Implementing finite element models in FEniCS

Patrick Farrell

Oxford

May 2019

FEniCS is an automated programming environment for differential equations

- C++/Python library
- Began in 2003
- Thousands of downloads/month
- Developers all over the world
- Licensed under the GNU LGPL

`http://fenicsproject.org/`

Installing FEniCS

To do the exercises for this course, you need:

- Docker [`http://www.docker.com/products/docker-toolbox`]
- FEniCS installed via `fenicsproject run`
- Paraview [`http://www.paraview.org/download`]
- A text editor, e.g. `vim` or [`http://www.sublimetext.com`]

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 a more complex solver.

From PDE to variational problem

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

$$-\int_\Omega (\Delta u) v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x$$

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

$$-\int_\Omega (\Delta u) v \,\mathrm{d}x = \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \underbrace{\int_{\partial\Omega} \frac{\partial u}{\partial n} v \,\mathrm{d}s}_{=0}$$

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

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x = \int_\Omega f v \,\mathrm{d}x$$

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

From continuous (i) to discrete (ii) problem

Choose finite dimensional subspaces of $V = H_0^1(\Omega)$:

$$V_h \subset V$$

with e.g. piecewise linear finite elements.

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

$$\int_\Omega \nabla u_h \cdot \nabla v_h \,\mathrm{d}x = \int_\Omega f v_h \,\mathrm{d}x$$

for all $v_h \in V_h$.

A test problem

We construct a test problem for which we can easily check the answer. We first define the exact solution by

$$u(x, y) = 1 + x^2 + 2y^2$$

We insert this into Poisson's equation:

$$f = -\Delta u = -\Delta(1 + x^2 + 2y^2) = -(2 + 4) = -6$$

and we know that $u = 1 + x^2 + 2y^2$ on the boundary of Ω.

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

Poisson solver in FEniCS: implementation

```
from dolfin import *

mesh = UnitSquareMesh(32, 32)
(x, y) = SpatialCoordinate(mesh)

element = FiniteElement("Lagrange", mesh.ufl_cell(), 1)
V = FunctionSpace(mesh, element)

u = Function(V)
v = TestFunction(V)

f = Constant(-6.0)
g = 1 + x**2 + 2*y**2
bc = DirichletBC(V, g, DomainBoundary())

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

solve(F == 0, u, bc)
File("poisson.pvd") << u
```

Step by step: the first line

The first line of a FEniCS program usually begins with

```
from dolfin import *
```

This imports key classes like `UnitSquareMesh`, `FunctionSpace`, `Function` and so forth, from the FEniCS user interface (DOLFIN).

Step by step: creating a mesh

Next, we create a mesh of our domain Ω:

```
mesh = UnitSquareMesh(32, 32)
```

defines a (triangular) mesh with 32 elements along each edge.

Other useful classes for creating meshes include `UnitIntervalMesh`, `UnitCubeMesh`, `UnitCircleMesh`, `UnitSphereMesh`, `RectangleMesh` and `BoxMesh`.

Complex geometries should be built in dedicated mesh generation tools and imported:

```
mesh = Mesh("complexmesh.xdmf")
```

Mesh generation is a huge subject in its own right.

Step by step: creating a function space

The following lines create our discrete function space on Ω:

```
element = FiniteElement("Lagrange", triangle, 1)
V = FunctionSpace(mesh, element)
```

The first argument specifies the family of element, while the third argument is the degree of the basis functions on the element.

Other types of elements include

- `"Discontinuous Lagrange"`,
- `"Brezzi-Douglas-Marini"`,
- `"Raviart-Thomas"`,
- `"Crouzeix-Raviart"`,
- `"Nedelec 1st kind H(curl)"`.

See `help(FiniteElement)` for a list.

Step by step: defining expressions

Next, we define an expression for the boundary value:

```
(x, y) = SpatialCoordinate(mesh)
g = 1 + x**2 + 2*y**2
```

Another way:

```
g = Expression("1 + x[0]*x[0] + 2*x[1]*x[1]",
               element=V.ufl_element())
```

This is compiled to C++.

The `Expression` class is very flexible and can be used to create complex user-defined expressions: looking up information from a database or the internet, solving a local problem for material parameters, etc.

Step by step: defining a boundary condition

The following code defines a Dirichlet boundary condition:

```
bc = DirichletBC(V, g, DomainBoundary())
```

This boundary condition states that a function in the function space defined by `V` should be equal to `g` on the domain defined by `DomainBoundary()`.

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 a Dirichlet boundary condition, a simple domain can be defined by a C++ string

```
"on_boundary" # The entire boundary, same as DomainBoundary()
```

Alternatively, domains can be defined by subclassing `SubDomain`

```
class Boundary(SubDomain):
    def inside(self, x, on_boundary):
        return on_boundary # same as DomainBoundary()
```

Other examples:

```
"near(x[0], 0.0)"
"near(x[0], 0.0) || near(x[1], 1.0)"
```

There are many more possibilities, see

```
help(SubDomain)
help(DirichletBC)
```

Step by step: defining the right-hand side

The right-hand side $f = -6$ may be defined as follows:

```
f = Expression("-6")
```

or (more efficiently) as

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

Using a `Constant` means that the intermediate C++ code won't be regenerated when the value changes.

Step by step: defining variational problems

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

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

We now have all the objects we need in order to specify the form:

```
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: post-processing

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

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

Poisson solver in FEniCS: implementation

```
from dolfin import *

mesh = UnitSquareMesh(32, 32)
(x, y) = SpatialCoordinate(mesh)

element = FiniteElement("Lagrange", mesh.ufl_cell(), 1)
V = FunctionSpace(mesh, element)

u = Function(V)
v = TestFunction(V)

f = Constant(-6.0)
g = 1 + x**2 + 2*y**2
bc = DirichletBC(V, g, DomainBoundary())

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

solve(F == 0, u, bc)
File("poisson.pvd") << u
```

FEniCS 01 Challenge!

- Run the code and look at the solution in paraview.
- Compute the norms of the error with

```
print errornorm(g, u, 'L2')
print errornorm(g, u, 'H1')
```

- Try using quadratic Lagrange elements instead of linear elements. What happens to the error?
- Formulate a problem on $\Omega = [0, 1]^3$ whose solution you know using the method of manufactured solutions.
  Change the code to solve this problem.
- More advanced question: what solvers would have optimal complexity for this problem?

L-shaped domains: meshing, singularities and adaptivity

Patrick Farrell

Oxford

May 2019

Constructing the L-shaped domain

Three classes of domains:

- Trivial: use built-in classes (`BoxMesh`, `RectangleMesh`, etc)
- Complex: use dedicated mesh generator (aircraft, engine, etc)
- Intermediate: use `mshr`, package for constructive solid geometry

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

Constructing the L-shaped domain

We want to construct

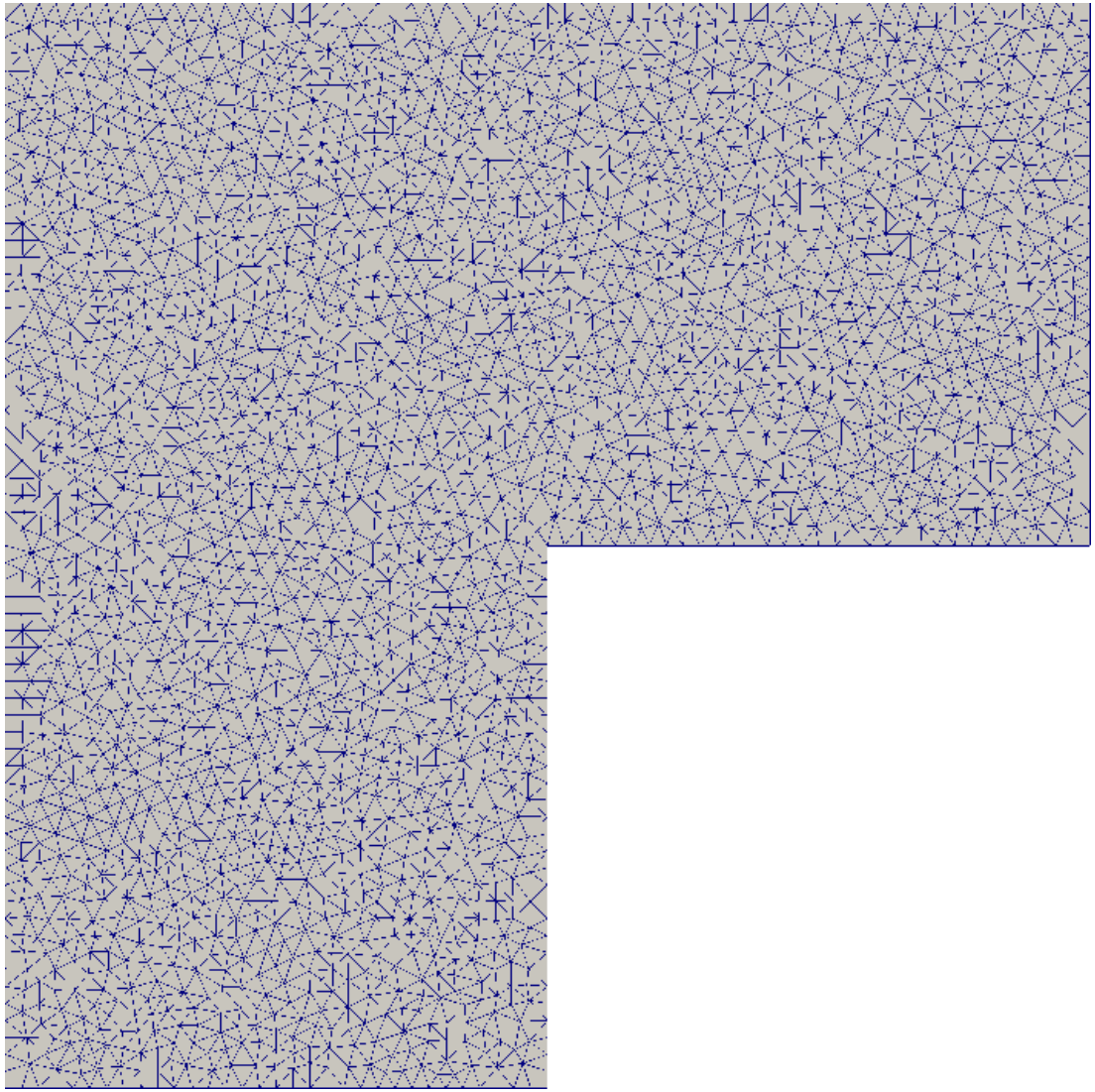
$$\Omega = [-1, 1]^2 \setminus [0, 1] \times [0, -1].$$

We can do this with

```
from dolfin import *
from mshr   import *

square = Rectangle(Point(-1, -1), Point(1, 1))
cutout = Rectangle(Point(+0, -1), Point(1, 0))
domain = square - cutout
mesh   = generate_mesh(domain, 50)
```

Constructing the L-shaped domain



Constructing the L-shaped domain

We can make a sequence of meshes with

```
from mshr import *

square = Rectangle(Point(-1, -1), Point(1, 1))
cutout = Rectangle(Point(+0, -1), Point(1, 0))
domain = square - cutout

for mesh_size in [50, 100, 200]:
    mesh   = generate_mesh(domain, mesh_size)
    # ... solve PDE here
```

Defining the boundary data

We want to construct

$$g(r, \theta) = r^{\frac{2}{3}} \sin \frac{2}{3}\theta.$$

We can do this with

```
from math import atan2
class BoundaryData(UserExpression):
    def eval(self, values, x):
        r = sqrt(x[0]**2 + x[1]**2)
        theta = atan2(x[1], x[0])

        # atan2 gives output in [-pi, pi];
        # change to [0, 2*pi]
        if theta < 0: theta += 2*pi

        values[0] = r**(2.0/3.0) * sin((2.0/3.0) * theta)

g = BoundaryData(degree=5)
```

FEniCS 02 Challenge!

Solve the Laplace equation with the given boundary data.

Examine the order of convergence as the mesh is refined (`mesh_size` $= 50, 100, 200$).

Natural boundary conditions

Patrick Farrell

Oxford

May 2019

Two kinds of boundary conditions

Two kinds of boundary conditions:

Essential boundary conditions

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

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

Natural boundary conditions

Implied by the weak form of the problem.

Poisson with Robin boundary conditions

Consider Poisson's equation with a Robin boundary condition:

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

Poisson with Robin boundary conditions

Consider Poisson's equation with a Robin boundary condition:

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

Integrating by parts, we find

$$\begin{aligned} \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} (\nabla u \cdot n) v \,\mathrm{d}s &= \int_\Omega f v \,\mathrm{d}x \\ \implies \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x + \int_{\partial\Omega} \alpha u v \,\mathrm{d}s &= \int_\Omega f v \,\mathrm{d}x \end{aligned}$$

Integrating over surfaces

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

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x + \int_{\partial\Omega} \alpha u v \,\mathrm{d}s = \int_\Omega f v \,\mathrm{d}x$$

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

Imposing mixed boundary conditions

Consider Poisson's equation with mixed boundary conditions:

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

Two ingredients need to be modified: the *Dirichlet* condition, and the *surface measure*.

```
bc = DirichletBC(V, g, "x[0] > 0 && on_boundary")
```

Integrating over subdomains

To integrate over subsets of geometric entities, we need to *color* them.

```
# Color the facets of the mesh
colors = MeshFunction("size_t", mesh, 1)
colors.set_all(0) # default to zero
CompiledSubDomain("x[0] == 0").mark(colors, 1)

# Visualise the colors
File("colors.pvd") << colors
```

```
# Create the measure
ds = Measure("ds", subdomain_data=colors)

# Integrate over facets with color 1
F = ... + inner(alpha*u, v)*ds(1)
```

FEniCS 03 Challenge!

Solve the problem

$$\begin{aligned} -\Delta u &= 1 \quad \text{in } \Omega = [0,1]^2 \\ u + \nabla u \cdot n &= 0 \quad \text{on } \partial\Omega|_{x=0} \\ u &= 0 \quad \text{on } \partial\Omega|_{x>0}\,. \end{aligned}$$

Semilinear PDEs

Patrick Farrell

Oxford

May 2019

Nonlinear PDEs

The Poisson equation models stationary heat distribution. If we take energy emitted 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 is the temperature of the surrounding medium.

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

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} (\nabla u \cdot n) v \,\mathrm{d}s = \int_\Omega f v \,\mathrm{d}x$$

$$\implies \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \beta(c^4 - u^4) v \,\mathrm{d}s = \int_\Omega f v \,\mathrm{d}x$$

Poisson with Stefan-Boltzmann

Integrating by parts, we find

$$\int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} (\nabla u \cdot n) v \,\mathrm{d}s = \int_\Omega f v \,\mathrm{d}x$$
$$\implies \int_\Omega \nabla u \cdot \nabla v \,\mathrm{d}x - \int_{\partial\Omega} \beta(c^4 - u^4) v \,\mathrm{d}s = \int_\Omega f v \,\mathrm{d}x$$

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

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 linearized system.

Properties of Newton–Kantorovich

Some facts about Newton–Kantorovich:

Computational challenge

Main cost: solving the linearized 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 linearized system.

Fast convergence

Converges quadratically if close to a regular solution.

Divergence

Can diverge if initialized far from a solution.

Properties of Newton–Kantorovich

Some facts about Newton–Kantorovich:

Computational challenge

Main cost: solving the linearized system.

Fast convergence

Converges quadratically if close to a regular solution.

Divergence

Can diverge if initialized far from a solution.

Multiple solutions

Can converge to different solutions from different initial guesses.

Using Newton's method in FEniCS

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

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

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

Using Newton's method in FEniCS

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

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

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

Setting a good initial guess is crucial for convergence!

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

FEniCS 04 Challenge!

Solve the problem

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

Hint:

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

Continuation for nonlinear problems

Patrick Farrell

Oxford

May 2019

p-Laplace equation

Consider a Poisson-type equation where the diffusivity depends on the solution itself:

$$\begin{aligned} -\nabla \cdot (\gamma(u)\nabla u) &= f \quad \text{in } \Omega \\ u &= g \quad \text{on } \partial\Omega \end{aligned}$$

where

$$\gamma(u) = (\epsilon^2 + \frac{1}{2}|\nabla u|^2)^{(p-2)/2}$$

This particular choice of γ defines the p-Laplace equation.

Variational formulation

Multiplying with a test function and integrating by parts yields

$$F(u; v) = \int_\Omega \nabla v \cdot \gamma(u) \nabla u \, \mathrm{d}x - \int_\Omega f v \, \mathrm{d}x.$$

This can be written as

```
p = Constant(5.0)
epsilon = Constant(1.0e-5)

gamma = (epsilon**2 + 0.5 * inner(grad(u), grad(u)))**((p-2)/2)
F = inner(grad(v), gamma * grad(u))*dx - inner(f, v)*dx
```

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 p (here, $p = 2$).
- While not finished:
  - Use solution for p as initial guess for $p = p + \Delta p$.
  - Increment p.

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

```
F = ...

for p_val in [2, 3, 4, 5]:
    p.assign(p_val)
    solve(F == 0, u, bc)
```

FEniCS 05 Challenge!

Solve the p-Laplace equation on $[0,1]^2$ with $p = 5$, $f = 1$, $g = 0$, $\epsilon = 10^{-5}$

(i) by tackling the problem directly from the initial guess $u = 0$;

(ii) via continuation from $p = 2$.

Compare the number of Newton iterations required by both approaches.

Hint for (i):

```
solve(F == 0, u, bc, solver_parameters={"newton_solver": {"maximum_iterations": 100}})
```

Time-dependent linear PDEs: the heat equation

Patrick Farrell

Oxford

May 2019

The heat equation

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

$$\begin{aligned} \frac{\partial u}{\partial t} - \Delta u &= f \quad \text{in } \Omega \text{ for } t > 0 \\ u &= g \quad \text{on } \partial\Omega \text{ for } t > 0 \\ u &= u^0 \text{ in } \Omega \text{ at } t = 0 \end{aligned}$$

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 ($x = (x_0, x_1, ...)$) and time (t). The initial value u^0 is a function of space only.

Time-discretization of the heat equation

There are many discretizations in time, each with different stability and efficiency properties. We will implement BDF2, a *multistep* scheme. Given

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

we will solve for u_n with

$$u_n - \frac{4}{3}u_{n-1} + \frac{1}{3}u_{n-2} = \frac{2}{3}\Delta t h(u_n, t_n).$$

With $h(u, t) = f(t) + \Delta u$, we find

$$u_n - \frac{2}{3}\Delta t \Delta u_n = \frac{2}{3}\Delta t f(t_n) + \frac{4}{3}u_{n-1} - \frac{1}{3}u_{n-2}$$

Time-discretization of the heat equation

Algorithm?

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

$$u_n - \frac{2}{3}\Delta t \Delta u_n = \frac{2}{3}\Delta t f(t_n) + \frac{4}{3}u_{n-1} - \frac{1}{3}u_{n-2}$$

Time-discretization of the heat equation

Initialization

To use a multistep method you need several past solutions, but we are only given one initial condition.

Solution: use another scheme of the same order to compute the first few solutions. We will use Crank-Nicolson. We will solve for u_n with

$$u_n - u_{n-1} = \Delta t h(\frac{u_n + u_{n-1}}{2}, \frac{t_n + t_{n-1}}{2}).$$

With $h(u, t) = f(t) + \Delta u$, we find

$$u_n - \frac{\Delta t}{2}\Delta u_n = \Delta t f(\frac{t_n + t_{n-1}}{2}) + u_{n-1} + \frac{\Delta t}{2}\Delta u_{n-1}.$$

Time-discretization of the heat equation

Algorithm

- Start with u_0 and choose a timestep $\Delta t > 0$.
- Solve for u_1:

$$u_1 - \frac{\Delta t}{2}\Delta u_1 = \Delta t f(\frac{t_1 + t_0}{2}) + u_0 + \frac{\Delta t}{2}\Delta u_0.$$

- For $n = 2, \ldots,$ solve for u^n:

$$u_n - \frac{2}{3}\Delta t \Delta u_n = \frac{2}{3}\Delta t f(t_n) + \frac{4}{3}u_{n-1} - \frac{1}{3}u_{n-2}.$$

Variational problem for the heat equation

The semi-discretized BDF2 step for u_n is

$$u_n - \frac{2}{3}\Delta t \Delta u_n = \frac{2}{3}\Delta t f(t_n) + \frac{4}{3}u_{n-1} - \frac{1}{3}u_{n-2}.$$

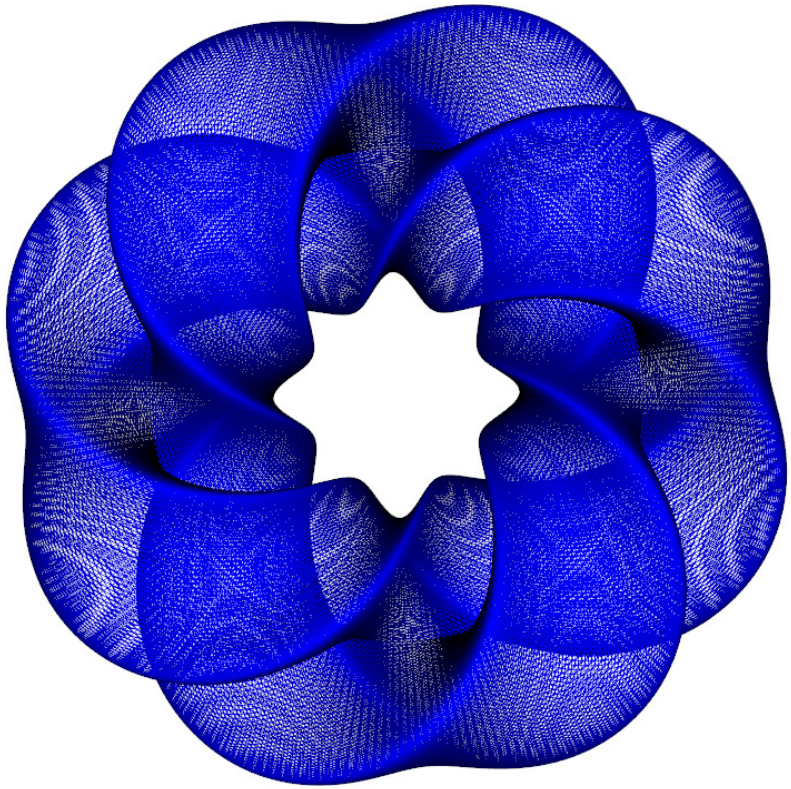
We also need to discretise this in space. With the finite element method, this yields the variational problem: find $u_n \in V_h$ such that

$$\int_\Omega u_n v \,\mathrm{d}x + \frac{2}{3}\Delta t \int_\Omega \nabla u_n \cdot \nabla v \,\mathrm{d}x = \frac{2}{3}\Delta t \int_\Omega f(t_n) v \,\mathrm{d}x + \frac{4}{3}\int_\Omega u_{n-1} v \,\mathrm{d}x - \frac{1}{3}\int_\Omega u_{n-2} v \,\mathrm{d}x$$

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

Manifolds

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



This is a *Gray's Klein bottle*.

Implementation

```
# Solve the heat equation with BDF2 and Crank-Nicolson

from dolfin import *
from decimal import Decimal

mesh = Mesh("klein.xml.gz")
V = FunctionSpace(mesh, "CG", 1)

u = Function(V)                          # u_n
u_prevs = [Function(V), Function(V)] # u_{n-1}, u_{n-2}

v = TestFunction(V)
```

Implementation

```
# Initial condition
g = interpolate(Expression("sin(x[2])*cos(x[1])", degree=2), V)

T  = Decimal("1.0")  # final time
t  = Decimal("0.0")  # current time we are solving for
h  = Decimal("0.02") # timestep size
dt = Constant(float(h)) # for use in the form
ntimestep = 0 # number of timesteps solved

u.assign(g)           # assign initial guess for solver
u_prevs[0].assign(g) # assign initial condition to u_0
```

Implementation

```
def rhs(u, v):
    return -inner(grad(u), grad(v))*dx

F_cn = (
         u*v*dx
       - u_prevs[0]*v*dx
       - dt*rhs(0.5*u + 0.5*u_prevs[0], v)
       )

F_bdf = (
          u*v*dx
        - 4.0/3.0 * u_prevs[0]*v*dx
        + 1.0/3.0 * u_prevs[1]*v*dx
        - 2.0/3.0 * dt*rhs(u, v)
        )
```

Implementation

```
output = File("heat.pvd")
output << (u, float(t))
while True:
    # Update the time we're solving for
    t += h; print("Solving for time: ", float(t))

    # check if we have enough initial data for BDF2
    if ntimestep < 1:
        solve(F_cn == 0, u)
    else:
        solve(F_bdf == 0, u)

    # Now cycle the variables
    u_prevs[1].assign(u_prevs[0])
    u_prevs[0].assign(u)

    ntimestep += 1
    output << (u, float(t))
    if t >= T: break
```

FEniCS 06 Challenge!

Run the code and look at the solution.

Change the code to use BDF3. (Initialize with two steps of Crank-Nicolson.)

Then change the code to use BDF4. (Initialize with Crank-Nicolson and BDF3.)

Mixed problems: the Stokes equations

Patrick Farrell

Oxford

May 2019


Kilmarnock
Ayr
Londonderry / Derry
Letterkenny
Northern Ireland
Bangor
United Kingdom
Middlesbrough
Darlington
Sligo
Isle of Man
Dundalk
Castlebar
Blackpool
Leeds
Hull
Preston
Drogheda
Liverpool
Sheffield
Dublin
Galway
Ireland
Naas
Bray
Nottingham
Stoke-on-Trent
Carlow
Derby
Limerick
Kilkenny
England
Tralee
Clonmel
Waterford
Cork
Wales
Headington
Swansea
London
Bristol
Cardiff
Reading
Barnstaple
Southampton

George Stokes

- Born in Sligo, Ireland
- Lucasian Professor of Mathematics, Cambridge, 1849–1903
- Member of Parliament, 1887–1892
- President of the Royal Society, 1885–1890
- Fundamental contributions to mathematics and physics.
- No one knows he is Irish.

The Stokes equations

We consider the stationary Stokes equations: find the velocity u and the pressure p such that

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

where $\epsilon(u) = \frac{1}{2}\left(\nabla u + (\nabla u)^T\right)$ and with boundary conditions

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

If viscosity ν varies with u (or p),

$$\nu = \nu(u)$$

this is a nonlinear system of partial differential equations.

The Stokes equations: variational formulation

If $u \in V$ and $p \in Q$, then $w = (u, p) \in V \times Q = W$.

Step 1

Multiply by test functions $(v, q) \in W$ and integrate first equation by parts:

$$\int_\Omega 2\nu\epsilon(u) \cdot \nabla v \,\mathrm{d}x - \int_\Omega p \nabla \cdot v \,\mathrm{d}x - \int_{\partial\Omega} (2\nu\epsilon(u) - p\,\mathrm{I}) \cdot n \cdot v \,\mathrm{d}s = 0$$

$$\int_\Omega \nabla \cdot u \, q \,\mathrm{d}x = 0$$

The Stokes equations: variational formulation

If $u \in V$ and $p \in Q$, then $w = (u, p) \in V \times Q = W$.

Step 2

Add the equations and incorporate the boundary conditions: find $(u, p) \in W = V_0 \times Q$ such that

$$\int_\Omega 2\nu\epsilon(u) \cdot \nabla v \,\mathrm{d}x - \int_\Omega p \nabla \cdot v \,\mathrm{d}x - \int_\Omega \nabla \cdot u \, q \,\mathrm{d}x + \int_{\partial\Omega_N} p_0 \, v \cdot n \,\mathrm{d}s = 0$$

for all $(v, q) \in W = V_0 \times Q$ where $V_0 = \{v \in V \text{ such that } v|_{\partial\Omega_D} = 0\}$.

Step by step: creating mixed function spaces

To create a mixed function space, first make a mixed element:

```
V = VectorElement("CG", triangle, 2)
Q = FiniteElement("CG", triangle, 1)
W = FunctionSpace(mesh, MixedElement([V, Q]))
```

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

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

... and arguments:

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

Choice of mixed function space is **crucial**.

Step by step: defining a boundary condition on a subspace

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

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

The following code defines a homogenous Dirichlet (boundary) condition on the first subspace at the part where $x_0 = 0$.

```
bc = DirichletBC(W.sub(0), (0, 0), "near(x[0], 0.0)")
```

Stokes: defining the variational form

Given

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

we can define the form `F` with

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

`dx`: integration over cells; `ds`: integration over exterior (boundary) facets.

Stokes implementation

```
# Define mesh and geometry
mesh = Mesh("dolphin.xml")
n = FacetNormal(mesh)

# Define Taylor--Hood function space W
V = VectorElement("CG", triangle, 2)
Q = FiniteElement("CG", triangle, 1)
W = FunctionSpace(mesh, MixedElement([V, Q]))

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

# Define viscosity and bcs
nu = Constant(0.2)
p0 = Expression("1.0-x[0]", degree=1)
bcs = DirichletBC(W.sub(0), (0.0, 0.0), "on_boundary && !(near(x[0], 0.0) || near(x[0], 1.0))")

# Define variational form
epsilon = sym(grad(u))
F = (2*nu*inner(epsilon, grad(v)) - div(u)*q - div(v)*p)*dx + p0*dot(v,n)*ds

# Solve problem
solve(F == 0, w, bcs)

# Plot solutions
(u, p) = w.split()
File("velocity.pvd") << u
File("pressure.pvd") << p
```

A confusion: two kinds of splits

There are two kinds of splits.

This creates a *view*:

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

`u` and `p` point to parts of `w`'s memory. Use this in your mixed form.

This creates an *independent copy*:

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

`u` and `p` are independent of `w`. Use this to plot.

FEniCS 07 Challenge!

Solve the Stokes problem on Ω defined by the `dolphin.xml` mesh, defined by the following data

$$
\begin{aligned}
-\nabla \cdot (2\nu\epsilon(u) - p\,\mathrm{I}) &= 0 && \text{in } \Omega \\
\nabla \cdot u &= 0 && \text{in } \Omega \\
-(2\nu\epsilon(u) - p\,\mathrm{I}) \cdot n &= p_0\, n && \text{on } \partial\Omega_N = \{(x_0, x_1) |\, x_0 = 0 \text{ or } x_0 = 1\} \\
p_0 &= 1 - x_0 \\
u &= 0 && \text{on } \partial\Omega_D = \partial\Omega \backslash \partial\Omega_N
\end{aligned}
$$

- Compute and plot the solutions for a constant viscosity $\nu = 0.2$.
- Use this solution as initial guess for the nonlinear viscosity

$$\nu = \nu(u) = 0.5(\nabla u : \nabla u)^{1/(2(k-1))}, k = 4.$$

Compute and plot the solutions.

Hyperelasticity

Patrick Farrell

Oxford

May 2019

Static hyperelasticity

$$
\begin{aligned}
-\nabla \cdot P &= B \quad && \text{in } \Omega \\
u &= g \quad && \text{on } \Gamma_{\mathrm{D}} \\
P \cdot n &= T \quad && \text{on } \Gamma_{\mathrm{N}}
\end{aligned}
$$

- u is the displacement (vector-valued)
- $P = P(u)$ is the first Piola–Kirchhoff stress tensor
- B is a given body force per unit volume
- g is a given boundary displacement
- T is a given boundary traction

Variational problem

Multiply by a test function $v \in \hat{V}$ and integrate by parts:

$$-\int_\Omega \nabla \cdot (P \cdot v)\,\mathrm{d}x = \int_\Omega P : \nabla v\,\mathrm{d}x - \int_{\partial\Omega} (P \cdot n) \cdot v\,\mathrm{d}s$$

Note that $v = 0$ on Γ_D and $P \cdot n = T$ on Γ_N

Find $u \in V$ such that

$$\int_\Omega P : \nabla v\,\mathrm{d}x = \int_\Omega B \cdot v\,\mathrm{d}x + \int_{\Gamma_\mathrm{N}} T \cdot v\,\mathrm{d}s$$

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

Stress–strain relations

- $F = I + \nabla u$ is the deformation gradient
- $C = F^\top F$ is the right Cauchy–Green tensor
- $E = \frac{1}{2}(C - I)$ is the Green–Lagrange strain tensor
- $W = W(E)$ is the strain energy density
- $S_{ij} = \frac{\partial W}{\partial E_{ij}}$ is the second Piola–Kirchhoff stress tensor
- $P = FS$ is the first Piola–Kirchhoff stress tensor

St. Venant–Kirchhoff strain energy function:

$$W(E) = \frac{\lambda}{2}(\mathrm{tr}(E))^2 + \mu\,\mathrm{tr}(E^2)$$

Useful FEniCS tools (I)

Loading a mesh from a file:

```
mesh = Mesh("whatever.xml")
```

Vector-valued function spaces:

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

Vector-valued Constants:

```
g = Constant((0, 0, -9.81)) # accel. due to gravity
```

Useful FEniCS tools (II)

Defining subdomains/boundaries:

```
MyBoundary = CompiledSubDomain("near(x[0], 0.0) && on_boundar
```

Marking boundaries:

```
my_boundary_1 = MyBoundary1()
my_boundary_2 = MyBoundary2()
boundaries = MeshFunction("size_t", mesh, 2)
boundaries.set_all(0)
my_boundary_1.mark(boundaries, 1)
my_boundary_2.mark(boundaries, 2)
ds = Measure("ds", subdomain_data=boundaries)
F = ...*ds(0) + ...*ds(1)
```

Useful FEniCS tools (III)

Computing derivatives of expressions:

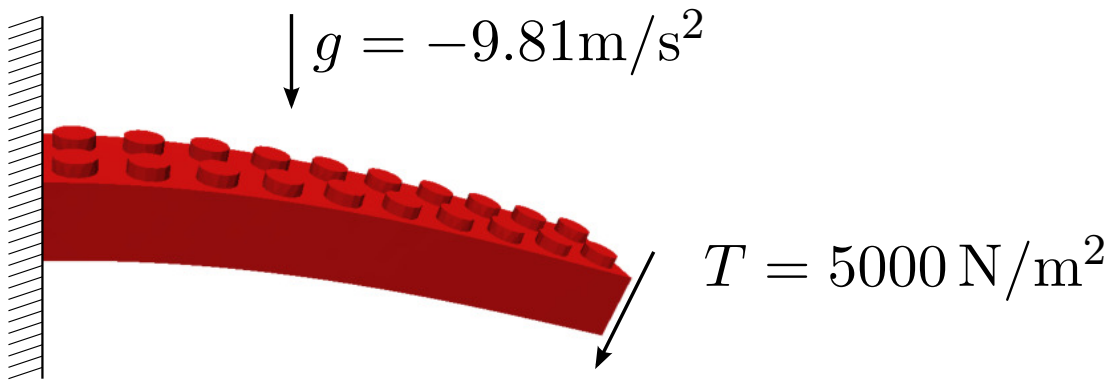
```
I = Identity(3)
F = I + grad(u)
C = F.T * F
...
E = variable(...)
W = ...
S = diff(W, E)
P = F*S
```

Computing functionals of a solution:

```
J = assemble(u[0]*dx) / assemble(Constant(1)*dx(mesh))
```

FEniCS 08 Challenge!

Compute the deflection of a regular 10×2 LEGO brick. Use the St. Venant–Kirchhoff model and assume that the LEGO brick is made of PVC plastic. The LEGO brick is subject to gravity of size $g = -9.81\ \mathrm{m/s^2}$ and a downward traction of size $5000\ \mathrm{N/m^2}$ at its right surface.



Compute the average value of the displacement in the z-direction.

The obstacle problem

Patrick Farrell

Oxford

May 2019

Obstacle problems

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 the equilibrium position $z = u(x, y)$ satisfies

$$\begin{aligned} -\nabla^2 u &= f, \\ u &= 0 \text{ on } \partial\Omega. \end{aligned}$$

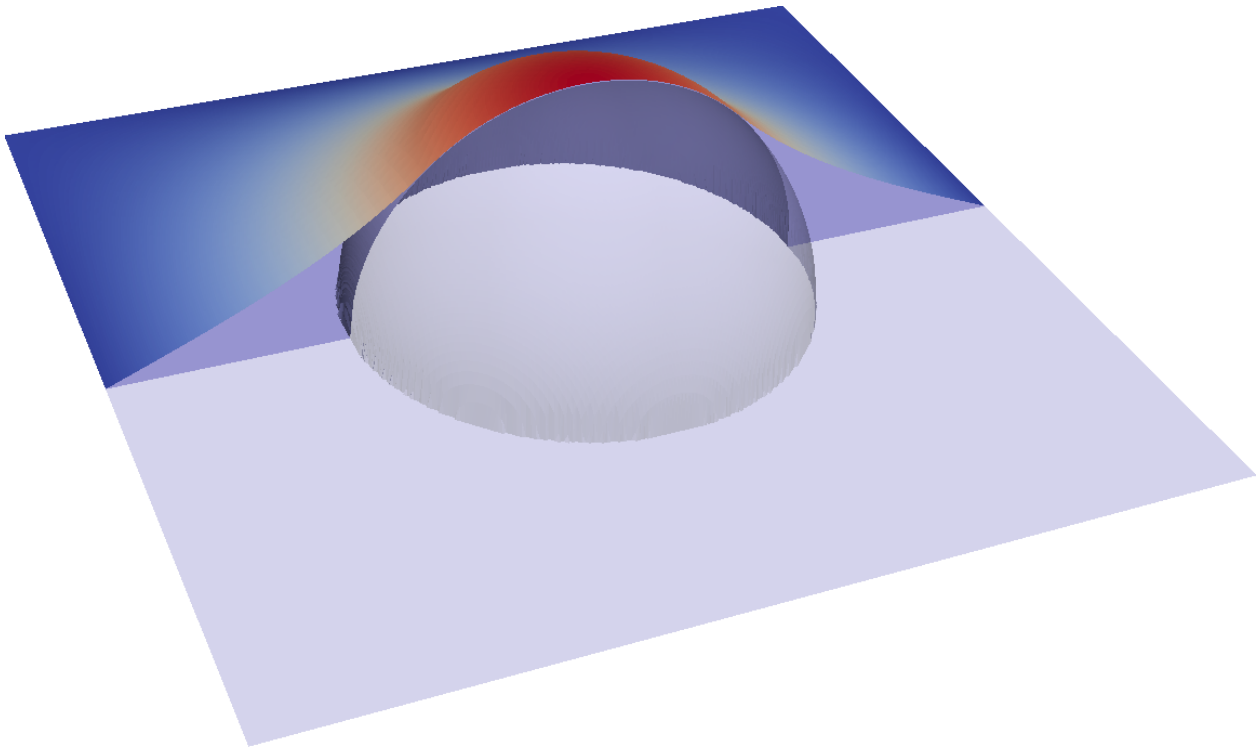
Obstacle problems

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 the equilibrium position $z = u(x, y)$ satisfies

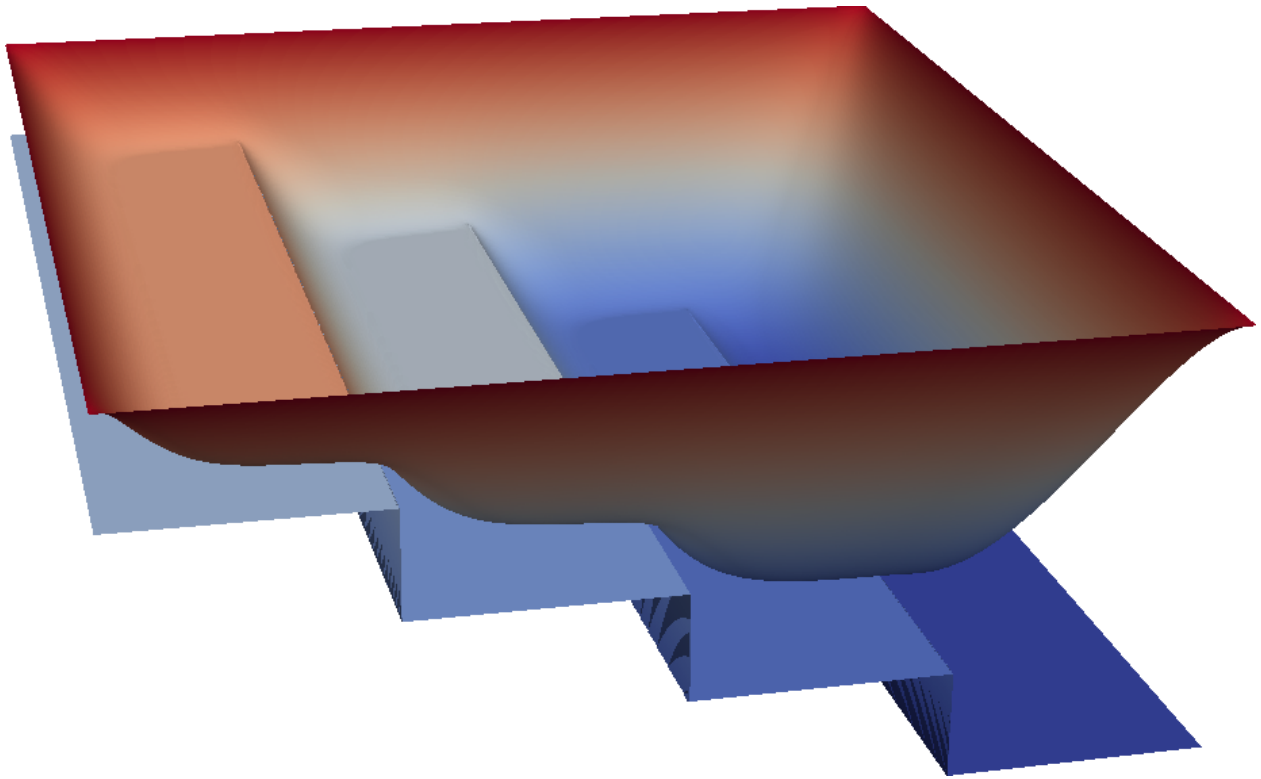
$$\begin{aligned} -\nabla^2 u &= f, \\ u &= 0 \text{ on } \partial\Omega. \end{aligned}$$

Now suppose that *an obstacle is placed underneath the membrane*. The obstacle $z = \psi(x, y)$ is continuous, differentiable and $\psi|_{\partial\Omega} \leq 0$. The task is to find a region R and solution u such that u coincides with ψ on R, and u satisfies the PDE on $\Omega \setminus R$.

Obstacle problem



Obstacle problem



Variational formulation

Let

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

Then the variational formulation is a *variational inequality*: find $u \in K_\psi$ such that

$$\int_\Omega \nabla u \cdot \nabla(v - u) \geq \int_\Omega f(v - u) \quad \forall \quad v \in K_\psi.$$

To solve this, we will reformulate the problem as a *nonsmooth rootfinding problem*.

Nonsmooth reformulation

Step 1

$$-\nabla^2 u \geq f$$
$$u \geq \psi.$$

Nonsmooth reformulation

Step 1

$$-\nabla^2 u \geq f$$
$$u \geq \psi.$$

Step 2

$$-\nabla^2 u - \lambda = f$$
$$u - \psi \geq 0, \quad \lambda \geq 0, \quad \lambda(u - \psi) = 0.$$

Nonsmooth reformulation

Step 1

$$-\nabla^2 u \geq f$$
$$u \geq \psi.$$

Step 2

$$-\nabla^2 u - \lambda = f$$
$$u - \psi \geq 0, \quad \lambda \geq 0, \quad \lambda(u - \psi) = 0.$$

Step 3

$$-\nabla^2 u - \lambda = f$$
$$\lambda - \max(\lambda - (u - \psi), 0) = 0.$$

FEniCS 09 Challenge!

Solve the Poisson obstacle problem with $\Omega = [-1, 1]^2$, $f = -10$ and

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

Eigenvalue problems

Patrick Farrell

Oxford

May 2019

Eigenvalue problems

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

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

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

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

To solve this, we will assemble matrices corresponding to the two bilinear forms.

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

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

Assembling a matrix

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

dummy = inner(Constant(1), v)*dx
bc = DirichletBC(V, 0, "on_boundary")

a = inner(grad(u), grad(v))*dx
asm = SystemAssembler(a, dummy, bc)
A = PETScMatrix(); asm.assemble(A)
```

Assembling a matrix

```
b = inner(u, v)*dx
asm = SystemAssembler(b, dummy) # no bc
B = PETScMatrix(); asm.assemble(B)
bc.zero(B)
```

Constructing the eigensolver

```
solver = SLEPcEigenSolver(A, B)
solver.parameters["solver"] = "krylov-schur"
solver.parameters["spectrum"] = "target magnitude"
solver.parameters["problem_type"] = "gen_hermitian"
solver.parameters["spectral_transform"] = "shift-and-invert"
solver.parameters["spectral_shift"] = 10.
solver.solve(1)
```

Fetching the data

```
eigenmodes = File("eigenmodes.pvd")
eigenfunction = Function(V, name="Eigenfunction")
for i in range(solver.get_number_converged()):
    (r, _, rv, _) = solver.get_eigenpair(i)

    eigenfunction.vector().zero()
    eigenfunction.vector().axpy(1, rv)
    eigenmodes << eigenfunction
```

FEniCS 10 Challenge!

Solve the eigenvalue problem for the Laplacian on the L-shaped domain.

Plot the first eigenmode. Does it look familiar?

PDE-constrained optimisation problems

Patrick Farrell

Oxford

May 2019

What is PDE-constrained optimisation?

Optimisation problems where at least one constraint is a partial differential equation.

Applications

- Shape and topology optimisation (e.g. optimal shape of a wing)
- Data assimilation (e.g. weather prediction)
- Inverse problems (e.g. petroleum exploration)
- ...

Hello World of PDE-constrained optimisation

We will solve an optimisation problem involving the Poisson equation:

$$\min_{u,m} \frac{1}{2}\int_\Omega (u - u_d)^2 \,\mathrm{d}x + \frac{\alpha}{2}\int_\Omega m^2 \,\mathrm{d}x$$

$$\begin{aligned}\text{subject to}\\ -\Delta u &= m \quad \text{in } \Omega\\ u &= 0 \quad \text{on } \partial\Omega\end{aligned}$$

This problem can be physically interpreted as: find the heating/cooling term m for which u best approximates the desired heat distribution u_d.

The regularisation term in the functional ensures existence and uniqueness for $\alpha > 0$.

The canonical abstract form

$$\min_{u,m} \ \mathcal{J}(u,m)$$

$$\text{subject to:}$$

$$\mathcal{F}(u,m) = 0,$$

with

- the objective functional $\mathcal{J}$.
- the parameter m.
- the PDE operator $\mathcal{F}$ with solution $u \in \mathcal{U}$, parametrised by $m \in \mathcal{M}$.

Oneshot solution strategy

We form the *Lagrangian* $\mathcal{L}$:

$$\mathcal{L}(u, \lambda, m) = \mathcal{J}(u, m) + \lambda^* F(u, m)$$

Optimality conditions (Karush-Kuhn-Tucker): $\nabla \mathcal{L} = 0$ at an optimum:

$$\frac{\mathrm{d}\mathcal{L}}{\mathrm{d}u} = 0, \quad \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}\lambda} = 0, \quad \frac{\mathrm{d}\mathcal{L}}{\mathrm{d}m} = 0.$$

Oneshot approach: solve these three (coupled, often nonlinear) PDEs together.

Comments on oneshot approach

The oneshot approach can be extremely fast, but very difficult to converge. (Very difficult to ensure convergence of Newton's method, and to solve the resulting linear systems.)

Oneshot approaches are mainly employed for steady PDEs; for time-dependent PDEs the reduced approach is usually much faster.

For help in implementing the reduced approach with FEniCS, see dolfin-adjoint: `http://dolfin-adjoint.org`

Forming the Lagrangian

$$\mathcal{L}(u, \lambda, m) = \frac{1}{2}\int_\Omega (u - u_d)^2 \,\mathrm{d}x + \frac{\alpha}{2}\int_\Omega m^2 \,\mathrm{d}x + \int_\Omega \nabla\lambda \cdot \nabla u - \lambda m \,\mathrm{d}x$$

```
Z = VectorFunctionSpace(mesh, "Lagrange", 1, dim=3)
z = Function(Z)
(u, lmbd, m) = split(z)

L = (0.5*inner(u-ud, u-ud)*dx
   + 0.5*alpha*inner(m, m)*dx
   + inner(grad(u), grad(lmbda))*dx
   - m*lmbda*dx)
```

KKT conditions

$$-\nabla^2 u = m$$
$$-\nabla^2 \lambda = u - u_d$$
$$\alpha m = \lambda$$

```
F = derivative(L, z, TestFunction(Z))

bcs = [DirichletBC(Z.sub(0), 0, "on_boundary"),
       DirichletBC(Z.sub(1), 0, "on_boundary")]
solve(F == 0, z, bcs)
```

FEniCS 11 Challenge, part A!

Solve the mother problem on $\Omega = [0,1]^2$ with $\alpha = 10^{-7}$, and

$$u_d(x,y) = \begin{cases} 1 & \text{if } (x,y) \in [0,0.5]^2 \\ 0 & \text{otherwise .} \end{cases}$$

Then vary the strength of regularisation parameter over $10^{-3}, 10^{-4}, \ldots, 10^{-9}$, and plot the solutions.

FEniCS 11 Challenge, part B!

We consider the following optimal control problem, after Ito and Kunisch:

$$\begin{aligned} \min_{u,m} \quad & \frac{1}{2}\|u - u_d\|^2_{L^2(\Omega)} + \frac{\alpha}{2}\|m\|^2_{L^2(\Omega)} \\ \text{subject to } & -\nabla^2 u + u^3 - u = 0 \qquad \text{on } \Omega, \\ & \nabla u \cdot n = m \qquad \text{on } \Gamma. \end{aligned}$$

This Ginzburg-Landau PDE arises in superconductivity. Solve this optimal control problem with $\Omega = [0,1] \times [0,2], \alpha = 10^{-7}, u_d = 3$.

Course accreditation

Choose a

- nonlinear BVP
- time-dependent IBVP
- variational inequality
- eigenvalue problem
- optimisation problem

that we haven't done in the course.

Course accreditation

Write a report (ten to twenty pages) describing

(1) Introduction and motivation
(2) Strong statement of the problem
(3) Variational statement of the problem
(4) FEniCS implementation
(5) Results

Submit the report (.pdf) and code (.zip) to
`patrick.farrell@maths.ox.ac.uk` by [FIXME].

Possible ideas

Sources of inspiration:

(1) Your research (preferred)!

(2) The PDE coffee table book.

(3) SIAM News.