost: solving the linearised system.

Fast convergence

Converges quadratically if close to a regular solution.

Divergence

Can diverge if initialised far from a solution.

Properties of Newton–Kantorovich

Some facts about Newton–Kantorovich:

Computational challenge

Main cost: solving the linearised system.

Fast convergence

Converges quadratically if close to a regular solution.

Divergence

Can diverge if initialised far from a solution.

Multiple solutions

Can converge to different solutions from different initial guesses.

Using Newton–Kantorovich in Firedrake

On entry: `u` is the **initial guess**.

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

On exit: `u` is the **solution**.

Using Newton–Kantorovich in Firedrake

On entry: \mathbf{u} is the **initial guess**.

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

On exit: \mathbf{u} is the **solution**.

Setting a good initial guess is crucial for convergence!

```
u.interpolate(Constant(1))
solve(F == 0, u, bcs)
```

To control the nonlinear solve, use the `solver_parameters` dictionary.

```
sp = {"snes_monitor": None,           # monitor convergence info
      "snes_converged_reason": None,   # say why it finished
      "snes_atol": 1e-8,              # termination criterion
      "snes_linesearch_type": "l2"}    # use a line search

solve(F == 0, u, bcs, solver_parameters=sp)
```

Challenge!

QIII.2. Solve the problem

$$\begin{aligned} -\Delta u &= 1000x(1-x)y(1-y) && \text{in } \Omega = (0, 1)^2 \\ \nabla u \cdot n &= (0.5^4 - u^4) && \text{on } \partial\Omega. \end{aligned}$$

Solving PDEs with Firedrake: time-dependent problems

Patrick E. Farrell



University of Oxford

February 2024

The heat equation

We will solve the simplest extension of the Poisson problem into the time domain, the heat equation:

$$\frac{\partial u}{\partial t} - \Delta u = f \text{ in } \Omega \text{ for } t > 0$$

$$u = g \text{ on } \partial\Omega \text{ for } t > 0$$

$$u = u^0 \text{ in } \Omega \text{ at } t = 0$$

The solution $u = u(x, t)$, the right-hand side $f = f(x, t)$ and the boundary value $g = g(x, t)$ may vary in space ($x = (x_0, x_1, \dots)$) and time (t). The initial value u^0 is a function of space only.

Time-discretisation of the heat equation

There are many discretisations in time, each with different stability and efficiency properties. We will first implement backward Euler. Casting the problem as

$$\frac{\partial u}{\partial t} = h(u, t),$$

we will solve for u_n with

$$\frac{u_n - u_{n-1}}{\Delta t} = h(u_n, t_n),$$

for some timestep $\Delta t > 0$. The error in time is $\mathcal{O}(\Delta t)$.

This is an implicit (linear or nonlinear) equation for u_n .

Time-discretisation of the heat equation

Algorithm

- ▶ Start with u_0 and choose a timestep $\Delta t > 0$.
- ▶ For $n = 1, 2, \dots$, solve for u^n :

$$\frac{u_n - u_{n-1}}{\Delta t} = h(u_n, t_n).$$

Variational problem for the heat equation

The semi-discretised backward Euler problem for u_n is

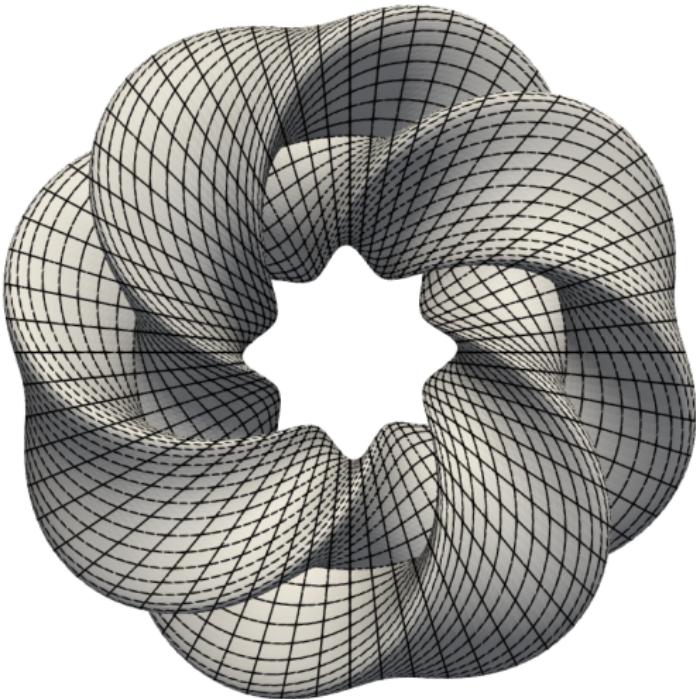
$$\frac{u_n - u_{n-1}}{\Delta t} = h(u_n, t_n).$$

We also need to discretise this in space. For the heat equation, this yields the variational problem: find $u_n \in V_h$ such that

$$\int_{\Omega} \frac{u_n - u_{n-1}}{\Delta t} v \, dx + \Delta t \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\Omega} f v \, dx = 0,$$

for all $v \in \hat{V}_h$.

For fun, let's solve the problem on a two-dimensional manifold in three-dimensional space.



This is a *Gray's Klein bottle*.

```
from firedrake import *

# Parametric domain and coordinates
p_mesh = PeriodicRectangleMesh(64, 64, 4*pi, 2*pi, quadrilateral=True)
(u, v) = SpatialCoordinate(p_mesh)

# Parameters for Klein manifold
(a, n, m) = (2, 2, 1)

# Coordinate transformation
x = (a + cos(n*u/2.0) * sin(v) - sin(n*u/2.0) * sin(2*v)) * cos(m*u/2.0)
y = (a + cos(n*u/2.0) * sin(v) - sin(n*u/2.0) * sin(2*v)) * sin(m*u/2.0)
z = sin(n*u/2.0) * sin(v) + cos(n*u/2.0) * sin(2*v)

# Interpolate the coordinates into a vector field
V = VectorFunctionSpace(p_mesh, "CG", 3, dim=3)
coords = Function(V)
coords.interpolate(as_vector([x, y, z]))

# Make a mesh, using the topology of the base mesh,
# with coordinates from the supplied vector field
mesh = Mesh(coords)
```

With the domain constructed, we can now solve the heat equation.

```
# mesh = ...

V = FunctionSpace(mesh, "CG", 1)
u = Function(V, name="Solution")                      # u_n
u_prev = Function(V, name="PrevSolution")            # u_{n-1}
v = TestFunction(V)

# Initial condition
(x, y, z) = SpatialCoordinate(mesh)
g = sin(z) * cos(x)

T = 1          # final time
t = 0          # current time we are solving for
dt = Constant(0.02) # timestep

u.interpolate(g) # assign initial guess for solver
u_prev.assign(u) # assign initial condition to u_0

F = (
    1/dt * inner(u - u_prev, v)*dx
    + inner(grad(u), grad(v))*dx
)
```

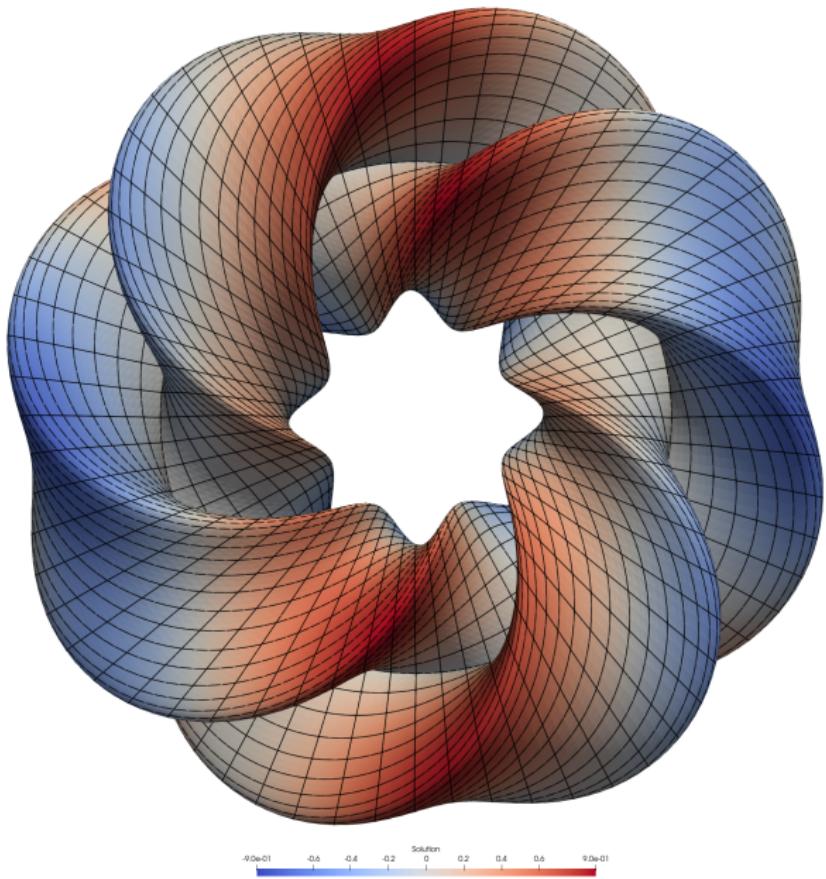
```
output = File("output/heat.pvd")
output.write(u, time=t)

# Main timestepping loop
while True:
    # Update the time we're solving for
    t += float(dt)
    print(f"Solving for time: {t:.2f}")

    solve(F == 0, u)

    # Now cycle the variables
    u_prev.assign(u)

    output.write(u, time=t)
```



Section 2

Time discretisation with irksome

Suppose now we wanted to use a higher-order scheme in time. We would have to substantially change the code; the PDE we solve at every timestep would differ.



Rob Kirby

Suppose now we wanted to use a higher-order scheme in time. We would have to substantially change the code; the PDE we solve at every timestep would differ.

Contrast this to the spatial discretisation. If we want to change the polynomial degree, we just have to change one line.



Rob Kirby

Suppose now we wanted to use a higher-order scheme in time. We would have to substantially change the code; the PDE we solve at every timestep would differ.

Contrast this to the spatial discretisation. If we want to change the polynomial degree, we just have to change one line.

Can we express our time discretisations with the same flexibility? Yes! Rob Kirby wrote the `irksome` package for exactly this.



Rob Kirby

For the variational form of the heat equation

$$(u_t, v) + (\nabla u, \nabla v) = (f, v),$$

we write

```
from irksome import Dt
F = (
    inner(Dt(u), v)*dx
+ inner(grad(u), grad(v))*dx
- inner(f, v)*dx
)
```

This lets us separate

the statement of the problem

from

the specification of how to discretise it.

In particular, with irksome, we can apply any Runge–Kutta scheme.

In particular, with irksome, we can apply any Runge–Kutta scheme.

Runge–Kutta schemes solve for s stages in each timestep, described by a Butcher tableau

$$\begin{array}{c|c} c & A \\ \hline & b \end{array}$$

In particular, with irksome, we can apply any Runge–Kutta scheme.

Runge–Kutta schemes solve for s stages in each timestep, described by a Butcher tableau

$$\begin{array}{c|c} c & A \\ \hline & b \end{array}$$

To discretise

$$u_t + h(u, t) = 0,$$

the associated Runge–Kutta step is to solve for the stages $\{k_i\}_{i=1}^s$

$$k_i + h(u_{n-1} + \Delta t \sum_{j=1}^s A_{ij} k_j, t_{n-1} + c_i \Delta t) = 0,$$

then to evaluate

$$u_n = u_{n-1} + \Delta t \sum_{i=1}^s b_i k_i.$$

Runge–Kutta schemes can be explicit, diagonally implicit, or fully implicit.

Explicit Runge–Kutta schemes

The Butcher matrix is strictly lower-triangular.

Runge–Kutta schemes can be explicit, diagonally implicit, or fully implicit.

Explicit Runge–Kutta schemes

The Butcher matrix is strictly lower-triangular.

Diagonally implicit Runge–Kutta schemes

The Butcher matrix is lower-triangular.

Runge–Kutta schemes can be explicit, diagonally implicit, or fully implicit.

Explicit Runge–Kutta schemes

The Butcher matrix is strictly lower-triangular.

Diagonally implicit Runge–Kutta schemes

The Butcher matrix is lower-triangular.

Fully implicit Runge–Kutta schemes

The Butcher matrix is not lower-triangular.

Fully implicit Runge–Kutta schemes ...

...can, however, be very powerful if you can develop solvers.

Family	order	generalises	A-stable?	L-stable?	symplectic?
Gauss–Legendre	$2s$	implicit midpoint	✓	✗	✓
RadauIIA	$2s - 1$	backward Euler	✓	✓	✗
LobattoIII A	$2s - 2$	Crank–Nicolson	✓	✗	✗

Fully implicit Runge–Kutta schemes ...

...can, however, be very powerful if you can develop solvers.

Family	order	generalises	A-stable?	L-stable?	symplectic?
Gauss–Legendre	$2s$	implicit midpoint	✓	✗	✓
RadauIIA	$2s - 1$	backward Euler	✓	✓	✗
LobattoIII A	$2s - 2$	Crank–Nicolson	✓	✗	✗

The heat equation is the prototypical stiff problem, where L-stability is important. So we will use RadauIIA.

```
# mesh = ...

V = FunctionSpace(mesh, "CG", 1)
u = Function(V, name="Solution")  # u_n
v = TestFunction(V)

# Initial condition
(x, y, z) = SpatialCoordinate(mesh)
g = sin(z) * cos(x)
u.interpolate(g)  # assign initial condition

F = (
    inner(Dt(u), v)*dx
    + inner(grad(u), grad(v))*dx
)
```

```
# Irksome setup. RadauIIA is L-stable. RadauIIA(1) is backward Euler.
tableau = RadauIIA(1)
factory = MeshConstant(mesh) # constants in space, not in time
dt = factory.Constant(0.02) # timestep
t = factory.Constant(0) # current time value
stepper = TimeStepper(F, tableau, t, dt, u)
T = 1 # final time

output = File("output/irksome_heat.pvd")
output.write(u, time=float(t))

# Main timestepping loop
while True:
    # Solve for the next timestep
    print(f"Solving for time: {float(t + dt):.2f}")
    stepper.advance() # derive and solve RK system
    t.assign(t + dt)

    output.write(u, time=float(t))
    if float(t) >= T:
        break
```

Challenge!

QIV.1. Solve the heat equation on the unit square $\Omega = (0, 1)^2$. Set the data so that the exact solution is

$$u_{\text{ex}}(x, y, t) = xy(1 - x)(1 - y) \sin(10\pi x) \cos(5\pi y) \left(1 + \frac{1}{2} \sin(4\pi t)\right).$$

Use CG_p and RadauIIA(p) for $p \geq 1$, with $\Delta t \propto h$.

Verify that

$$\|u(t=1) - u_h(t=1)\| \lesssim h^p.$$

Hint:

```
from ufl.algorithms.ad import expand_derivatives
f = expand_derivatives(diff(u_ex, t)) - div(grad(u_ex))
```

Solving PDEs with Firedrake: the Stokes and Navier–Stokes equations

Patrick E. Farrell



University of Oxford

February 2024

The Stokes equations describe the dynamics of slow-moving, viscous fluids.

We consider the stationary, incompressible equations:
find $u : \Omega \rightarrow \mathbb{R}^d$ and $p : \Omega \rightarrow \mathbb{R}$ such that

$$\begin{aligned}-\nabla \cdot (2\nu\epsilon(u) - p\mathbf{I}) &= f \quad \text{in } \Omega, \\ \nabla \cdot u &= 0 \quad \text{in } \Omega,\end{aligned}$$



George Gabriel Stokes

where $\nu > 0$ and $\epsilon(u) = \frac{1}{2} (\nabla u + (\nabla u)^T)$.

We impose boundary conditions

$$\begin{aligned}u &= 0 \quad \text{on } \partial\Omega_D, \\ -(2\nu\epsilon - p\mathbf{I}) \cdot n &= p_0 n \quad \text{on } \partial\Omega_N.\end{aligned}$$

If the viscosity ν varies with u , $\nu = \nu(u)$, the system is nonlinear.

The Stokes equations: variational formulation

Let $u \in V$ and $p \in Q$; the solution $w = (u, p) \in V \times Q =: W$.

Set $f = 0$. Multiplying by test functions $(v, q) \in W$, we get

$$\int_{\Omega} -\nabla \cdot (2\nu\epsilon(u)) \cdot v \, dx + \int_{\Omega} \nabla \cdot (pI) \cdot v \, dx = 0,$$

$$\int_{\Omega} q \nabla \cdot u \, dx = 0.$$

The Stokes equations: variational formulation

Let $u \in V$ and $p \in Q$; the solution $w = (u, p) \in V \times Q =: W$.

Set $f = 0$. Multiplying by test functions $(v, q) \in W$, we get

$$\int_{\Omega} -\nabla \cdot (2\nu\epsilon(u)) \cdot v \, dx + \int_{\Omega} \nabla \cdot (pI) \cdot v \, dx = 0,$$

$$\int_{\Omega} q \nabla \cdot u \, dx = 0.$$

Integrating by parts, using $I : \nabla v = \nabla \cdot v$, and negating, we find

$$\int_{\Omega} 2\nu\epsilon(u) : \nabla v \, dx - \int_{\Omega} p \nabla \cdot v \, dx - \int_{\partial\Omega} (2\nu\epsilon(u) - pI) \cdot n \cdot v \, ds = 0,$$

$$- \int_{\Omega} q \nabla \cdot u \, dx = 0.$$

The Stokes equations: variational formulation

Since $\epsilon(u)$ is a symmetric matrix, we can use the fact that

$$A : B = A : \text{sym}(B) \text{ if } A \text{ symmetric}$$

to make our form symmetric. We also substitute known boundary data:

$$\begin{aligned} \int_{\Omega} 2\nu\epsilon(u) : \epsilon(v) \, dx - \int_{\Omega} p\nabla \cdot v \, dx + \int_{\partial\Omega_N} p_0 n \cdot v \, ds &= 0, \\ - \int_{\Omega} q\nabla \cdot u \, dx &= 0. \end{aligned}$$

Finally, we can add variational statements together. Our problem becomes: find $(u, p) \in W$ such that

$$\int_{\Omega} 2\nu\epsilon(u) : \epsilon(v) \, dx - \int_{\Omega} p\nabla \cdot v \, dx - \int_{\Omega} q\nabla \cdot u \, dx = \int_{\Omega} f \cdot v \, dx - \int_{\partial\Omega_N} p_0 n \cdot v \, ds$$

for all $(v, q) \in W$.

Step by step: creating mixed function spaces

Since we are approximating multiple variables, we need a mixed function space.

```
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = MixedFunctionSpace([V, Q])
```

Step by step: creating mixed function spaces

Since we are approximating multiple variables, we need a mixed function space.

```
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = MixedFunctionSpace([V, Q])
```

You can define functions on mixed spaces and split into components:

```
w = Function(W)
(u, p) = split(w)
```

... and test functions:

```
y = TestFunction(W)
(v, q) = split(y)
```

Step by step: creating mixed function spaces

Since we are approximating multiple variables, we need a mixed function space.

```
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = MixedFunctionSpace([V, Q])
```

You can define functions on mixed spaces and split into components:

```
w = Function(W)
(u, p) = split(w)
```

... and test functions:

```
y = TestFunction(W)
(v, q) = split(y)
```

The choice of mixed function space is **crucial** (inf-sup conditions).

Step by step: defining a boundary condition on a subspace

The subspaces of W can be retrieved using `sub`:

```
W0 = W.sub(0)
```

You can impose boundary conditions on subspaces:

```
bc = DirichletBC(W.sub(0), Constant((1, 1)), 1)
bc = DirichletBC(W.sub(0).sub(1), 0, 2)
```

Stokes: defining the variational form

Given

```
w = Function(W)
(u, p) = split(w)
(v, q) = split(TestFunction(W))
p0 = ...; nu = ...; n = FacetNormal(mesh)
```

we can define the form F with

```
F = (
    inner(2*nu*sym(grad(u)), sym(grad(v)))*dx
    - inner(div(u), q)*dx
    - inner(div(v), p)*dx
    + inner(p0*n, v)*ds
)
```

Lastly, there are two ways to access components of a mixed function space.

This creates a *view*:

```
(u, p) = split(w)
```

u and p are symbolic pointers to components of w.

Use this in your mixed form.

You can also view them as Functions in their own right:

```
(u_, p_) = w.subfunctions
```

Use this to plot.

```
from firedrake import *
from netgen.occ import *
from ngsPETSc import NetgenHierarchy

# Mesh: a hole removed from a rectangle
disk = WorkPlane(Axes((0,0,0), n=Z, h=X)).Circle(1).Face()
rect = WorkPlane(Axes((-3,-3,0), n=Z, h=X)).Rectangle(13, 6).Face()
domain = rect - disk

# Label boundaries
domain.edges.name = "wall" # all default to wall
domain.edges.Min(X).name = "inlet"
domain.edges.Max(X).name = "outlet"
geo = OCCGeometry(domain, dim=2)

ngmesh = geo.GenerateMesh(maxh=1)
mh = NetgenHierarchy(ngmesh, 2, order=2)
mesh = mh[-1]

walls = [i + 1 for (i, name) in
         enumerate(ngmesh.GetRegionNames(codim=1)) if name == "wall"]
```

```
n = FacetNormal(mesh)
(x, y) = SpatialCoordinate(mesh)

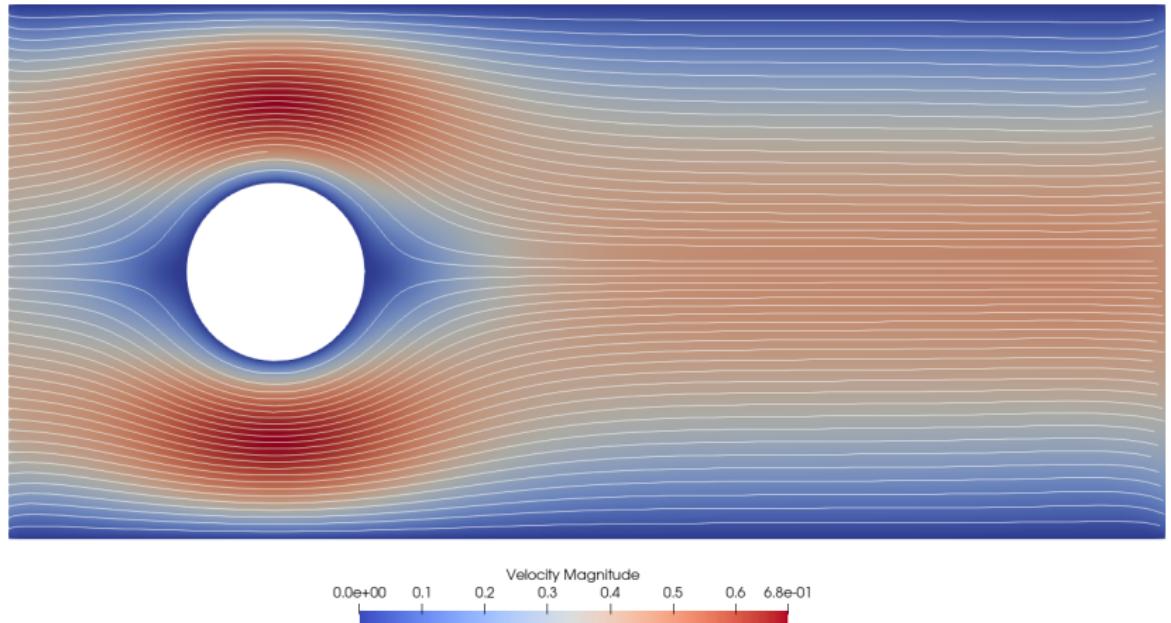
# Define Taylor--Hood function space W
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = MixedFunctionSpace([V, Q])

# Define Function and TestFunction(s)
w = Function(W, name="Solution")
(u, p) = split(w)
(v, q) = split(TestFunction(W))

# Define viscosity and bcs
nu = Constant(0.2)
p0 = 10/13 - x/13 # 1 at left, 0 at right
bcs = DirichletBC(W.sub(0), Constant((0, 0)), walls)
```

```
# Define variational form
F = (
    inner(2*nu*sym(grad(u)), sym(grad(v)))*dx
    - div(u)*q*dx
    - div(v)*p*dx
    + p0*dot(v,n)*ds
)
# Solve problem
solve(F == 0, w, bcs)

# Save solutions
(u_, p_) = w.subfunctions
u_.rename("Velocity")
p_.rename("Pressure")
File("output/stokes.pvd").write(u_, p_)
```



Challenge!

QV.1. Solve the Stokes problem on the same domain, but with a nonlinear (power-law) rheology:

$$\nu(u) = \nu_c (1 + \epsilon(u) : \epsilon(u))^{k-2},$$

with parameters

$$\nu_c = 0.2, \quad k = 4.$$

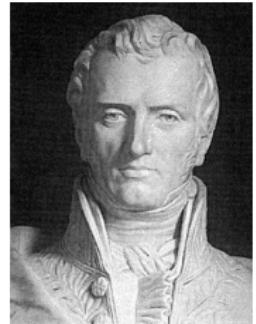
Now let us consider the stationary, incompressible Navier–Stokes equations: find $u : \Omega \rightarrow \mathbb{R}^d$ and $p : \Omega \rightarrow \mathbb{R}$ such that

$$\begin{aligned}-\nabla \cdot (2\nu\epsilon(u) - p\mathbf{I} - u \otimes u) &= f \quad \text{in } \Omega, \\ \nabla \cdot u &= 0 \quad \text{in } \Omega,\end{aligned}$$

where $\nu > 0$ and $\epsilon(u) = \frac{1}{2} (\nabla u + (\nabla u)^T)$.

The advection term, $\nabla \cdot (u \otimes u)$, models the transport of the fluid by itself.

This term is nonlinear, independent of rheology.



Claude-Louis Navier

Let us consider the weak form of just this term.

Multiplying by our test function v , we get

$$\int_{\Omega} \nabla \cdot (u \otimes u) \cdot v \, dx = - \int_{\Omega} (u \otimes u) : \epsilon(v) \, dx + \int_{\partial\Omega} (u \otimes u) n \cdot v \, ds$$

where the surface integral vanishes on Dirichlet boundaries.

Let us consider the weak form of just this term.

Multiplying by our test function v , we get

$$\int_{\Omega} \nabla \cdot (u \otimes u) \cdot v \, dx = - \int_{\Omega} (u \otimes u) : \epsilon(v) \, dx + \int_{\partial\Omega} (u \otimes u)n \cdot v \, ds$$

where the surface integral vanishes on Dirichlet boundaries.

We therefore add to our Stokes weak form:

```
F = (
    ...
    - inner(outer(u, u), sym(grad(v)))*dx
    + inner(dot(outer(u, u), n), v)*ds
)
```

Challenge!

QV.2. Using the same data and rheology as QV.1, solve the non-Newtonian Navier–Stokes problem.

Solving PDEs with Firedrake: hyperelasticity

Patrick E. Farrell



University of Oxford

February 2024

We will build a solver for a nontrivial problem: compressible hyperelasticity.

We will build a solver for a nontrivial problem: compressible hyperelasticity.

Like linear elasticity, these equations describe how a structure deforms under a load. We set Ω to be the undeformed configuration, and solve for a displacement $u : \Omega \rightarrow \mathbb{R}^d$ that describes how each point $X \in \Omega$ maps to the deformed configuration:

$$x(X) = X + u(X).$$

We will build a solver for a nontrivial problem: compressible hyperelasticity.

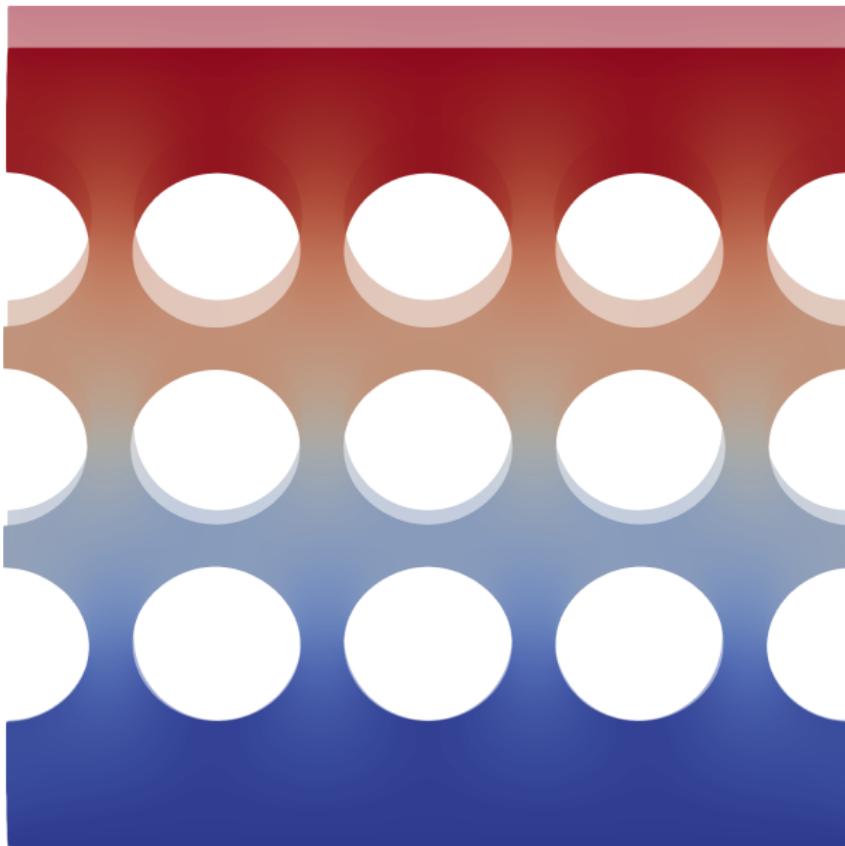
Like linear elasticity, these equations describe how a structure deforms under a load. We set Ω to be the undeformed configuration, and solve for a displacement $u : \Omega \rightarrow \mathbb{R}^d$ that describes how each point $X \in \Omega$ maps to the deformed configuration:

$$x(X) = X + u(X).$$

Unlike linear elasticity, hyperelasticity is more realistic because

- ▶ (constitutive nonlinearity) the stress-strain curve is not necessarily linear;
- ▶ (geometric nonlinearity) the displacements are not necessarily small.

The equations are thus nonlinear.



Challenge!

In this exercise, you will write your own code from scratch.

Good news!

I will tell you everything you need to know.

Section 2

Minimisation and saddle point problems

Many problems can be cast in an optimisation framework.

For example, the Poisson equation arises as the minimisation of the Dirichlet energy

$$J(u) = \frac{1}{2} \int_{\Omega} \nabla u \cdot \nabla u \, dx - \int_{\Omega} fu \, dx.$$

Many problems can be cast in an optimisation framework.

For example, the Poisson equation arises as the minimisation of the Dirichlet energy

$$J(u) = \frac{1}{2} \int_{\Omega} \nabla u \cdot \nabla u \, dx - \int_{\Omega} fu \, dx.$$

We can see this by taking its Fréchet derivative and setting it to zero:

$$\begin{aligned} J_u(u; v) &:= \lim_{\epsilon \rightarrow 0} \frac{J(u + \epsilon v) - J(u)}{\epsilon} \\ &= \lim_{\epsilon \rightarrow 0} \frac{1}{\epsilon} \left(\epsilon \int_{\Omega} \nabla u \cdot \nabla v \, dx + \epsilon^2 \int_{\Omega} \nabla v \cdot \nabla v \, dx - \epsilon \int_{\Omega} fv \, dx \right) \\ &= \int_{\Omega} \nabla u \cdot \nabla v \, dx - \int_{\Omega} fv \, dx = 0, \end{aligned}$$

the weak statement of the Poisson equation.

We can get Firedrake to do this calculation for us:

```
# Functional to optimise
J = (
    0.5 * inner(grad(u), grad(u))*dx
    -
    inner(f, u)*dx
)
# Calculate the optimality condition (equation to solve)
F = derivative(J, u)
```

We can get Firedrake to do this calculation for us:

```
# Functional to optimise
J = (
    0.5 * inner(grad(u), grad(u))*dx
    -      inner(f, u)*dx
)
# Calculate the optimality condition (equation to solve)
F = derivative(J, u)
```

Firedrake uses `derivative` inside `solve` to calculate the Jacobian.

Section 3

Hyperelasticity energy functional

Our *dramatis personae*:

- ▶ $\Omega \subset \mathbb{R}^d$, the domain;



Ronald Rivlin

Our *dramatis personae*:

- ▶ $\Omega \subset \mathbb{R}^d$, the domain;
- ▶ $u : \Omega \rightarrow \mathbb{R}^d$, the displacement;



Ronald Rivlin

Our *dramatis personae*:

- ▶ $\Omega \subset \mathbb{R}^d$, the domain;
- ▶ $u : \Omega \rightarrow \mathbb{R}^d$, the displacement;
- ▶ $F = I + \nabla u$, the deformation gradient;



Ronald Rivlin

Our *dramatis personae*:

- ▶ $\Omega \subset \mathbb{R}^d$, the domain;
- ▶ $u : \Omega \rightarrow \mathbb{R}^d$, the displacement;
- ▶ $F = I + \nabla u$, the deformation gradient;
- ▶ $C = F^\top F$, the right Cauchy–Green tensor;



Ronald Rivlin

Our *dramatis personae*:

- ▶ $\Omega \subset \mathbb{R}^d$, the domain;
- ▶ $u : \Omega \rightarrow \mathbb{R}^d$, the displacement;
- ▶ $F = I + \nabla u$, the deformation gradient;
- ▶ $C = F^\top F$, the right Cauchy–Green tensor;
- ▶ $I_c = \text{tr}(C), J = \det(C)$, invariants;



Ronald Rivlin

Our *dramatis personae*:

- ▶ $\Omega \subset \mathbb{R}^d$, the domain;
- ▶ $u : \Omega \rightarrow \mathbb{R}^d$, the displacement;
- ▶ $F = I + \nabla u$, the deformation gradient;
- ▶ $C = F^\top F$, the right Cauchy–Green tensor;
- ▶ $I_c = \text{tr}(C), J = \det(C)$, invariants;
- ▶ μ, λ , Lamé parameters.



Ronald Rivlin

Our *dramatis personae*:

- ▶ $\Omega \subset \mathbb{R}^d$, the domain;
- ▶ $u : \Omega \rightarrow \mathbb{R}^d$, the displacement;
- ▶ $F = I + \nabla u$, the deformation gradient;
- ▶ $C = F^\top F$, the right Cauchy–Green tensor;
- ▶ $I_c = \text{tr}(C)$, $J = \det(C)$, invariants;
- ▶ μ, λ , Lamé parameters.



Ronald Rivlin

With these, we form the compressible neo-Hookean energy:

$$J(u) = \int_{\Omega} \frac{\mu}{2} (I_c - d) - \mu \ln J + \frac{\lambda}{2} (\ln J)^2 \, dx.$$

Stating this in Firedrake:

```
d = mesh.geometric_dimension()
I = Identity(d)
F = I + grad(u)
C = F.T * F
I_c = tr(C)
J = det(C)
```

Section 4

Continuation

Continuation is an extremely powerful algorithm for solving difficult nonlinear problems.

Idea: construct a good initial guess by solving an easier problem.

Continuation

- ▶ Solve the problem for easy parameter value.
- ▶ While not finished:
 - ▶ Use solution for previous parameter as initial guess for next parameter.
 - ▶ Increment parameter.

To do continuation in Firedrake, update the parameter in a loop and solve:

```
strain = Constant(0) # placeholder Constant
# Use the strain as our boundary condition value:
bcs = [...,
        DirichletBC(V.sub(0).sub(1), strain, top),
        ...]

strains = ...
for strain_ in strains:
    strain.assign(strain_) # update parameter value
    solve(F == 0, u, bcs) # solve for next parameter
```

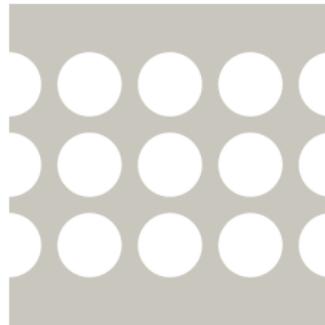
Challenge!

Solve the equations of hyperelasticity on the domain

$$\Omega = (0, 1)^2 \setminus \left(\bigcup_{ij} D_{ij} \right),$$

where $i \in \{1, \dots, 4\}$, $j \in \{1, \dots, 6\}$, and

$$D_{ij} = \left\{ (x, y) \in \mathbb{R}^2 : \left(x - \frac{j-1}{4} \right)^2 + \left(y - \frac{i}{4} \right)^2 \leq 0.1^2 \right\}.$$



Challenge!

Solve the problem with $\mu = 4 \times 10^5$, $\lambda = 6 \times 10^5$, and boundary conditions

$$\begin{aligned} u &= (0, 0) \quad \text{on } \{y = 0\}, \\ u &= (0, -s) \quad \text{on } \{y = 1\}, \end{aligned}$$

for $s = 0.1$.

Apply natural (i.e. do-nothing, stress-free) boundary conditions on all other boundaries.

Hint: you will probably need to employ continuation.

Solving PDEs with Firedrake: variational inequalities

Patrick E. Farrell



University of Oxford

February 2024

Suppose we wish to solve

$$\min_{x \in \mathbb{R}} f(x)$$

where $f \in C^1(\mathbb{R}, \mathbb{R})$ has a lower bound.

Suppose we wish to solve

$$\min_{x \in \mathbb{R}} f(x)$$

where $f \in C^1(\mathbb{R}, \mathbb{R})$ has a lower bound.

We know from school that a minimiser must satisfy

$$f'(x) = 0$$

which we could usefully rewrite as

$$f'(x) \cdot +1 \geq 0,$$

$$f'(x) \cdot -1 \geq 0,$$

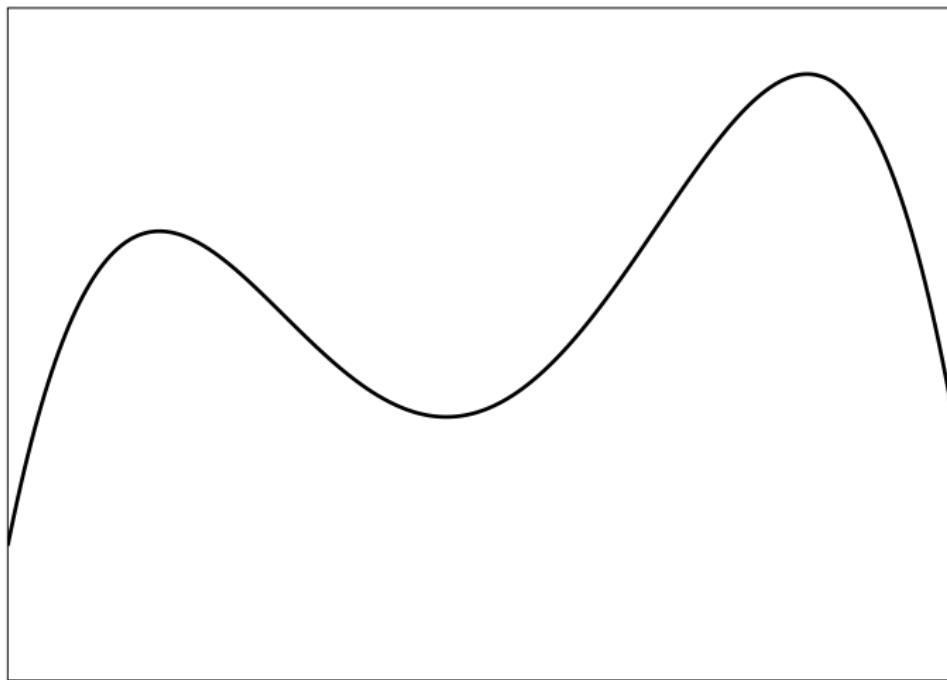
saying that in both feasible directions the function increases, to first-order.

Now let's suppose we want to solve

$$\min_{x \in [a,b]} f(x)$$

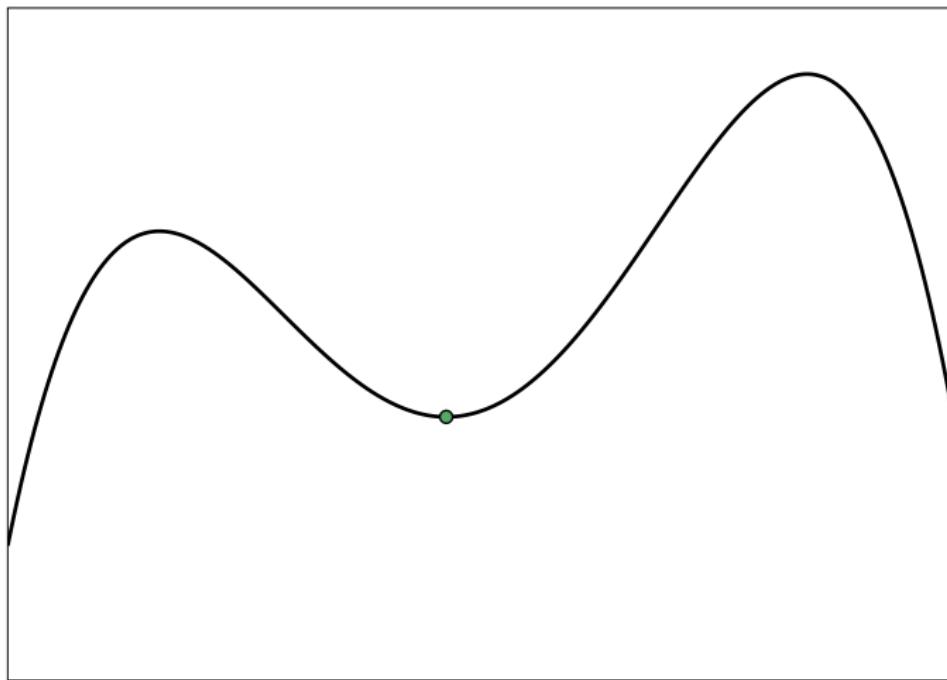
Now let's suppose we want to solve

$$\min_{x \in [a,b]} f(x)$$



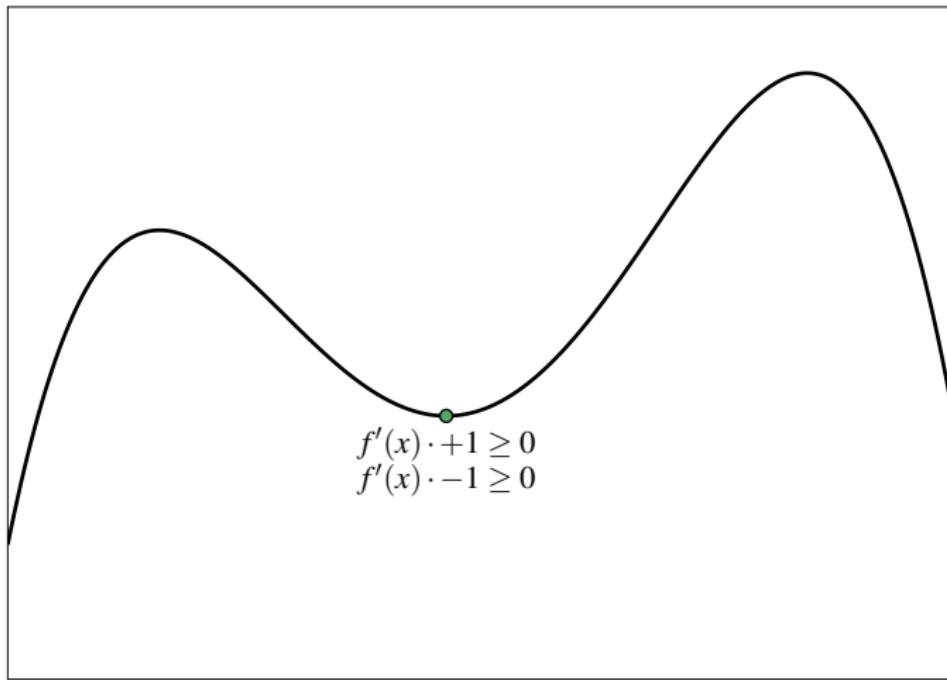
Now let's suppose we want to solve

$$\min_{x \in [a,b]} f(x)$$



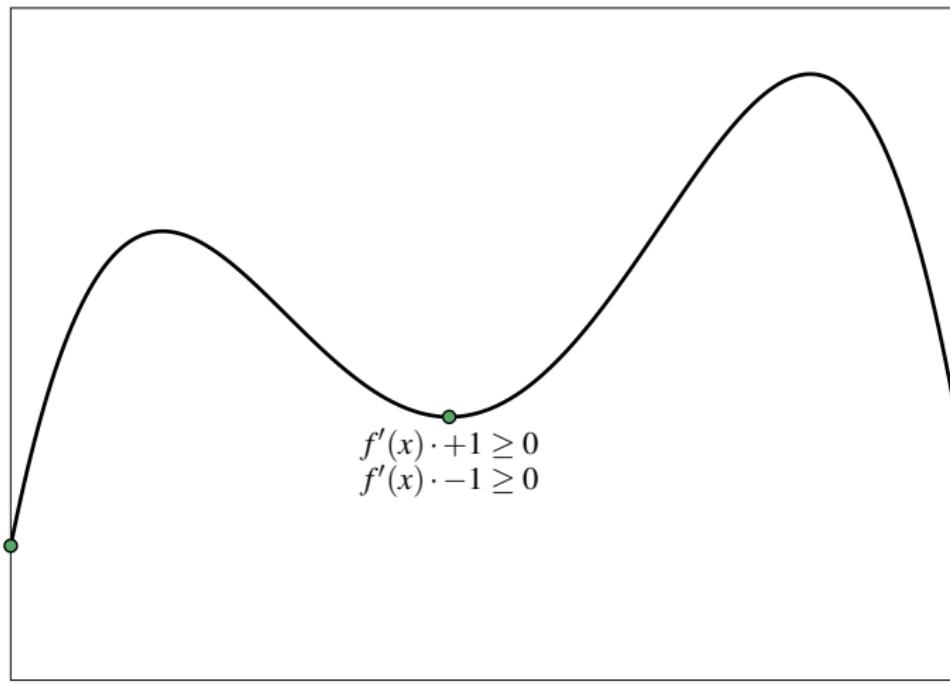
Now let's suppose we want to solve

$$\min_{x \in [a,b]} f(x)$$



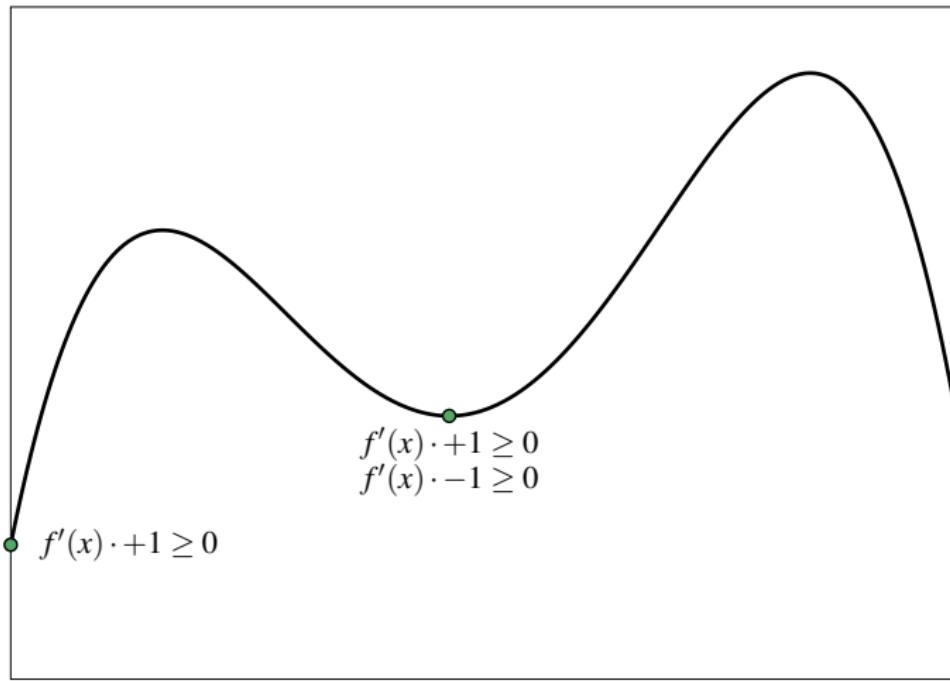
Now let's suppose we want to solve

$$\min_{x \in [a,b]} f(x)$$



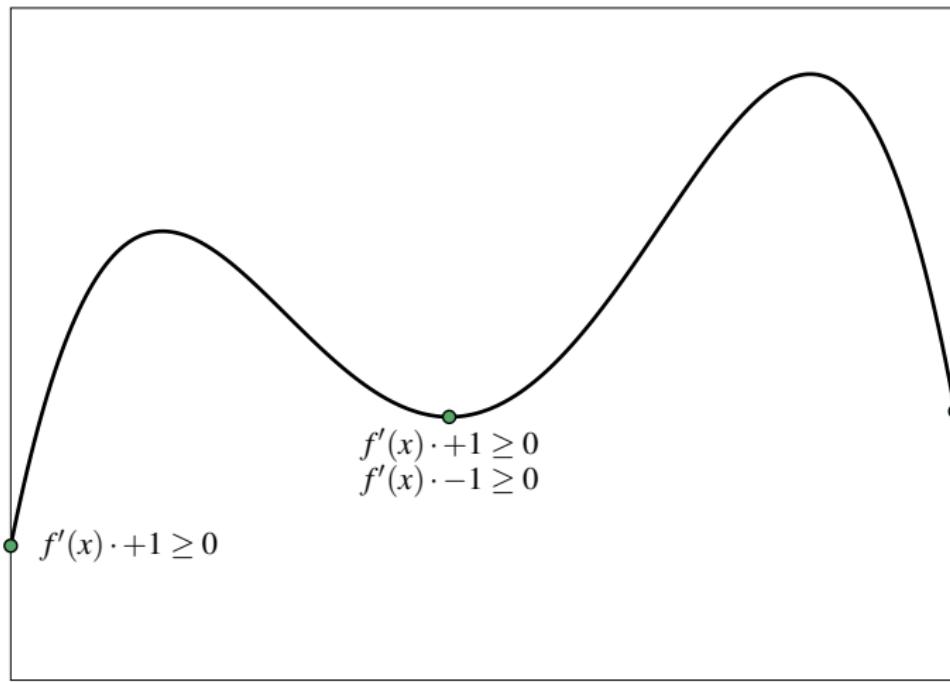
Now let's suppose we want to solve

$$\min_{x \in [a,b]} f(x)$$



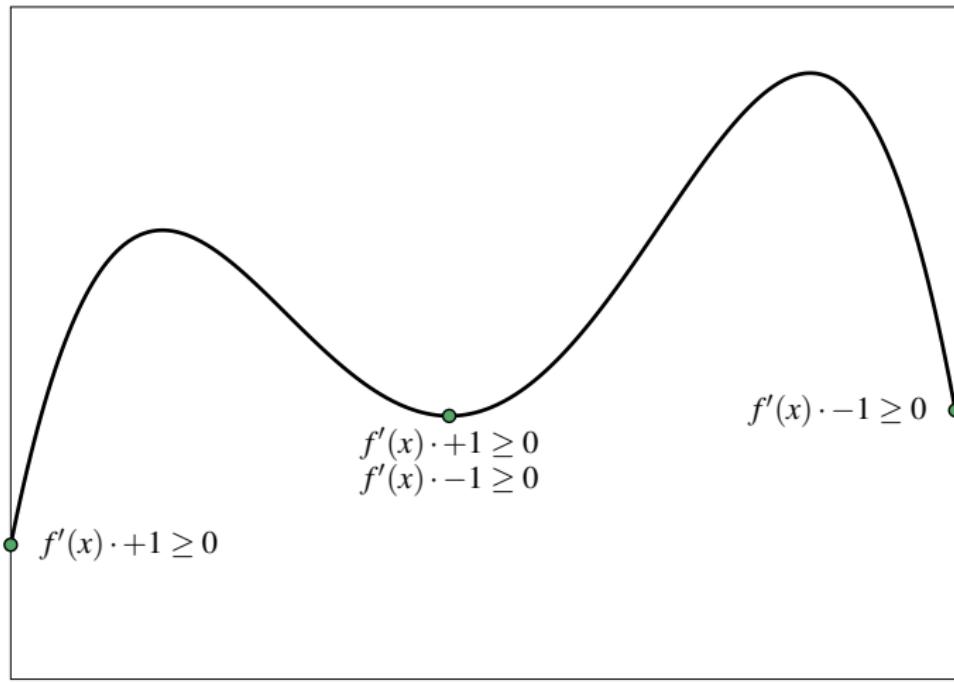
Now let's suppose we want to solve

$$\min_{x \in [a,b]} f(x)$$



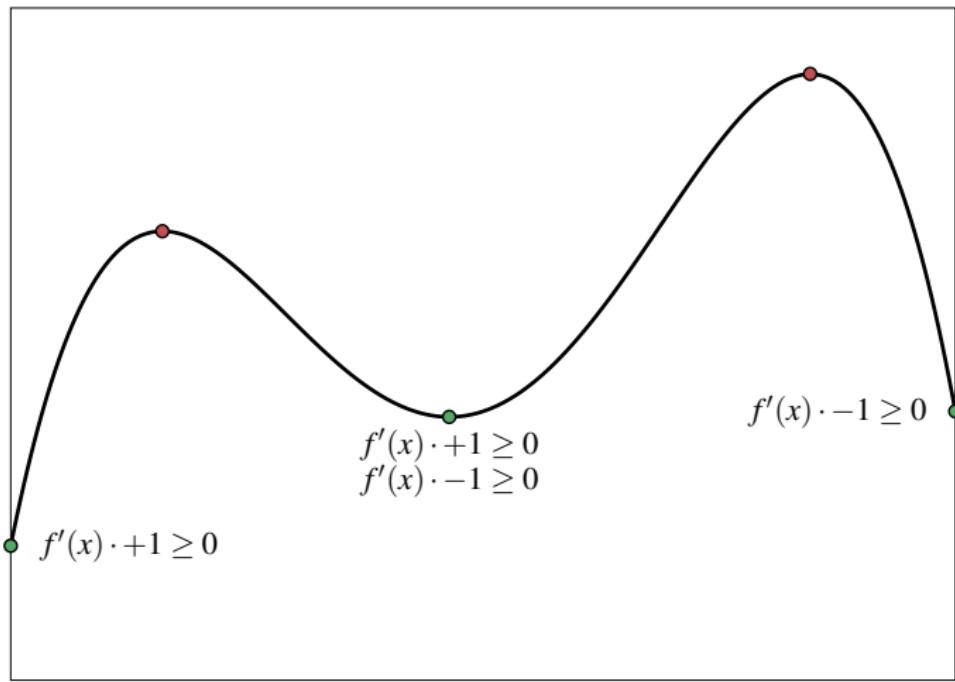
Now let's suppose we want to solve

$$\min_{x \in [a,b]} f(x)$$



Now let's suppose we want to solve

$$\min_{x \in [a,b]} f(x)$$



So we want to capture the notion that

$$f'(x) \cdot d \geq 0 \text{ for all feasible } d.$$

How do we construct all feasible directions?

So we want to capture the notion that

$$f'(x) \cdot d \geq 0 \text{ for all feasible } d.$$

How do we construct all feasible directions?

Let K be a closed, convex set. Then all feasible directions from a base point $u \in K$ are given by

$$v - u \text{ for all } v \in K.$$

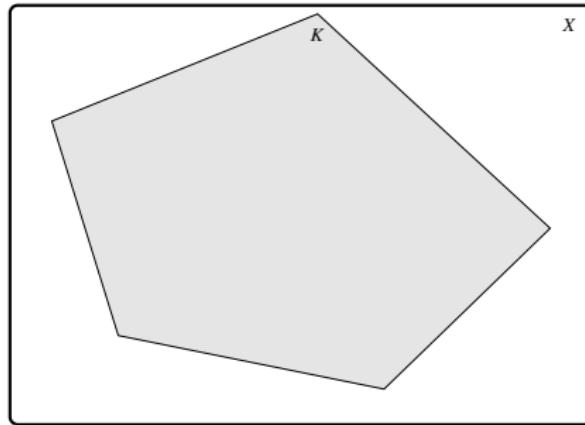
So we want to capture the notion that

$$f'(x) \cdot d \geq 0 \text{ for all feasible } d.$$

How do we construct all feasible directions?

Let K be a closed, convex set. Then all feasible directions from a base point $u \in K$ are given by

$$v - u \text{ for all } v \in K.$$



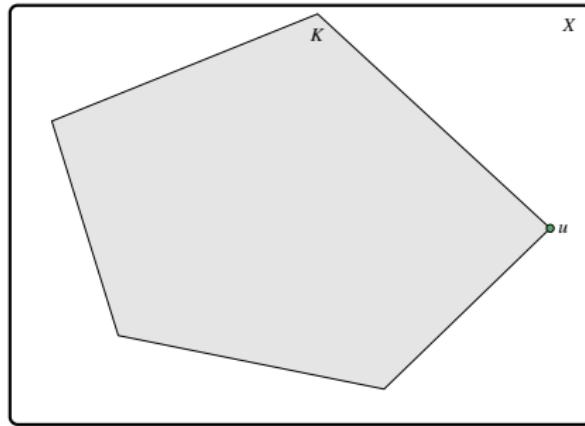
So we want to capture the notion that

$$f'(x) \cdot d \geq 0 \text{ for all feasible } d.$$

How do we construct all feasible directions?

Let K be a closed, convex set. Then all feasible directions from a base point $u \in K$ are given by

$$v - u \text{ for all } v \in K.$$



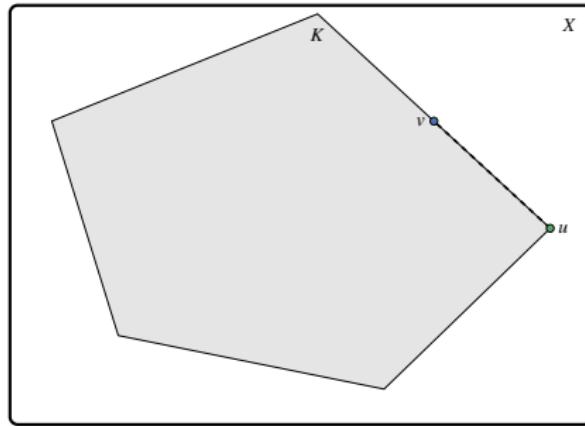
So we want to capture the notion that

$$f'(x) \cdot d \geq 0 \text{ for all feasible } d.$$

How do we construct all feasible directions?

Let K be a closed, convex set. Then all feasible directions from a base point $u \in K$ are given by

$$v - u \text{ for all } v \in K.$$



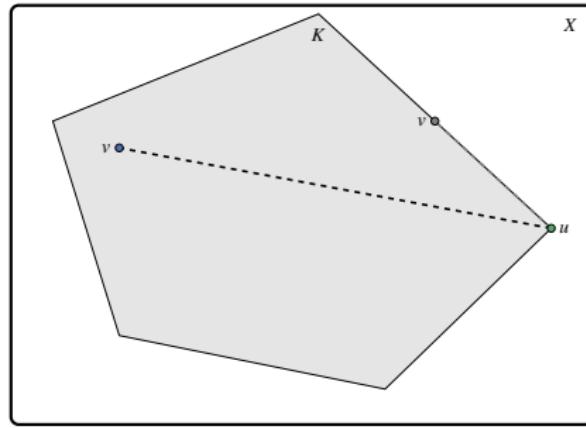
So we want to capture the notion that

$$f'(x) \cdot d \geq 0 \text{ for all feasible } d.$$

How do we construct all feasible directions?

Let K be a closed, convex set. Then all feasible directions from a base point $u \in K$ are given by

$$v - u \text{ for all } v \in K.$$



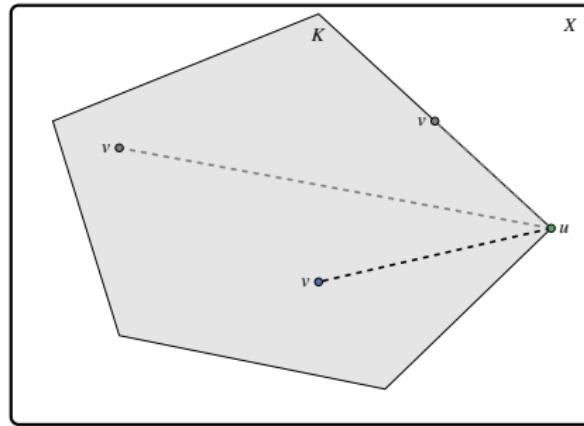
So we want to capture the notion that

$$f'(x) \cdot d \geq 0 \text{ for all feasible } d.$$

How do we construct all feasible directions?

Let K be a closed, convex set. Then all feasible directions from a base point $u \in K$ are given by

$$v - u \text{ for all } v \in K.$$



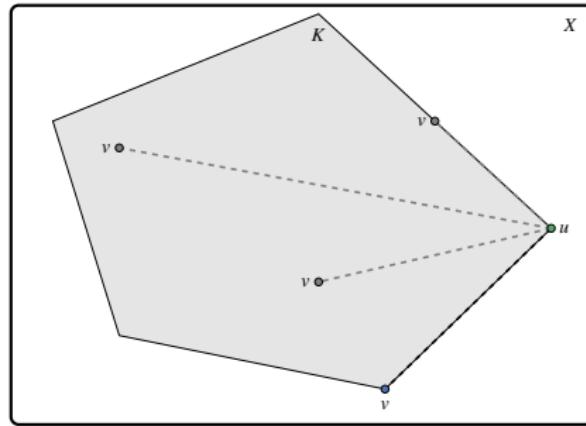
So we want to capture the notion that

$$f'(x) \cdot d \geq 0 \text{ for all feasible } d.$$

How do we construct all feasible directions?

Let K be a closed, convex set. Then all feasible directions from a base point $u \in K$ are given by

$$v - u \text{ for all } v \in K.$$



VI(Q, K)

Let X be a real reflexive Banach space, $K \subset X$ a closed convex subset, and $Q : K \rightarrow X^*$. The task is to

$$\text{find } u^* \in K \text{ such that } \langle Q(u^*), v - u^* \rangle \geq 0 \text{ for all } v \in K.$$

The optimality condition for the problem

$$\min_{x \in K} f(x)$$

is

$$\text{VI}(Df, K),$$

where Df is the Jacobian.

Some facts about variational inequalities:

Generalisation of equations

$\text{VI}(Q, K)$ is an equation if K is a vector space.

Some facts about variational inequalities:

Generalisation of equations

$\text{VI}(Q, K)$ is an equation if K is a vector space.

Applications

They are ubiquitous, in contact, economics, thermodynamics,

Some facts about variational inequalities:

Generalisation of equations

$\text{VI}(Q, K)$ is an equation if K is a vector space.

Applications

They are ubiquitous, in contact, economics, thermodynamics,

Multiple solutions

Like nonlinear equations, they can often support multiple solutions.

Some facts about variational inequalities:

Generalisation of equations

$\text{VI}(Q, K)$ is an equation if K is a vector space.

Applications

They are ubiquitous, in contact, economics, thermodynamics,

Multiple solutions

Like nonlinear equations, they can often support multiple solutions.

Computational challenge

They can be quite hard to solve!

Section 2

The obstacle problem

Suppose an elastic membrane is attached to a flat wire frame which encloses a region Ω of the plane. Suppose this membrane is subject to a distributed load $f(x, y)$. Then for small displacements the equilibrium position $z = u(x, y)$ satisfies

$$u = \arg \min_{v \in H_0^1(\Omega)} J(v) := \frac{1}{2} \int_{\Omega} \nabla v \cdot \nabla v \, dx - \int_{\Omega} fv \, dx.$$

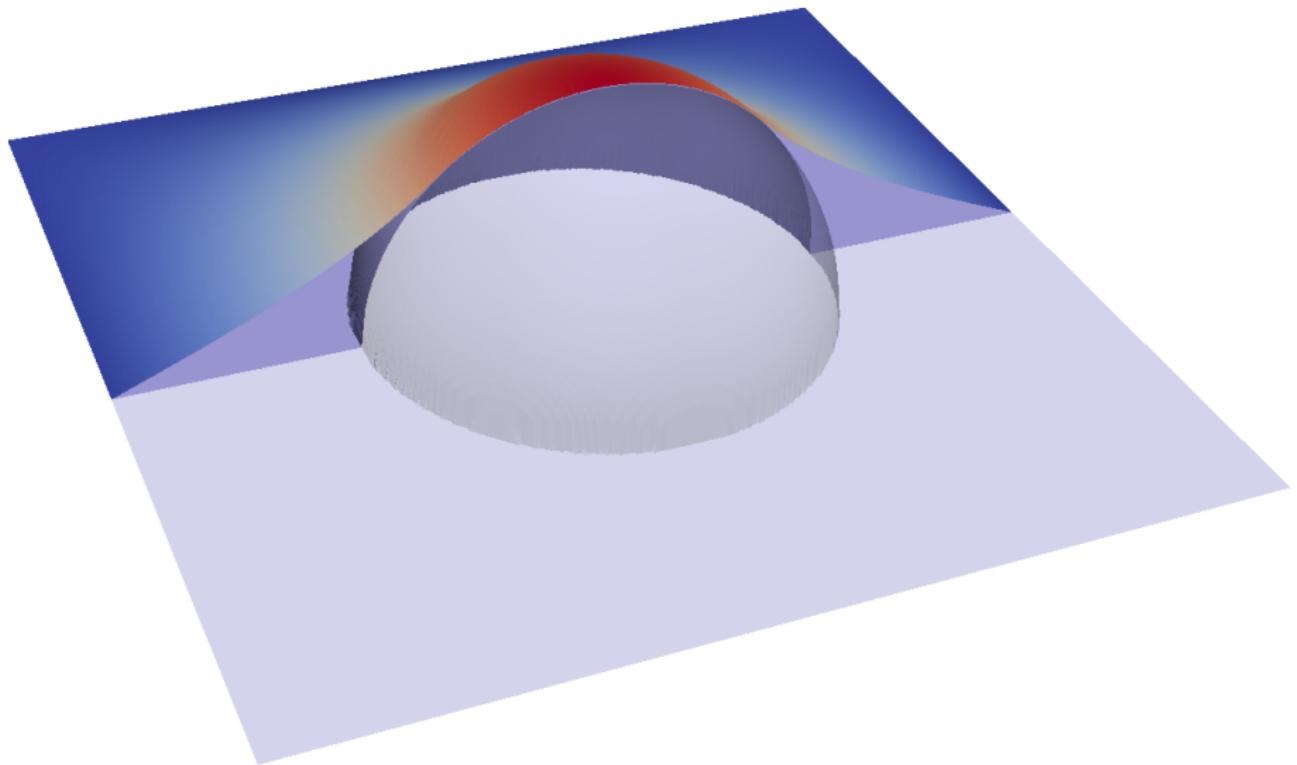
Suppose an elastic membrane is attached to a flat wire frame which encloses a region Ω of the plane. Suppose this membrane is subject to a distributed load $f(x, y)$. Then for small displacements the equilibrium position $z = u(x, y)$ satisfies

$$u = \arg \min_{v \in H_0^1(\Omega)} J(v) := \frac{1}{2} \int_{\Omega} \nabla v \cdot \nabla v \, dx - \int_{\Omega} fv \, dx.$$

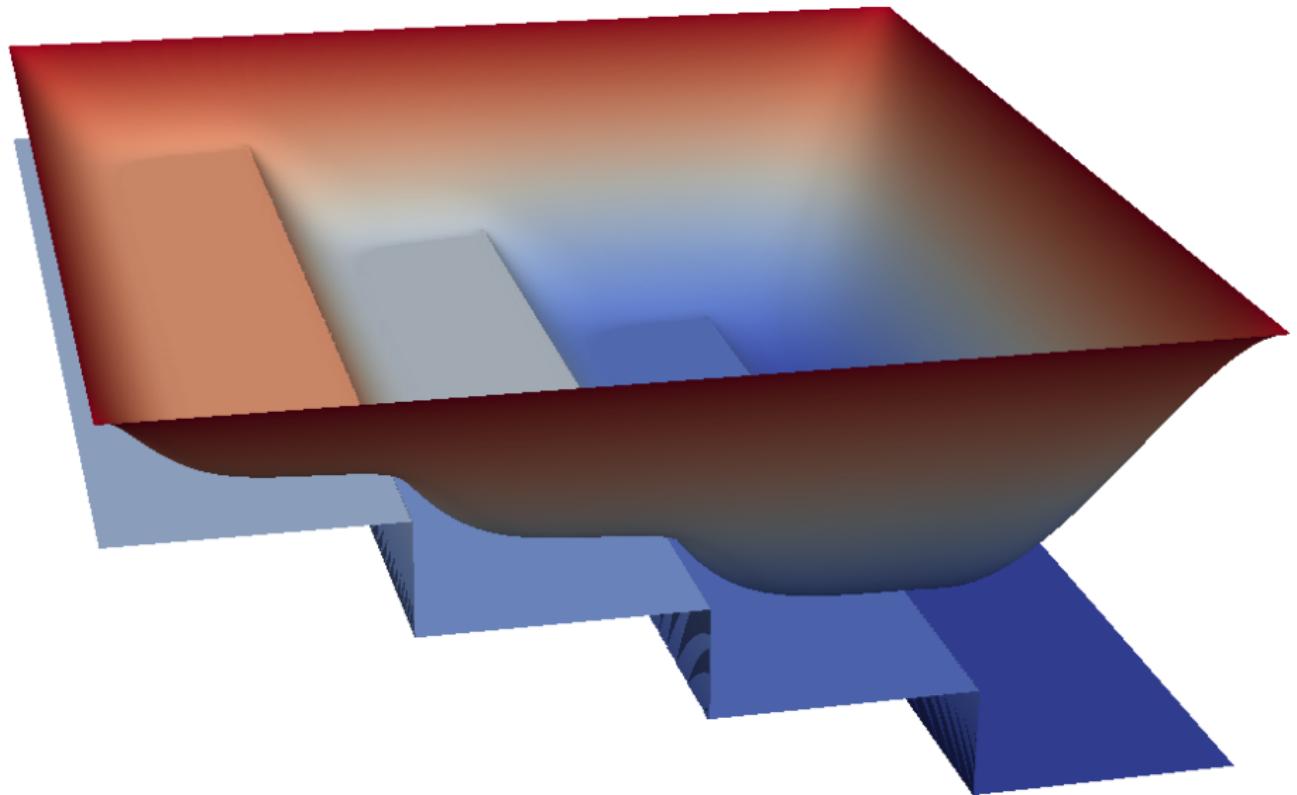
Now suppose that *an obstacle is placed underneath the membrane*. The obstacle $z = \psi(x, y)$ is square-integrable with $\psi|_{\partial\Omega} \leq 0$. With this new constraint, the problem becomes

$$\begin{aligned} u &= \arg \min_{v \in H_0^1(\Omega)} && J(v) \\ &\text{subject to} && v \geq \psi. \end{aligned}$$

Obstacle problem



Obstacle problem



Defining

$$K_\psi = \{v \in H_0^1(\Omega) \mid v \geq \psi\},$$

our problem is the variational inequality

$$\text{VI}(J', K_\psi).$$

A key tool in solving variational inequalities is to reformulate them as a system of equations.

A key tool in solving variational inequalities is to reformulate them as a system of equations.

For example, $\text{VI}(Q, K)$ with

$$K = \{x \in \mathbb{R} : x \geq 0\}$$

is equivalent to

$$S(x) := \sqrt{x^2 + [Q(x)]^2} - x - Q(x) = 0.$$

A key tool in solving variational inequalities is to reformulate them as a system of equations.

For example, $\text{VI}(Q, K)$ with

$$K = \{x \in \mathbb{R} : x \geq 0\}$$

is equivalent to

$$S(x) := \sqrt{x^2 + [Q(x)]^2} - x - Q(x) = 0.$$

The price we pay ...

...is that S is not smooth (C^1).

Good news

S is just smooth enough to define a Newton-type method with superlinear convergence.



Michael Hintermüller



Michael Ulbrich

Good news

S is just smooth enough to define a Newton-type method with superlinear convergence.

Semismooth Newton works just like normal:

$$u_{i+1} = u_i - [H(u_i)]^{-1} S(u_i),$$



Michael Hintermüller

where H is the *Newton derivative*.

This algorithm usually converges superlinearly.



Michael Ulbrich

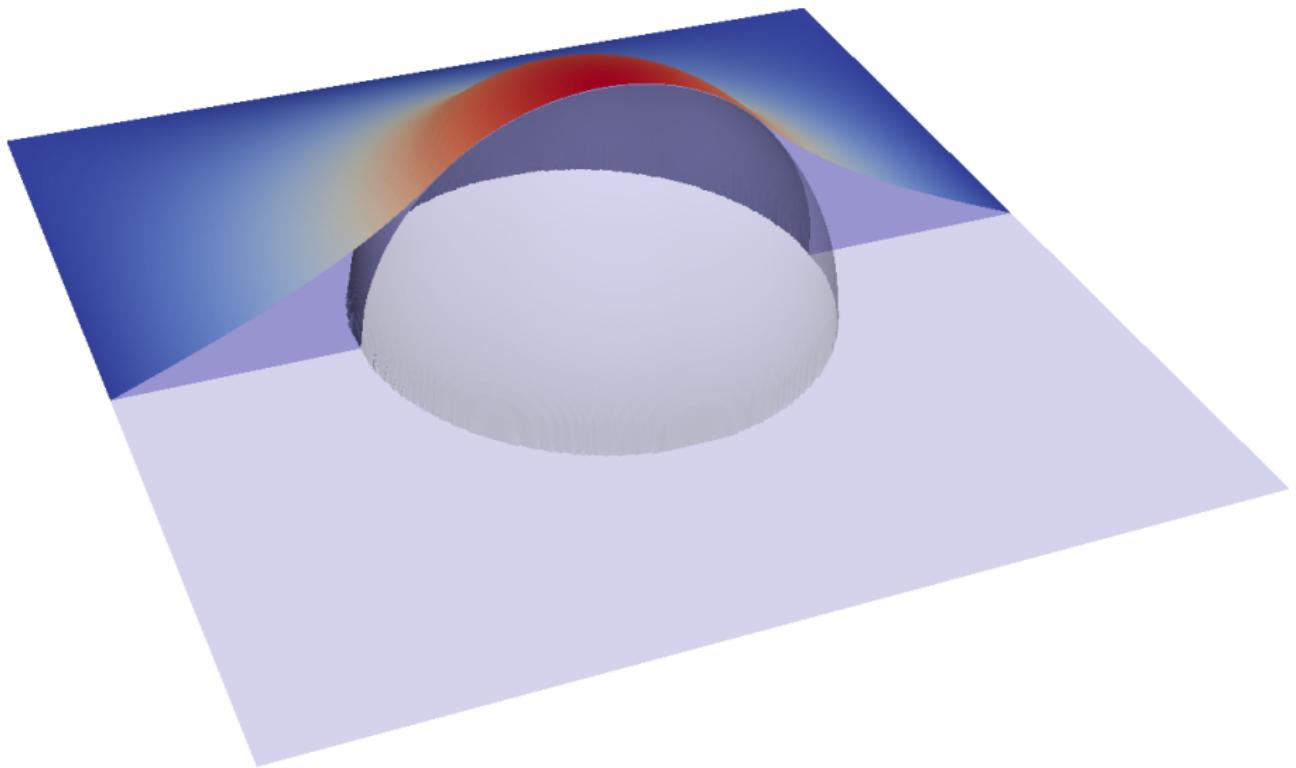
```
from firedrake import *

mesh = UnitSquareMesh(64, 64, quadrilateral=True)
V = FunctionSpace(mesh, "CG", 1)

u = Function(V, name="Solution")
J = 0.5 * inner(grad(u), grad(u))*dx
F = derivative(J, u)
bcs = DirichletBC(V, 0, "on_boundary")

# Define obstacle
(x, y) = SpatialCoordinate(mesh)
r = 0.25 # radius
psi = 4*(r**2 - (x - 0.5)**2 - (y - 0.5)**2)
obstacle = Function(V, name="Obstacle")
obstacle.interpolate(conditional(lt(psi, 0), 0, psi))
```

```
# We have to use a slightly lower-level interface:  
# under the hood, solve( $F == 0$ ) makes these objects  
sp = {"snes_type": "vinewtonrsls",  
      "snes_monitor": None}  
problem = NonlinearVariationalProblem(F, u, bcs)  
solver = NonlinearVariationalSolver(problem, solver_parameters=sp)  
  
# Pass bounds in call to solver.solve: (lower bound, upper bound)  
# Unfortunately we need to pass an upper bound, also.  
upper = Function(V).interpolate(Constant(1e10))  
solver.solve(bounds=(obstacle, upper))
```



Challenge!

QVII.1. Solve the Poisson obstacle problem with $\Omega = (0, 1)^2$, $f = -10$ and

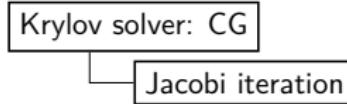
$$\psi(x, y) = \begin{cases} -0.2 & \text{if } x \in [0, 0.25), \\ -0.4 & \text{if } x \in [0.25, 0.5), \\ -0.6 & \text{if } x \in [0.5, 0.75), \\ -0.8 & \text{if } x \in [0.75, 1]. \end{cases}$$

Tabulate the number of semismooth Newton iterations required as a function of mesh size.

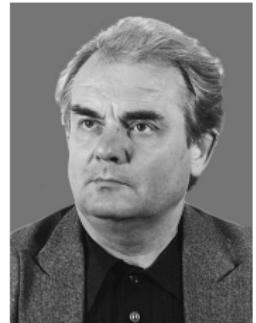
Section 3

Multilevel solvers

In the 1950s and 1960s, the only preconditioners for linear systems were simple iterations like Jacobi.



Mesh-dependent iterative solver. \times

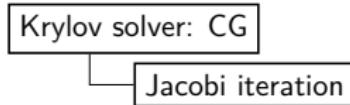


Radii Fedorenko



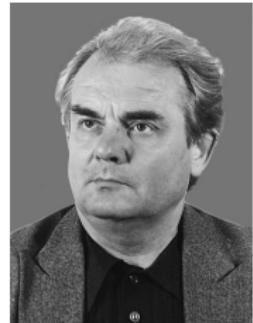
Achi Brandt

In the 1950s and 1960s, the only preconditioners for linear systems were simple iterations like Jacobi.



Mesh-dependent iterative solver. \times

In the 1970s, we learned we could make mesh-independent solvers with a multilevel approach.

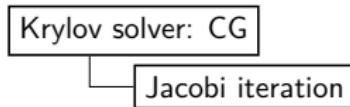


Radii Fedorenko

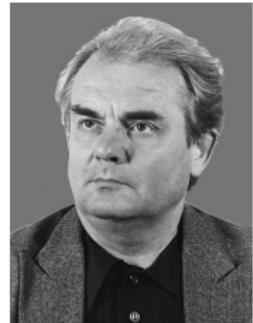


Achi Brandt

In the 1950s and 1960s, the only preconditioners for linear systems were simple iterations like Jacobi.

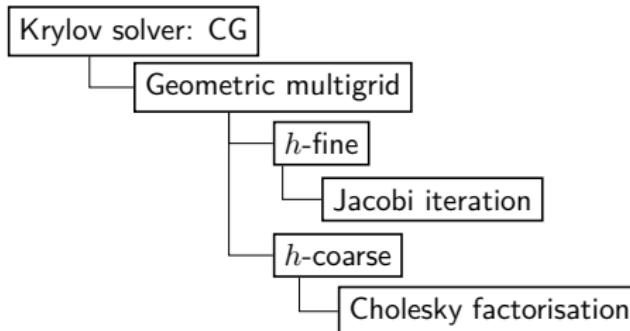


Mesh-dependent iterative solver. ✗



In the 1970s, we learned we could make mesh-independent solvers with a multilevel approach.

Radii Fedorenko



Mesh-independent iterative solver. ✓



Achi Brandt

Employing semismooth Newton in a multilevel context improves on its mesh-dependence.



Xue-Cheng Tai



Ed Bueler

Employing semismooth Newton in a multilevel context improves on its mesh-dependence.

As VIs are nonlinear, we employ a nonlinear version of multigrid known as *full approximation scheme* (FAS).



Xue-Cheng Tai



Ed Bueler

Employing semismooth Newton in a multilevel context improves on its mesh-dependence.

As VIs are nonlinear, we employ a nonlinear version of multigrid known as *full approximation scheme* (FAS).

We formulate the bounds on each level with a *constraint decomposition* (CD).



Xue-Cheng Tai



Ed Bueler

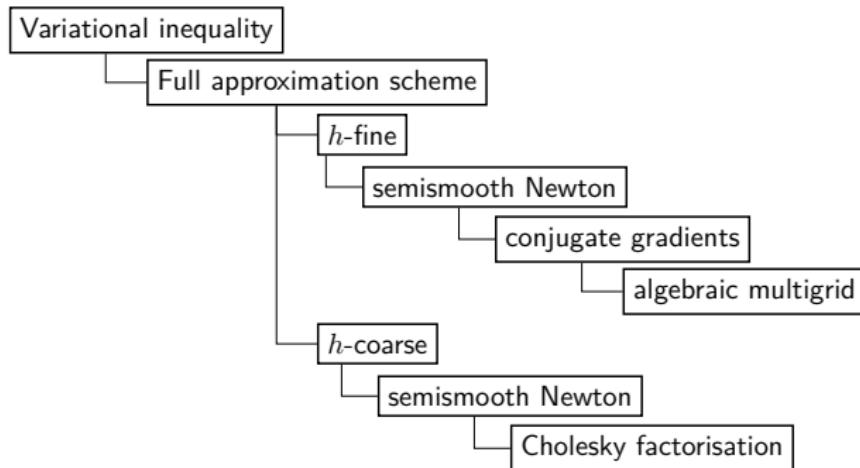
Employing semismooth Newton in a multilevel context improves on its mesh-dependence.

As VIs are nonlinear, we employ a nonlinear version of multigrid known as *full approximation scheme* (FAS).

We formulate the bounds on each level with a *constraint decomposition* (CD).



Xue-Cheng Tai



Ed Bueler

```
from fascd import FASCD Solver
sp = {"fascd_monitor": None,
       "fascd_cycle_type": "full",
       "fascd_levels_snes_type": "vinewtonrsls",
       "fascd_levels_snes_max_it": 1,
       "fascd_levels_snes_vi_zero_tolerance": 1.0e-12,
       "fascd_levels_snes_linesearch_type": "basic",
       "fascd_levels_ksp_type": "cg",
       "fascd_levels_ksp_max_it": 1,
       "fascd_levels_ksp_convergence_test": "skip",
       "fascd_levels_pc_type": "gamg",
       "fascd_coarse_snes_rtol": 1.0e-8,
       "fascd_coarse_snes_atol": 1.0e-12,
       "fascd_coarse_snes_stol": 1.0e-12,
       "fascd_coarse_snes_type": "vinewtonrsls",
       "fascd_coarse_snes_vi_zero_tolerance": 1.0e-12,
       "fascd_coarse_snes_linesearch_type": "basic",
       "fascd_coarse_ksp_type": "preonly",
       "fascd_coarse_pc_type": "cholesky",
       "fascd_coarse_pc_factor_mat_solver_type": "mumps"}
problem = NonlinearVariationalProblem(F, u, bcs)
solver = FASCD Solver(problem, solver_parameters=sp, bounds=(obstacle, None))
solver.solve()
```

Challenge!

QVII.2. Solve the same obstacle problem with FASCD.

Tabulate the number of FAS cycles as a function of mesh size.

Hint: to see more of what is going on, add

```
sp = {...,
      "fascd_levels_snes_monitor": None,
      "fascd_coarse_snes_monitor": None,
      ...}
```

Solving PDEs with Firedrake: eigenproblems

Patrick E. Farrell



University of Oxford

February 2024

Eigenvalue problems

We seek eigenvalues of the Laplacian with Dirichlet boundary conditions:
find $u \neq 0, \lambda \in \mathbb{R}$ such that

$$\begin{aligned} -\Delta u &= \lambda u && \text{in } \Omega, \\ u &= 0 && \text{on } \partial\Omega. \end{aligned}$$

Eigenvalue problems

We seek eigenvalues of the Laplacian with Dirichlet boundary conditions:
find $u \neq 0, \lambda \in \mathbb{R}$ such that

$$\begin{aligned} -\Delta u &= \lambda u && \text{in } \Omega, \\ u &= 0 && \text{on } \partial\Omega. \end{aligned}$$

At least twenty Nobel prizes have been awarded for phenomena related to eigenvalues.

Variational formulation

As usual, we multiply by a test function and integrate by parts: find $0 \neq u \in H_0^1(\Omega)$, $\lambda \in \mathbb{R}$ such that

$$\int_{\Omega} \nabla u \cdot \nabla v \, dx = \lambda \int_{\Omega} uv \, dx \quad \forall \, v \in H_0^1(\Omega).$$

To solve this, we will specify forms for both operators on the left and right.

TestFunctions and TrialFunctions

Consider the code

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

Three cases:

TestFunctions and TrialFunctions

Consider the code

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

Three cases:

If u is a Function and v is a Function, $a \in \mathbb{R}$.

TestFunctions and TrialFunctions

Consider the code

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

Three cases:

If u is a Function and v is a TestFunction, $a \in \mathbb{R}^n$, with

$$a_i = \int_{\Omega} \nabla u \cdot \nabla \phi_i.$$

TestFunctions and TrialFunctions

Consider the code

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

Three cases:

If u is a TrialFunction and v is a TestFunction, $a \in \mathbb{R}^{n \times n}$, with

$$a_{ij} = \int_{\Omega} \nabla \phi_i \cdot \nabla \phi_j.$$

Building eigensolvers

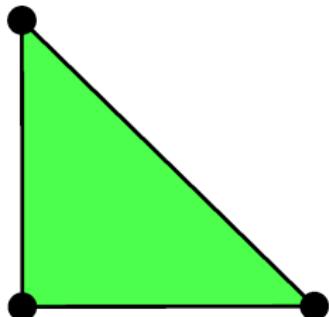
```
u = TrialFunction(V)
v = TestFunction(V)

a = inner(grad(u), grad(v))*dx
b = inner(u, v)*dx
bc = DirichletBC(V, 0, "on_boundary")
problem = LinearEigenproblem(a, b, bc)
```

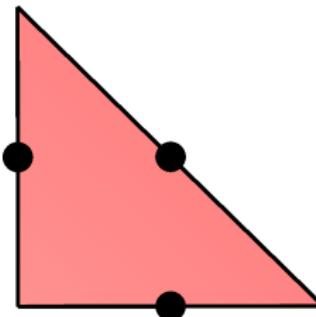
```
sp = {"eps_gen_hermitian": None,      # kind of problem
      "eps_smallest_real": None,        # which eigenvalues
      "eps_monitor": None,             # monitor convergence
      "eps_type": "krylovschur"}       # algorithm

# request ten eigenvalues
solver = LinearEigensolver(problem, 10, solver_parameters=sp)
# find out how many eigenvalues converged; maybe more than 10
ncv = solver.solve()

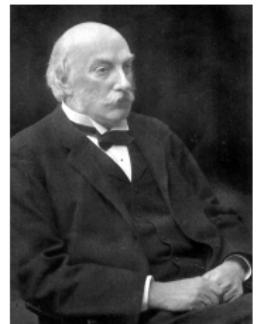
# Take real part, since we know it is Hermitian
values = [solver.eigenvalue(i).real for i in range(ncv)]
# Only take real part; .eigenfunction returns (real, complex)
efuncs = [solver.eigenfunction(i)[0] for i in range(ncv)]
```



Lagrange



Crouzeix–Raviart



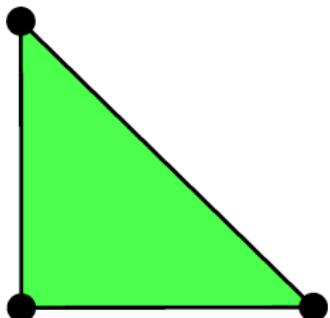
John William Strutt

With a **conforming** discretisation of a symmetric coercive problem, the eigenvalues are approximated *from above*:

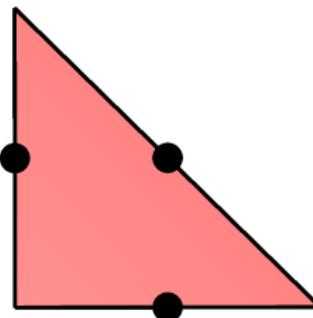
$$\lambda_{i,h} \geq \lambda_i, \quad i = 1, \dots$$



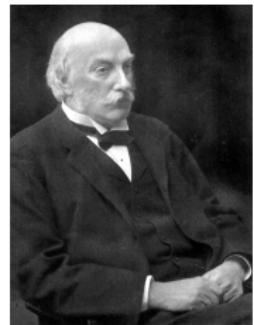
Carsten Carstensen



Lagrange



Crouzeix–Raviart



John William Strutt

With a **conforming** discretisation of a symmetric coercive problem, the eigenvalues are approximated *from above*:

$$\lambda_{i,h} \geq \lambda_i, \quad i = 1, \dots$$

With a **nonconforming** discretisation, on a sufficiently fine mesh, the eigenvalues are approximated *from below*:

$$\lambda_{i,h} \leq \lambda_i, \quad i = 1, \dots$$



Carsten Carstensen

Challenge!

QVIII.1. Solve the eigenvalue problem for the Laplacian with Dirichlet conditions on $\Omega = (0, \pi)^2$, with exact eigenvalues

$$\lambda_{nm} = n^2 + m^2, \quad n, m \in \mathbb{N}_+.$$

Verify that both

```
V_cg = FunctionSpace(mesh, "CG", 1)
V_cr = FunctionSpace(mesh, "CR", 1)
```

give second-order approximations from above and below to the first ten eigenvalues.

Challenge!

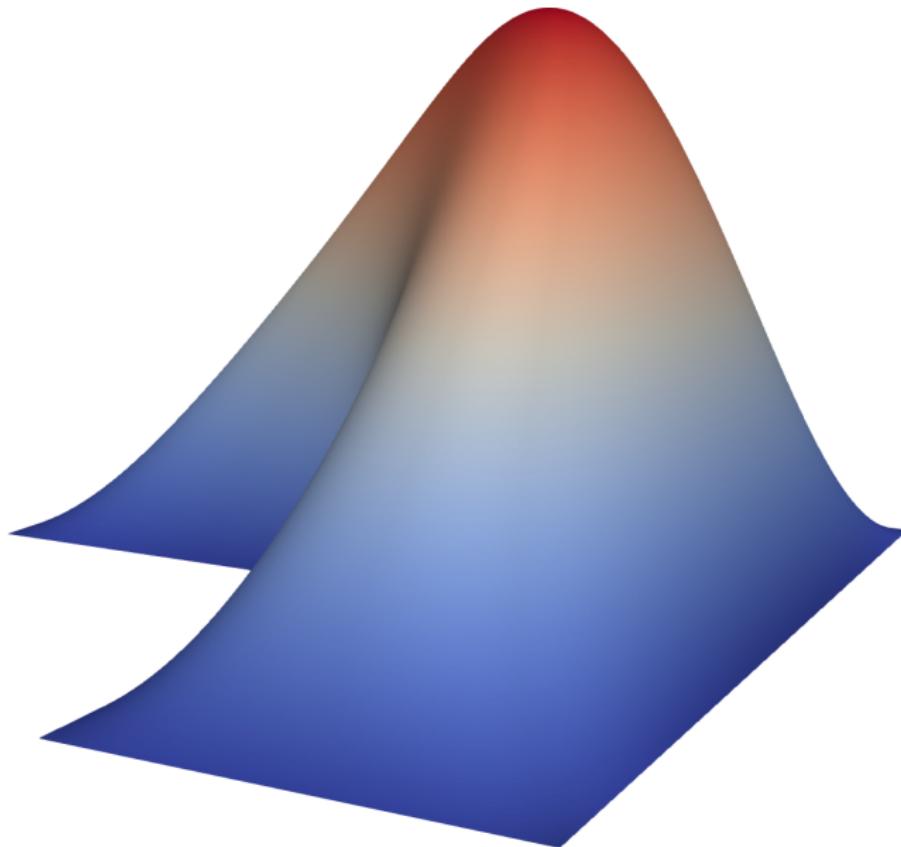
QVIII.2. Compute the first eigenvalue of the Laplacian with Dirichlet conditions on the L-shaped domain with an adaptive procedure.

Terminate when your error estimator (given by the difference between the CG₁ and CR₁ discretisations) is less than 10⁻³.

Drive the adaptation procedure with the residual estimator

$$\eta_K^2 = h_K^2 \int_K |\lambda_h u_h + \nabla^2 u_h|^2 \, dx + \frac{h_K}{2} \int_{\partial K \setminus \partial \Omega} [\![\nabla u_h \cdot n]\!]^2 \, ds,$$

where u_h and λ_h are the CG₁ approximation to the eigenfunction and eigenvalue respectively.



The first eigenfunction on the L-shaped domain.

Solving PDEs with Firedrake: saddle point solvers

Patrick E. Farrell



University of Oxford

February 2024

Recall the incompressible Newtonian Stokes equations:

$$\begin{aligned} -2\text{Re}^{-1}\nabla \cdot (\epsilon(u)) + \nabla p &= f, \\ -\nabla \cdot u &= 0. \end{aligned}$$

Recall the incompressible Newtonian Stokes equations:

$$\begin{aligned} -2\text{Re}^{-1}\nabla \cdot (\epsilon(u)) + \nabla p &= f, \\ -\nabla \cdot u &= 0. \end{aligned}$$

We can write this in matrix form as

$$\begin{bmatrix} -2\text{Re}^{-1}\nabla \cdot (\epsilon(\cdot)) & \nabla \\ -\nabla \cdot & \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}.$$

Recall the incompressible Newtonian Stokes equations:

$$\begin{aligned} -2\text{Re}^{-1}\nabla \cdot (\epsilon(u)) + \nabla p &= f, \\ -\nabla \cdot u &= 0. \end{aligned}$$

We can write this in matrix form as

$$\begin{bmatrix} -2\text{Re}^{-1}\nabla \cdot (\epsilon(\cdot)) & \nabla \\ -\nabla \cdot & \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}.$$

When discretised, this yields a linear system of the form

$$\begin{bmatrix} A & B^\top \\ B & \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}.$$

Recall the incompressible Newtonian Stokes equations:

$$\begin{aligned} -2\text{Re}^{-1}\nabla \cdot (\epsilon(u)) + \nabla p &= f, \\ -\nabla \cdot u &= 0. \end{aligned}$$

We can write this in matrix form as

$$\begin{bmatrix} -2\text{Re}^{-1}\nabla \cdot (\epsilon(\cdot)) & \nabla \\ -\nabla \cdot & \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}.$$

When discretised, this yields a linear system of the form

$$\begin{bmatrix} A & B^\top \\ B & \end{bmatrix} \begin{bmatrix} u \\ p \end{bmatrix} = \begin{bmatrix} f \\ 0 \end{bmatrix}.$$

Problems of this form are referred to as *saddle point problems*.

Consider the following minimisation problem:

$$u = \arg \min_{v \in H_0^1(\Omega; \mathbb{R}^n)} \frac{1}{2} \int_{\Omega} 2\text{Re}^{-1}\epsilon(v) : \epsilon(v) \, dx - \int_{\Omega} f \cdot v \, dx,$$

subject to $\nabla \cdot v = 0$.

Consider the following minimisation problem:

$$\begin{aligned} u = \arg \min_{v \in H_0^1(\Omega; \mathbb{R}^n)} \quad & \frac{1}{2} \int_{\Omega} 2\text{Re}^{-1}\epsilon(v) : \epsilon(v) \, dx - \int_{\Omega} f \cdot v \, dx, \\ \text{subject to} \quad & \nabla \cdot v = 0. \end{aligned}$$

Introducing a Lagrange multiplier $p \in H_0^1(\Omega)$ for the incompressibility constraint yields the Lagrangian

$$L(u, p) = \frac{1}{2} \int_{\Omega} 2\text{Re}^{-1}\epsilon(u) : \epsilon(u) \, dx - \int_{\Omega} f \cdot u \, dx - \int_{\Omega} p \nabla \cdot u \, dx.$$

Consider the following minimisation problem:

$$\begin{aligned} u = \arg \min_{v \in H_0^1(\Omega; \mathbb{R}^n)} \quad & \frac{1}{2} \int_{\Omega} 2\text{Re}^{-1}\epsilon(v) : \epsilon(v) \, dx - \int_{\Omega} f \cdot v \, dx, \\ \text{subject to} \quad & \nabla \cdot v = 0. \end{aligned}$$

Introducing a Lagrange multiplier $p \in H_0^1(\Omega)$ for the incompressibility constraint yields the Lagrangian

$$L(u, p) = \frac{1}{2} \int_{\Omega} 2\text{Re}^{-1}\epsilon(u) : \epsilon(u) \, dx - \int_{\Omega} f \cdot u \, dx - \int_{\Omega} p \nabla \cdot u \, dx.$$

Taking the optimality conditions, we find exactly the Stokes equations. The solution (u, p) is a saddle point of the Lagrangian because

$$L(u, q) \leq L(u, p) \leq L(v, p) \text{ for all } v \in H_0^1(\Omega; \mathbb{R}^n), \quad q \in L_0^2(\Omega).$$

We want to build solvers for saddle point problems like

$$\begin{aligned} Au + B^\top p &= f, \\ Bu &= 0. \end{aligned}$$

We want to build solvers for saddle point problems like

$$\begin{aligned} Au + B^\top p &= f, \\ Bu &= 0. \end{aligned}$$

If A is invertible, then we can left-multiply the first equation by A^{-1} to get

$$u = A^{-1}f - A^{-1}B^\top p,$$

We want to build solvers for saddle point problems like

$$\begin{aligned} Au + B^\top p &= f, \\ Bu &= 0. \end{aligned}$$

If A is invertible, then we can left-multiply the first equation by A^{-1} to get

$$u = A^{-1}f - A^{-1}B^\top p,$$

and substituting this into the second equation yields

$$-BA^{-1}B^\top p = -BA^{-1}f,$$

where the new operator

$$S := -BA^{-1}B^\top$$

is called the *Schur complement*. The Schur complement is **dense**.

In fact, more generally, if A is invertible

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} I & 0 \\ CA^{-1} & I \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I & A^{-1}B \\ 0 & I \end{bmatrix}.$$

where $S = D - CA^{-1}B$ is the (dense!) Schur complement.

In fact, more generally, if A is invertible

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} = \begin{bmatrix} I & 0 \\ CA^{-1} & I \end{bmatrix} \begin{bmatrix} A & 0 \\ 0 & S \end{bmatrix} \begin{bmatrix} I & A^{-1}B \\ 0 & I \end{bmatrix}.$$

where $S = D - CA^{-1}B$ is the (dense!) Schur complement.

This is extremely useful, because we can write an explicit formula for the inverse:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix}^{-1} = \begin{bmatrix} I & -A^{-1}B \\ 0 & I \end{bmatrix} \begin{bmatrix} A^{-1} & 0 \\ 0 & S^{-1} \end{bmatrix} \begin{bmatrix} I & 0 \\ -CA^{-1} & I \end{bmatrix}.$$

This gives rise to four related theorems.

Theorem (full)

The choice

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

will yield Krylov convergence in **1 iteration**.



Andy Wathen



Gene Golub

This gives rise to four related theorems.

Theorem (lower)

The choice

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

will yield Krylov convergence in **2 iterations**.



Andy Wathen



Gene Golub

This gives rise to four related theorems.

Theorem (upper)

The choice

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

will yield Krylov convergence in **2 iterations**.



Andy Wathen



Gene Golub

This gives rise to four related theorems.

Theorem (diag)

The choice

$$\mathcal{P} = \begin{pmatrix} A & 0 \\ 0 & -S \end{pmatrix}$$

will yield Krylov convergence in **3 iterations, if $D = 0$.**



Andy Wathen



Gene Golub

This gives rise to four related theorems.

Theorem (diag)

The choice

$$\mathcal{P} = \begin{pmatrix} A & 0 \\ 0 & -S \end{pmatrix}$$

will yield Krylov convergence in **3 iterations**, **if** $D = 0$.



Andy Wathen

How do you use this?

We have to build solvers for A and S .



Gene Golub

For Stokes,

$$A_{ij} = 2\text{Re}^{-1} \int_{\Omega} \epsilon(\phi_j) : \epsilon(\phi_i) \, dx,$$

with boundary conditions a sym. coercive operator.



Andy Wathen



David Silvester

For Stokes,

$$A_{ij} = 2\text{Re}^{-1} \int_{\Omega} \epsilon(\phi_j) : \epsilon(\phi_i) \, dx,$$

with boundary conditions a sym. coercive operator.

Multigrid is the natural choice to approximate A^{-1} .



Andy Wathen



David Silvester

For Stokes,

$$A_{ij} = 2\text{Re}^{-1} \int_{\Omega} \epsilon(\phi_j) : \epsilon(\phi_i) \, dx,$$

with boundary conditions a sym. coercive operator.

Multigrid is the natural choice to approximate A^{-1} .



Andy Wathen

But what about the Schur complement S ?



David Silvester

For Stokes,

$$A_{ij} = 2\text{Re}^{-1} \int_{\Omega} \epsilon(\phi_j) : \epsilon(\phi_i) \, dx,$$

with boundary conditions a sym. coercive operator.

Multigrid is the natural choice to approximate A^{-1} .



Andy Wathen

But what about the Schur complement S ?

Theorem (Silvester & Wathen, 1994)

For a stable discretisation, the Schur complement is *spectrally equivalent* to the pressure mass matrix:

$$\underline{c}x^\top Q_\nu x \leq x^\top Sx \leq \bar{c}x^\top Q_\nu x,$$

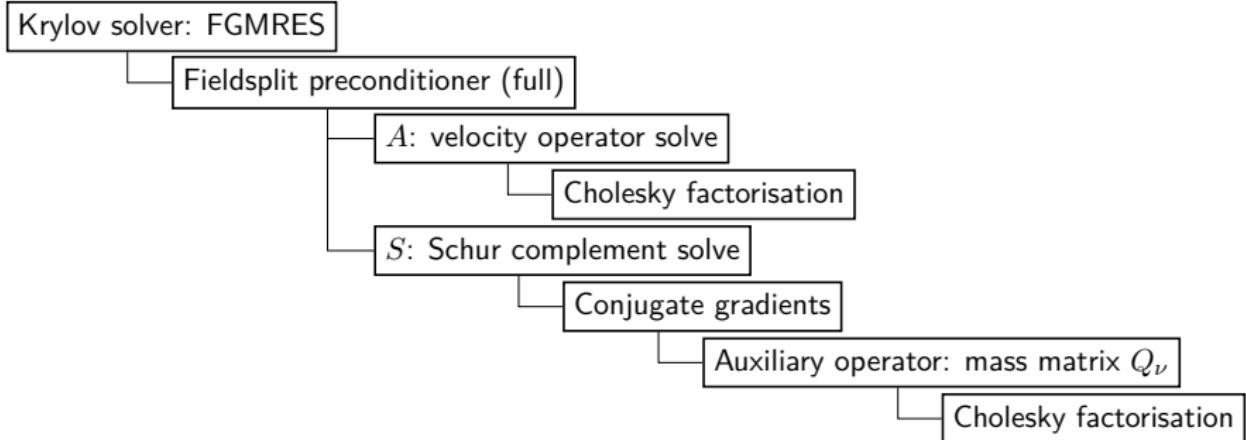
where

$$(Q_\nu)_{ij} = \int_{\Omega} \frac{\text{Re}}{2} \psi_j \psi_i \, dx.$$



David Silvester

Let's see how well the mass matrix approximates the Schur complement.



Testing the mass matrix approximation.

But wait ...

... the pressure mass matrix doesn't appear in our equations!

But wait ...

... the pressure mass matrix doesn't appear in our equations!

Good news!

In Firedrake it is simple to plug in new operators into the solver.

But wait ...

... the pressure mass matrix doesn't appear in our equations!

Good news!

In Firedrake it is simple to plug in new operators into the solver.

We try it out on a classical benchmark problem, the (regularised) lid-driven cavity.

```
from firedrake import *

# Use a triangular mesh
base = UnitSquareMesh(16, 16, diagonal="crossed")
mh = MeshHierarchy(base, 1)
mesh = mh[-1]
n = FacetNormal(mesh)
(x, y) = SpatialCoordinate(mesh)

# Define Taylor--Hood function space W
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = MixedFunctionSpace([V, Q])

# Define Reynolds number and bcs
Re = Constant(1)
bcs = [DirichletBC(W.sub(0), Constant((0, 0)), (1, 2, 3)),
       DirichletBC(W.sub(0), as_vector([16 * x**2 * (1-x)**2, 0]), (4,))]

w = Function(W, name="Solution")
(u, p) = split(w)
```

```
# Define Lagrangian
L = (
    0.5 * inner(2/Re * sym(grad(u)), sym(grad(u)))*dx
    -      inner(p, div(u))*dx
)
# Optimality conditions
F = derivative(L, w)

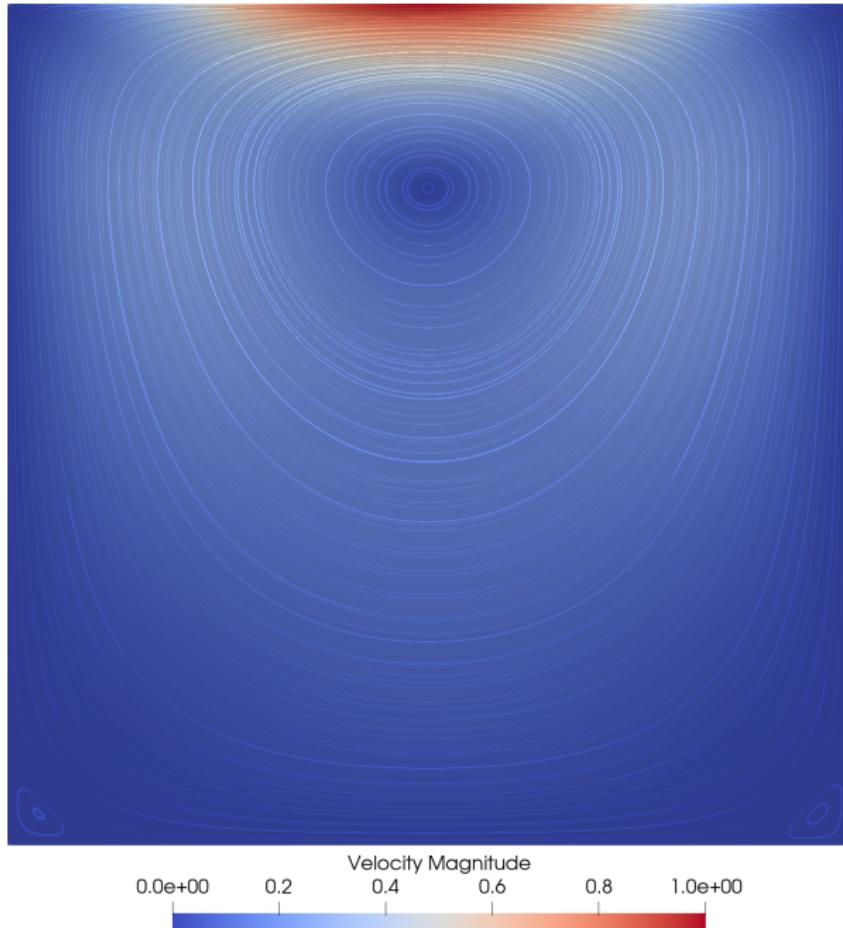
# Make an auxiliary operator representing
# the mass matrix on the pressure space,
# weighted inversely by the viscosity
class Mass(AuxiliaryOperatorPC):
    def form(self, pc, test, trial):
        a = inner(Re/2 * test, trial)*dx
        bcs = None
        return (a, bcs)
```

```
sp = {
    "mat_type": "aij",
    "snes_type": "ksponly",
    "ksp_type": "fgmres",
    "ksp_rtol": 1.0e-10,
    "ksp_monitor": None,
    "pc_type": "fieldsplit",
    "pc_fieldsplit_type": "schur",
    "pc_fieldsplit_schur_factorization_type": "full",
    "fieldsplit_0": {
        "ksp_type": "preonly",
        "pc_type": "cholesky",
        "pc_factor_mat_solver_type": "mumps",
    },
    "fieldsplit_1": {
        "ksp_type": "cg",
        "ksp_rtol": 1.0e-12,
        "ksp_converged_reason": None,
        "pc_type": "python",
        "pc_python_type": __name__ + ".Mass",
        "aux_pc_type": "cholesky",
        "aux_pc_factor_mat_solver_type": "mumps",
        "aux_pc_use_amat": False,
    },
}
```

```
solve(F == 0, w, bcs, solver_parameters=sp)

# Monitor incompressibility
print(f"||div u||: {norm(div(u), 'L2'):.2e}")

# Save solutions
(u_, p_) = w.subfunctions
u_.rename("Velocity")
p_.rename("Pressure")
File("output/stokes.pvd").write(u_, p_)
```



Residual norms for firedrake_0_ solve.

0 KSP Residual norm 1.577626537202e+01

1 KSP Residual norm 2.759307931408e-14

```
Residual norms for firedrake_0_ solve.  
0 KSP Residual norm 1.577626537202e+01  
1 KSP Residual norm 2.759307931408e-14
```

With `"pc_fieldsplit_schur_factorization_type": "upper"`:

```
Residual norms for firedrake_0_ solve.  
0 KSP Residual norm 1.577626537202e+01  
1 KSP Residual norm 1.305528573943e-02  
2 KSP Residual norm 2.503687002640e-14
```

```
Residual norms for firedrake_0_ solve.  
0 KSP Residual norm 1.577626537202e+01  
1 KSP Residual norm 2.759307931408e-14
```

With "pc_fieldsplit_schur_factorization_type": "upper":

```
Residual norms for firedrake_0_ solve.  
0 KSP Residual norm 1.577626537202e+01  
1 KSP Residual norm 1.305528573943e-02  
2 KSP Residual norm 2.503687002640e-14
```

With "pc_fieldsplit_schur_factorization_type": "diag":

```
Residual norms for firedrake_0_ solve.  
0 KSP Residual norm 1.577626537202e+01  
1 KSP Residual norm 2.614942935999e-01  
2 KSP Residual norm 1.454675122217e-01  
3 KSP Residual norm 9.195567042761e-13
```

```
Residual norms for firedrake_0_ solve.
```

```
0 KSP Residual norm 1.577626537202e+01
```

```
1 KSP Residual norm 2.759307931408e-14
```

With "pc_fieldsplit_schur_factorization_type": "upper":

```
Residual norms for firedrake_0_ solve.
```

```
0 KSP Residual norm 1.577626537202e+01
```

```
1 KSP Residual norm 1.305528573943e-02
```

```
2 KSP Residual norm 2.503687002640e-14
```

With "pc_fieldsplit_schur_factorization_type": "diag":

```
Residual norms for firedrake_0_ solve.
```

```
0 KSP Residual norm 1.577626537202e+01
```

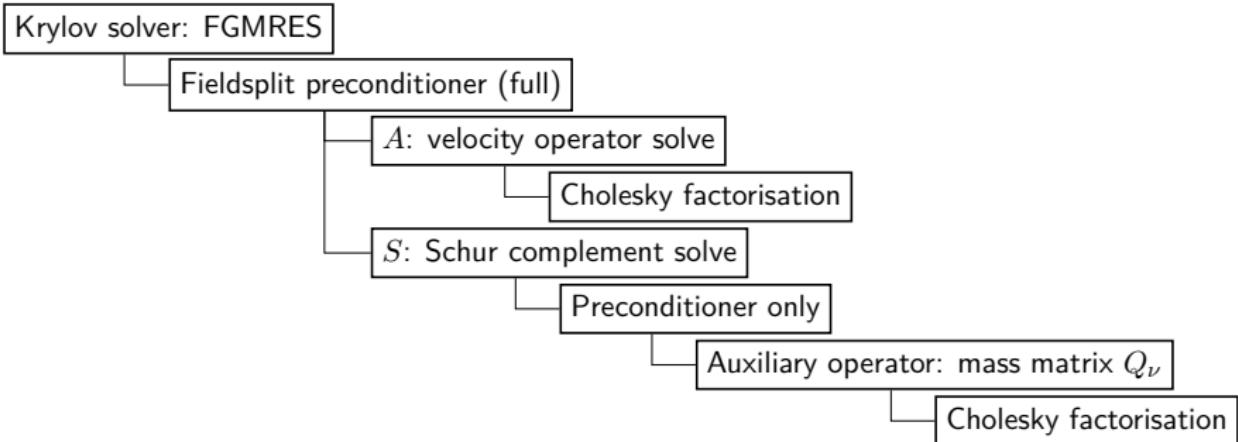
```
1 KSP Residual norm 2.614942935999e-01
```

```
2 KSP Residual norm 1.454675122217e-01
```

```
3 KSP Residual norm 9.195567042761e-13
```

Each inner Schur complement solve takes ~ 18 iterations.

Now let's solve the Schur complement problem inexactly.



Solving the Schur complement inexactly.

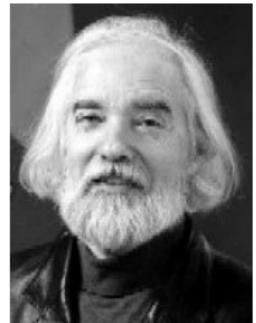
```
"fieldsplit_1": {  
    "ksp_type": "preonly",  
    "pc_type": "python",  
    "pc_python_type": __name__ + ".Mass",  
    "aux_pc_type": "cholesky",  
    "aux_pc_factor_mat_solver_type": "mumps",  
    "aux_pc_use_amat": False,  
},
```

```
Residual norms for firedrake_0_ solve.  
0 KSP Residual norm 1.577626537202e+01  
1 KSP Residual norm 1.981937819721e-02  
2 KSP Residual norm 4.154550231765e-03  
3 KSP Residual norm 1.040546738142e-03  
4 KSP Residual norm 2.347790415827e-04  
5 KSP Residual norm 5.016034801474e-05  
6 KSP Residual norm 9.211052558009e-06  
7 KSP Residual norm 1.084095101467e-06  
8 KSP Residual norm 1.259596554900e-07  
9 KSP Residual norm 2.915404476515e-08  
10 KSP Residual norm 1.045078417637e-08  
11 KSP Residual norm 3.184834624707e-09  
12 KSP Residual norm 1.999128476334e-09  
13 KSP Residual norm 1.472161552385e-09
```

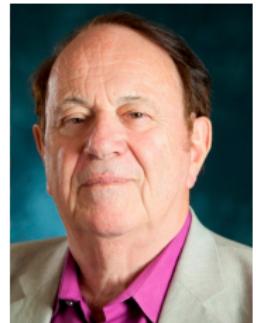
```
Residual norms for firedrake_0_ solve.  
0 KSP Residual norm 1.577626537202e+01  
1 KSP Residual norm 1.981937819721e-02  
2 KSP Residual norm 4.154550231765e-03  
3 KSP Residual norm 1.040546738142e-03  
4 KSP Residual norm 2.347790415827e-04  
5 KSP Residual norm 5.016034801474e-05  
6 KSP Residual norm 9.211052558009e-06  
7 KSP Residual norm 1.084095101467e-06  
8 KSP Residual norm 1.259596554900e-07  
9 KSP Residual norm 2.915404476515e-08  
10 KSP Residual norm 1.045078417637e-08  
11 KSP Residual norm 3.184834624707e-09  
12 KSP Residual norm 1.999128476334e-09  
13 KSP Residual norm 1.472161552385e-09
```

Converging, but could we improve it?

One idea is the *augmented Lagrangian* method.



Michel Fortin



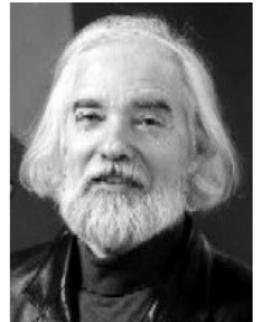
Roland Glowinski

One idea is the *augmented Lagrangian* method.

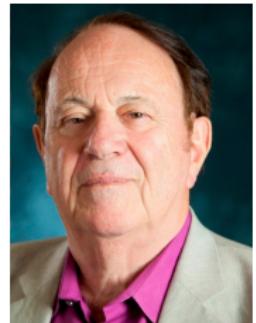
We augment the Lagrangian with a penalty term

$$L_\gamma(u, p) = L(u, p) + \frac{\gamma}{2} \int_{\Omega} (\nabla \cdot u)^2 \, dx$$

for $\gamma \geq 0$.



Michel Fortin



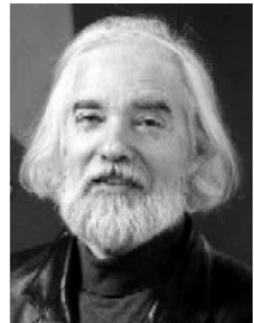
Roland Glowinski

One idea is the *augmented Lagrangian* method.

We augment the Lagrangian with a penalty term

$$L_\gamma(u, p) = L(u, p) + \frac{\gamma}{2} \int_{\Omega} (\nabla \cdot u)^2 \, dx$$

for $\gamma \geq 0$.



Michel Fortin

The Schur complement is approximated by

$$S \sim \left(\frac{2}{\text{Re}} + \gamma \right)^{-1} Q$$

with the spectral equivalence improving for larger γ .



Roland Glowinski

```
# Augmented Lagrangian term
gamma = Constant(1)

# Define Lagrangian
L = (
    0.5 * inner(2/Re * sym(grad(u)), sym(grad(u)))*dx
    - inner(p, div(u))*dx
    + 0.5 * gamma * inner(div(u), div(u))*dx
)
```

```
class Mass(AuxiliaryOperatorPC):
    def form(self, pc, test, trial):
        a = inner((2/Re + gamma)**(-1) * test, trial)*dx
        bcs = None
        return (a, bcs)
```

γ	# iterations	$\ \nabla \cdot u\ _{L^2}$
0	13	1.97×10^{-3}
1	10	1.72×10^{-3}
10	6	9.82×10^{-4}
100	4	2.26×10^{-4}
1000	2	2.68×10^{-5}
10000	2	2.72×10^{-6}

γ	# iterations	$\ \nabla \cdot u\ _{L^2}$
0	13	1.97×10^{-3}
1	10	1.72×10^{-3}
10	6	9.82×10^{-4}
100	4	2.26×10^{-4}
1000	2	2.68×10^{-5}
10000	2	2.72×10^{-6}

Good news

The Schur complement approximation improves, **and** we better enforce incompressibility!

The catch ...

... is that it makes the velocity solve **much** harder.

The catch ...

... is that it makes the velocity solve **much** harder.

The operator

$$A_{ij} = 2\text{Re}^{-1} \int_{\Omega} \epsilon(\phi_j) : \epsilon(\phi_i) \, dx$$

is very amenable to standard multigrid methods.

The catch ...

... is that it makes the velocity solve **much** harder.

The operator

$$A_{ij} = 2\text{Re}^{-1} \int_{\Omega} \epsilon(\phi_j) : \epsilon(\phi_i) \, dx$$

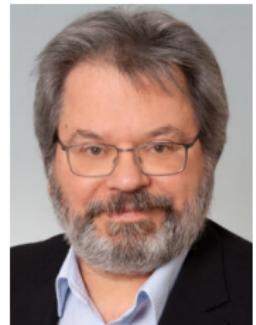
is very amenable to standard multigrid methods.

But the augmented operator

$$(A_\gamma)_{ij} = 2\text{Re}^{-1} \int_{\Omega} \epsilon(\phi_j) : \epsilon(\phi_i) \, dx + \gamma \int_{\Omega} (\nabla \cdot \phi_j)(\nabla \cdot \phi_i) \, dx$$

is quite subtle to solve for $\gamma \gg \text{Re}$.

Building a geometric multigrid solver for A_γ hinges on
the kernel of div .



Joachim Schöberl

Building a geometric multigrid solver for A_γ hinges on the kernel of div .

Schöberl's theory (1999, Numer. Math.)

For a parameter-robust multigrid method, you need:

- ▶ *kernel-capturing multigrid relaxation;*
- ▶ *kernel-mapping prolongation.*



Joachim Schöberl

Building a geometric multigrid solver for A_γ hinges on the kernel of div .

Schöberl's theory (1999, Numer. Math.)

For a parameter-robust multigrid method, you need:

- ▶ *kernel-capturing multigrid relaxation;*
- ▶ *kernel-mapping prolongation.*



Joachim Schöberl

Schöberl's theory applies to symmetric, coercive problems with singular terms. But amazingly **it works even for much harder problems!**

Building a geometric multigrid solver for A_γ hinges on the kernel of div .

Schöberl's theory (1999, Numer. Math.)

For a parameter-robust multigrid method, you need:

- ▶ *kernel-capturing multigrid relaxation;*
- ▶ *kernel-mapping prolongation.*



Joachim Schöberl

Schöberl's theory applies to symmetric, coercive problems with singular terms. But amazingly **it works even for much harder problems!**

Today we will only discuss the relaxation, since that is all we need.

Consider the variational problem: find $u \in V$, $\dim(V) < \infty$, such that

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

Consider the variational problem: find $u \in V$, $\dim(V) < \infty$, such that

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

The way we design relaxation methods is via *subspace correction*.

Subspace correction method (Xu, 1992, SIAM Rev.)

Choose an initial guess u_k and a space decomposition

$$V = \sum_i V_i.$$

Consider the variational problem: find $u \in V$, $\dim(V) < \infty$, such that

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

The way we design relaxation methods is via *subspace correction*.

Subspace correction method (Xu, 1992, SIAM Rev.)

Choose an initial guess u_k and a space decomposition

$$V = \sum_i V_i.$$

Solve for error approximations: for each i , find $V_i \ni e_i \approx u - u_k$ such that

$$a(e_i, v) = a(u, v) - a(u_k, v) = (f, v) - a(u_k, v) \quad \forall v \in V_i.$$

Consider the variational problem: find $u \in V$, $\dim(V) < \infty$, such that

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

The way we design relaxation methods is via *subspace correction*.

Subspace correction method (Xu, 1992, SIAM Rev.)

Choose an initial guess u_k and a space decomposition

$$V = \sum_i V_i.$$

Solve for error approximations: for each i , find $V_i \ni e_i \approx u - u_k$ such that

$$a(e_i, v) = a(u, v) - a(u_k, v) = (f, v) - a(u_k, v) \quad \forall v \in V_i.$$

Then combine the updates with weights:

$$u_{k+1} = u_k + \sum_i w_i(e_i).$$

Examples:

Jacobi

Let $V = \text{span}(\phi_1, \dots, \phi_N)$. Taking

$$V_i = \text{span}(\phi_i)$$

yields the usual Jacobi iteration.

Examples:

Jacobi

Let $V = \text{span}(\phi_1, \dots, \phi_N)$. Taking

$$V_i = \text{span}(\phi_i)$$

yields the usual Jacobi iteration.

Domain decomposition

If you partition the domain into overlapping $\Omega = \Omega_1 \cup \Omega_2 \cup \dots \cup \Omega_N$ and take

$$V_i = \{\text{functions in } V \text{ supported on } \Omega_i\}$$

you get a classical domain decomposition method.

Kernel-capturing multigrid relaxation

Now consider the problem: for $\alpha, \beta > 0$, find $u \in V$ such that

$$\alpha a(u, v) + \beta b(u, v) = (f, v) \quad \forall v \in V,$$

where a is symmetric coercive and b is symmetric positive semidefinite.

Kernel-capturing multigrid relaxation

Now consider the problem: for $\alpha, \beta > 0$, find $u \in V$ such that

$$\alpha a(u, v) + \beta b(u, v) = (f, v) \quad \forall v \in V,$$

where a is symmetric coercive and b is symmetric positive semidefinite.

For Stokes with augmented Lagrangian, we have

$$a(u, v) = \int_{\Omega} \epsilon(u) : \epsilon(v) \, dx, \quad b(u, v) = \int_{\Omega} \operatorname{div} u \cdot \operatorname{div} v \, dx.$$

Kernel-capturing multigrid relaxation

Now consider the problem: for $\alpha, \beta > 0$, find $u \in V$ such that

$$\alpha a(u, v) + \beta b(u, v) = (f, v) \quad \forall v \in V,$$

where a is symmetric coercive and b is symmetric positive semidefinite.

Theorem [Schöberl (1999), Lee, Wu, Xu, Zikatanov (2007)]

Let the kernel be

$$\mathcal{N} = \{u \in V : b(u, v) = 0 \quad \forall v \in V\}.$$

If *the decomposition captures the kernel*

$$\mathcal{N} = \sum_i \mathcal{N} \cap V_i,$$

in a stable way then the convergence will be robust wrt α and β .

In 2D, the *Stokes complex* is

$$\mathbb{R} \xrightarrow{\text{id}} H^2 \xrightarrow{\text{curl}} [H^1]^d \xrightarrow{\text{div}} L^2 \xrightarrow{\text{null}} 0.$$

This sequence is exact on a simply connected domain.



Michael Neilan



Johnny Guzman

In 2D, the *Stokes complex* is

$$\mathbb{R} \xrightarrow{\text{id}} H^2 \xrightarrow{\text{curl}} [H^1]^d \xrightarrow{\text{div}} L^2 \xrightarrow{\text{null}} 0.$$



Michael Neilan

Consequence

On a simply connected domain, $\ker(\text{div}) = \text{range}(\text{curl})$.



Johnny Guzman

In 2D, the *Stokes complex* is

$$\mathbb{R} \xrightarrow{\text{id}} H^2 \xrightarrow{\text{curl}} [H^1]^d \xrightarrow{\text{div}} L^2 \xrightarrow{\text{null}} 0.$$



Michael Neilan

Consequence

On a simply connected domain, $\ker(\text{div}) = \text{range}(\text{curl})$.

Consequence

By studying the space to the left, we can understand $\ker(\text{div})$.

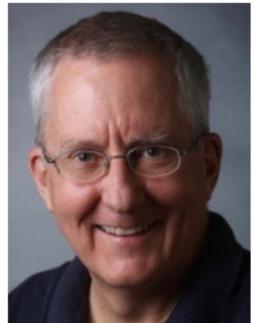


Johnny Guzman

In 2D, for velocity degree $p < 4$, we don't know what the potential space is.



John Morgan



Ridgway Scott

In 2D, for velocity degree $p < 4$, we don't know what the potential space is.

But for $p \geq 4$, we do: it is given by the *Morgan–Scott element*.



John Morgan



Ridgway Scott

In 2D, for velocity degree $p < 4$, we don't know what the potential space is.

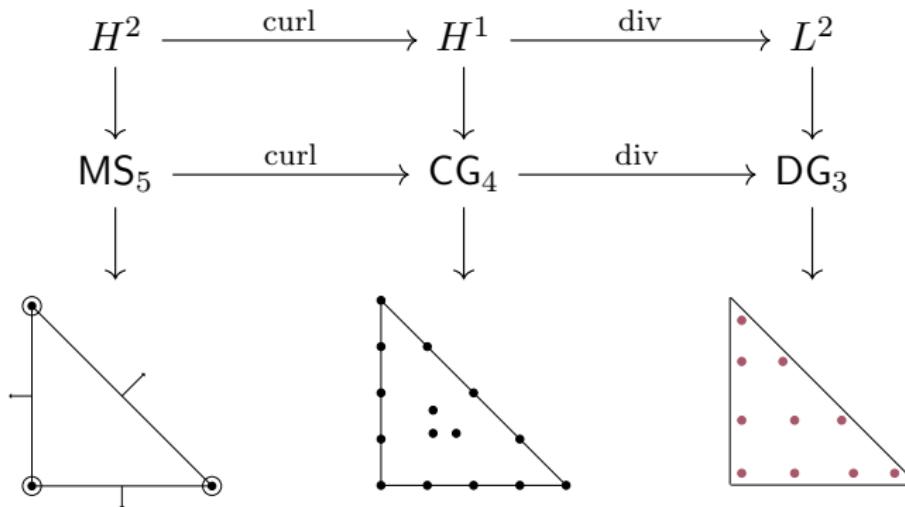
But for $p \geq 4$, we do: it is given by the *Morgan–Scott element*.



John Morgan



Ridgway Scott



Why is this useful?

Why is this useful?

By exactness of the complex, if $u \in \text{CG}_4$ and $\operatorname{div} u = 0$, then

$$u = \operatorname{curl} \phi, \quad \phi \in \text{MS}_5.$$

Why is this useful?

By exactness of the complex, if $u \in \text{CG}_4$ and $\operatorname{div} u = 0$, then

$$u = \operatorname{curl} \phi, \quad \phi \in \text{MS}_5.$$

Let $\{\zeta_1, \dots, \zeta_N\}$ be the (local) basis for MS_5 . Then we can write

$$\begin{aligned} u &= \operatorname{curl} \phi = \operatorname{curl} \sum_{i=1}^N c_i \zeta_i \\ &= \sum_{i=1}^N c_i \operatorname{curl} \zeta_i. \end{aligned}$$

Why is this useful?

By exactness of the complex, if $u \in \text{CG}_4$ and $\text{div } u = 0$, then

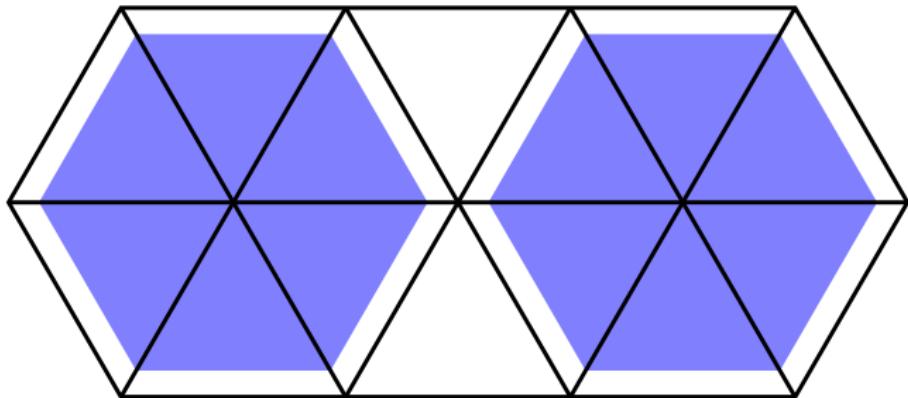
$$u = \text{curl } \phi, \quad \phi \in \text{MS}_5.$$

Let $\{\zeta_1, \dots, \zeta_N\}$ be the (local) basis for MS_5 . Then we can write

$$\begin{aligned} u = \text{curl } \phi &= \text{curl} \sum_{i=1}^N c_i \zeta_i \\ &= \sum_{i=1}^N c_i \text{curl } \zeta_i. \end{aligned}$$

If we *choose our space decomposition* so that each $\text{curl } \zeta_i$ is contained completely within at least one patch, this gives us a candidate kernel-capturing space decomposition!

This motivates the *vertex-star* space decomposition.



In our space decomposition

$$V = \sum V_i,$$

we construct each V_i by

$$V_i = \{\text{all functions supported on the patch of cells around a vertex}\}.$$

So, to ensure we have a γ -robust multigrid scheme for the augmented velocity operator, we switch to CG₄.



Ridgway Scott

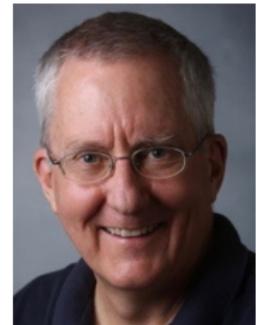


Michael Vogelius

So, to ensure we have a γ -robust multigrid scheme for the augmented velocity operator, we switch to CG₄.

A consequence

With $p = 4$, we can use DG₃ (Scott–Vogelius) for the pressure, instead of CG₃ (Taylor–Hood).



Ridgway Scott



Michael Vogelius

So, to ensure we have a γ -robust multigrid scheme for the augmented velocity operator, we switch to CG₄.

A consequence

With $p = 4$, we can use DG₃ (Scott–Vogelius) for the pressure, instead of CG₃ (Taylor–Hood).



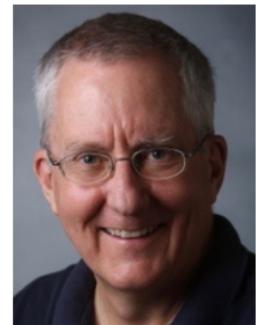
Ridgway Scott

For $p < 4$, Scott–Vogelius is not inf-sup stable except on special meshes. For $p \geq 4$, it is inf-sup stable under mild conditions on the mesh.



Michael Vogelius

So, to ensure we have a γ -robust multigrid scheme for the augmented velocity operator, we switch to CG₄.



A consequence

With $p = 4$, we can use DG₃ (Scott–Vogelius) for the pressure, instead of CG₃ (Taylor–Hood).

Ridgway Scott

For $p < 4$, Scott–Vogelius is not inf-sup stable except on special meshes. For $p \geq 4$, it is inf-sup stable under mild conditions on the mesh.

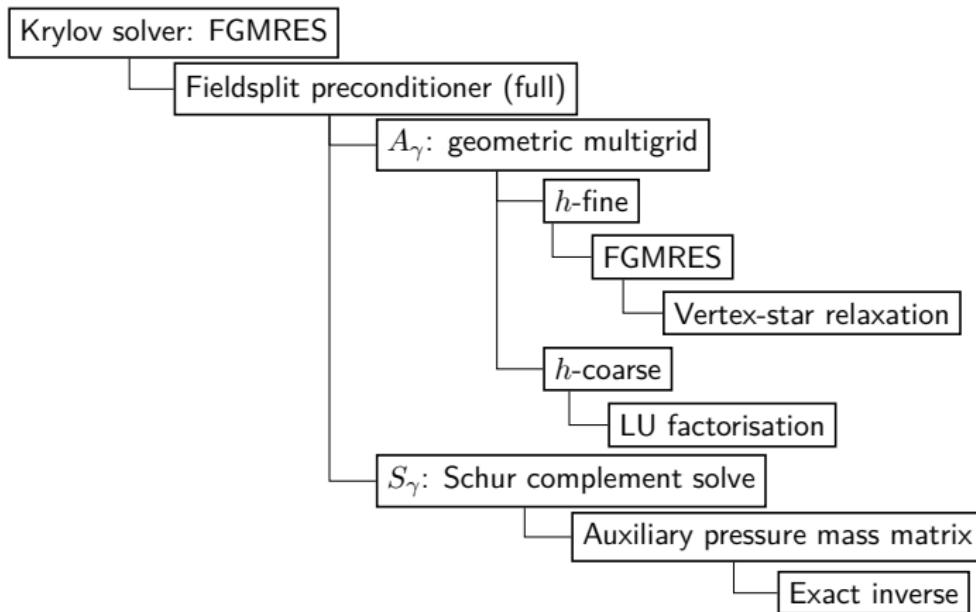


Good news

Scott–Vogelius is exactly divergence-free!

Michael Vogelius

With this knowledge, our solver diagram becomes



Augmented Lagrangian multigrid solver for Stokes.

```
from firedrake import *

# Use a triangular mesh
base = UnitSquareMesh(16, 16, diagonal="crossed")
mh = MeshHierarchy(base, 1)
mesh = mh[-1]
n = FacetNormal(mesh)
(x, y) = SpatialCoordinate(mesh)

# Define Scott--Vogelius function space W
V = VectorFunctionSpace(mesh, "CG", 4)
Q = FunctionSpace(mesh, "DG", 3)
W = MixedFunctionSpace([V, Q])

# Define Reynolds number and bcs
Re = Constant(1)
bcs = [DirichletBC(W.sub(0), Constant((0, 0)), (1, 2, 3)),
       DirichletBC(W.sub(0), as_vector([16 * x**2 * (1-x)**2, 0]), (4,))]
```

```
w = Function(W, name="Solution")
(u, p) = split(w)

# Augmented Lagrangian term
gamma = Constant(10000)

# Define Lagrangian
L = (
    0.5 * inner(2/Re * sym(grad(u)), sym(grad(u)))*dx
    - inner(p, div(u))*dx
    + 0.5 * gamma * inner(div(u), div(u))*dx
)

# Optimality conditions
F = derivative(L, w)
```

```
sp = {
    'mat_type': 'nest',
    'snes_monitor': None,
    'snes_converged_reason': None,
    'snes_max_it': 20,
    'snes_atol': 1e-8,
    'snes_rtol': 1e-12,
    'snes_stol': 1e-06,
    'ksp_type': 'fgmres',
    'ksp_converged_reason': None,
    'ksp_monitor_true_residual': None,
    'ksp_max_it': 500,
    'ksp_atol': 1e-08,
    'ksp_rtol': 1e-10,
    'pc_type': 'fieldsplit',
    'pc_fieldsplit_type': 'schur',
    'pc_fieldsplit_schur_factorization_type': 'full',
```

```
'fieldsplit_0': {'ksp_convergence_test': 'skip',
                  'ksp_max_it': 1,
                  'ksp_norm_type': 'unpreconditioned',
                  'ksp_richardson_self_scale': False,
                  'ksp_type': 'richardson',
                  'pc_type': 'mg',
                  'pc_mg_type': 'full',
                  'mg_coarse_assembled_pc_type': 'lu',
                  'mg_coarse_assembled_pc_factor_mat_solver_type': 'mumps',
                  'mg_coarse_pc_python_type': 'firedrake.AssembledPC',
                  'mg_coarse_pc_type': 'python',
                  'mg_levels': {'ksp_convergence_test': 'skip',
                                'ksp_max_it': 5,
                                'ksp_type': 'fgmres',
                                'pc_python_type': 'firedrake.ASMStarPC',
                                'pc_type': 'python'}},
},
'fieldsplit_1': {'ksp_type': 'preonly',
                  'pc_python_type': '__name__ + \'.DGMassInv\' ,
                  'pc_type': 'python'},
}
```

```
class DGMassInv(PCBase):
    def initialize(self, pc):
        _, P = pc.getOperators()
        appctx = self.get_appctx(pc)
        V = dmhooks.get_function_space(pc.getDM())
        # get function spaces
        u = TrialFunction(V)
        v = TestFunction(V)
        massinv = assemble(Tensor(inner(u, v)*dx).inv)
        self.massinv = massinv.petscmat

    def update(self, pc):
        pass

    def apply(self, pc, x, y):
        self.massinv.mult(x, y)
        scaling = 2/float(Re) + float(gamma)
        y.scale(-scaling)

    def applyTranspose(self, pc, x, y):
        raise NotImplementedError("Sorry!")
```

```
# Solve problem
solve(F == 0, w, bcs, solver_parameters=sp)

# Monitor incompressibility
print(f"||div u||: {norm(div(u), 'L2'):.2e}")

# Save solutions
(u_, p_) = w.subfunctions
u_.rename("Velocity")
p_.rename("Pressure")
File("output/stokes.pvd").write(u_, p_)
```

Challenge!

QIX.1. Adapt the Scott–Vogelius Stokes solver to solve the stationary incompressible Navier–Stokes equations.

Use continuation to go from $\text{Re} = 1$ to $\text{Re} = 5000$.

What do you observe about the iteration counts (Krylov iterations per Newton step) as the Reynolds number increases?