A higher level view on FEM

Andreas Dedner Robert Klöfkorn

Introduction

Typical Variational Problem

Given:

1. a form $a\colon V \times V \to \mathbb{R}$ linear in second argument
2. a finite dimensional space $V_h \subset V$ with functions $v_h\colon \Omega \to \mathbb{R}^r$

Aim: find $u_h \in V_h$ satisfying

$$a(u_h, v_h) = 0 \ , \qquad \forall v_h \in V_h \ .$$

Examples:

1. V_h piecewise polynomial functions and

$$a(u, v) := \int_\Omega \nabla u \cdot \nabla v + (u - f)v \ .$$

2. But also some non-linear form, e.g., V_h piecewise polynomial functions, zero on boundary and

$$a(u, v) := \int_\Omega |\nabla u|^{p-1} \nabla u \cdot \nabla v - fv \ .$$

3. Or from some time discretization of $\partial_t u - \triangle u = f$:

$$a(u, v) := \int_\Omega u^{n+1} v + \tau \nabla u^{n+1} \cdot \nabla v - (u^n + \tau f)v \ .$$

Typical Variational Problem

Given:

1. a form $a\colon V \times V \to \mathbb{R}$ linear in second argument
2. a finite dimensional space $V_h \subset V$ with functions $v_h\colon \Omega \to \mathbb{R}^r$

Aim: find $u_h \in V_h$ satisfying

$$a(u_h, v_h) = 0 \ , \qquad \forall v_h \in V_h \ .$$

Examples:

1. V_h piecewise polynomial functions and

$$a(u, v) := \int_\Omega \nabla u \cdot \nabla v + (u - f)v \ .$$

2. But also some non-linear form, e.g., V_h piecewise polynomial functions, zero on boundary and

$$a(u, v) := \int_\Omega |\nabla u|^{p-1} \nabla u \cdot \nabla v - fv \ .$$

3. Or from some time discretization of $\partial_t u - \triangle u = f$:

$$a(u, v) := \int_\Omega u^{n+1} v + \tau \nabla u^{n+1} \cdot \nabla v - (u^n + \tau f)v \ .$$

Typical Variational Problem

Given:

1. a form $a\colon V \times V \to \mathbb{R}$ linear in second argument
2. a finite dimensional space $V_h \subset V$ with functions $v_h\colon \Omega \to \mathbb{R}^r$

Aim: find $u_h \in V_h$ satisfying

$$a(u_h, v_h) = 0 \ , \qquad \forall v_h \in V_h \ .$$

Examples:

1. V_h piecewise polynomial functions and

$$a(u, v) := \int_\Omega \nabla u \cdot \nabla v + (u - f)v \ .$$

2. But also some non-linear form, e.g., V_h piecewise polynomial functions, zero on boundary and

$$a(u, v) := \int_\Omega |\nabla u|^{p-1} \nabla u \cdot \nabla v - fv \ .$$

3. Or from some time discretization of $\partial_t u - \triangle u = f$:

$$a(u, v) := \int_\Omega u^{n+1} v + \tau \nabla u^{n+1} \cdot \nabla v - (u^n + \tau f)v \ .$$

Typical Variational Problem

Given:

1. a form $a\colon V \times V \to \mathbb{R}$ linear in second argument
2. a finite dimensional space $V_h \subset V$ with functions $v_h\colon \Omega \to \mathbb{R}^r$

Aim: find $u_h \in V_h$ satisfying

$$a(u_h, v_h) = 0 \ , \qquad \forall v_h \in V_h \ .$$

Examples:

1. V_h piecewise polynomial functions and
$$a(u, v) := \int_\Omega \nabla u \cdot \nabla v + (u - f)v \ .$$
2. But also some non-linear form, e.g., V_h piecewise polynomial functions, zero on boundary and
$$a(u, v) := \int_\Omega |\nabla u|^{p-1} \nabla u \cdot \nabla v - fv \ .$$
3. Or from some time discretization of $\partial_t u - \triangle u = f$:
$$a(u, v) := \int_\Omega u^{n+1} v + \tau \nabla u^{n+1} \cdot \nabla v - (u^n + \tau f)v \ .$$

Typical workflow

1. Construct a grid over Ω (discussed already)
2. Construct a discrete space V_h
3. Define the form a (using UFL)
4. Setup the *scheme* to solve the actual problem

UFL is a domain specific language (*DSL*) which was developed to describe mathematical expressions and forms.
See UFL documentation for details.

- The form is described symbolically.
- Can be manipulated in this symbolic form.
- Used to generate C++ code before assembly.

Typical workflow

1. Construct a grid over Ω (discussed already)
2. Construct a discrete space V_h
3. Define the form a (using UFL)
4. Setup the *scheme* to solve the actual problem

UFL is a domain specific language (*DSL*) which was developed to describe mathematical expressions and forms.
See UFL documentation for details.

- The form is described symbolically.
- Can be manipulated in this symbolic form.
- Used to generate C++ code before assembly.

UFL Grid Functions

Our first example uses UFL not to describe a form but a grid function:

```
from dune.grid import cartesianDomain
from dune.alugrid import aluConformGrid as leafGrid
domain = cartesianDomain([-0.5,-0.5],[0.5,0.5],[10,10])
view = leafGrid(domain)

from ufl import ( SpatialCoordinate, triangle,
                  sin, sqrt, dot )
from dune.fem.function import uflFunction
x = SpatialCoordinate(triangle)
expr = sin(20*x[0]*x[1])*sqrt(dot(x,x))
u_h = uflFunction( view, order=3, name="sin", ufl=expr )
u_h.plot()
for e in view.elements:
    hatx = e.geometry.referenceElement.position(0,0)
    print( u_h(e, hatx) )
```

Difference to `gridFunction` decorator:
C++ code is generated so no callback to Python is required,
so using a `uflFunction` does not incur any performance penalty.
Callback of course makes it possible to use complex Python statements.

UFL Grid Functions

Our first example uses UFL not to describe a form but a grid function:

```
from dune.grid import cartesianDomain
from dune.alugrid import aluConformGrid as leafGrid
domain = cartesianDomain([-0.5,-0.5],[0.5,0.5],[10,10])
view = leafGrid(domain)

from ufl import ( SpatialCoordinate, triangle,
                  sin, sqrt, dot )
from dune.fem.function import uflFunction
x = SpatialCoordinate(triangle)
expr = sin(20*x[0]*x[1])*sqrt(dot(x,x))
u_h = uflFunction( view, order=3, name="sin", ufl=expr )
u_h.plot()
for e in view.elements:
    hatx = e.geometry.referenceElement.position(0,0)
    print( u_h(e, hatx) )
```

Difference to `gridFunction` decorator:
C++ code is generated so no callback to Python is required,
so using a `uflFunction` does not incur any performance penalty.
Callback of course makes it possible to use complex Python statements.

Discrete Space

There are many available finite-element spaces but we will focus on Lagrange Finite-Elements:

$$V_h := \{v_h : \Omega \to \mathbb{R}^r : v_h \in C^0(\Omega)\ ,\ v_h|_E \in P^q(E)\ \forall E \in T_h\}$$

- $q \geq 1$: order
- $r \geq 1$: range dimension (`dimRange`)

```
from dune.fem.space import lagrange
q,r = 3,2
space = lagrange(view, order=q, dimRange=r)
```

Piecewise linear scalar functions

```
space = lagrange(view, order=1)
```

Attributes on the space:

```
print( space.dimRange, space.order )
print( "number of degrees of freedom:", space.size )
```

Discrete Functions

Associated with every discrete space V_h is an interpolation $I_h \colon V \to V_h$:

```
from ufl import as_vector
from dune.fem.space import lagrange
q,r = 3,2
space = lagrange(view, order=q, dimRange=r)
u1 = space.interpolate(as_vector( [expr,dot(x,x)] ),
                       name="u1")
u1.plot(level=3)
```

Piecewise linear scalar functions

```
space = lagrange(view, order=1)
u2 = space.interpolate(expr, name="u2")
u2.plot()
```

Setting up a form

In $a(u, v)$ we refer to u as *trial function* and v as *test function*:

```
from ufl import TrialFunction, TestFunction
u = TrialFunction(space)
v = TestFunction(space)
```

We can define *gradients* of test/trial functions and do other operations. We use *dx* to denote an integral:

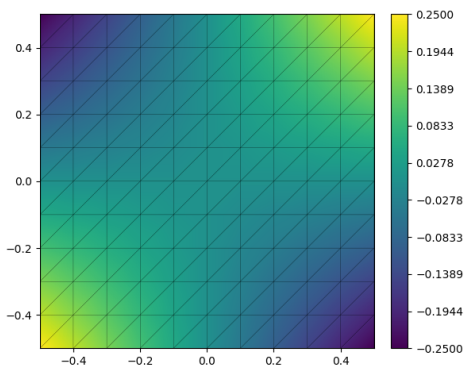
```
from ufl import grad, div, inner, dx, pi
a = inner( grad(u), grad(v) ) * dx + dot(u,v) * dx

# note that this function has normal gradients equal to
    zero on the boundary
exact = sin(pi*x[0]) + sin(pi*x[1])
b = (-div(grad(exact)) + exact) * v * dx

equation = (a == b)
```

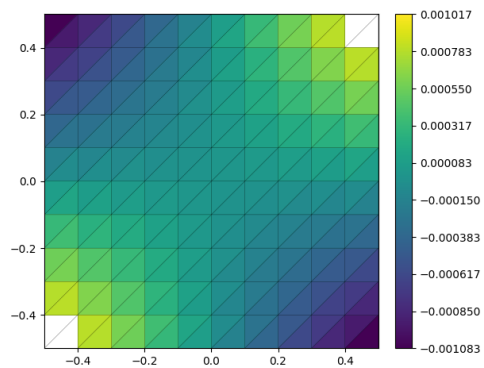
Setting up a scheme

Given a form $a\colon V_h \times W_h \to \mathbb{R}$ we can define an operator $L\colon V_h \to W_h^\star$:

$$< L[u], v >:= a(u, v) \ .$$

```
from dune.fem.space import lagrange, finiteVolume
domainSpace = lagrange(view, order=1)
rangeSpace = finiteVolume(view)

from dune.fem.operator import galerkin as galerkinOperator
u,v = TrialFunction(domainSpace), TestFunction(rangeSpace)
op = galerkinOperator( u*v*dx )

x = SpatialCoordinate(space)
uh = domainSpace.interpolate(x[0]*x[1], name="solution")
average = rangeSpace.interpolate(0,name="average")
op(uh, average)
```



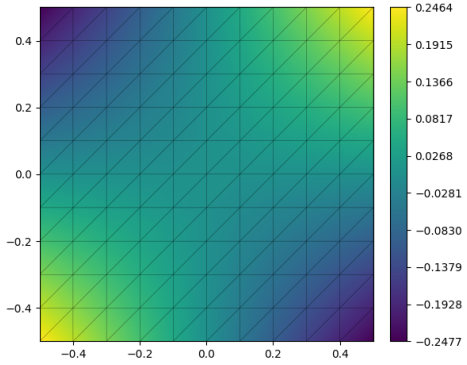


Setting up a scheme

A *scheme* is an operator that allows to compute the inverse.
So need $W_h = V_h$:

```
from dune.fem.scheme import galerkin
space = lagrange(view, order=1)
u = TrialFunction(space)
v = TestFunction(space)
equation = ( u * v * dx == sin(x[0]*x[1]) * v * dx )
scheme = galerkin(equation)

uh = space.interpolate(0, name="solution")
scheme.solve(target=uh)
uh.plot()
```

This computes the L^2 projection of $\sin(xy)$:



Laplace with Neumann boundary conditions

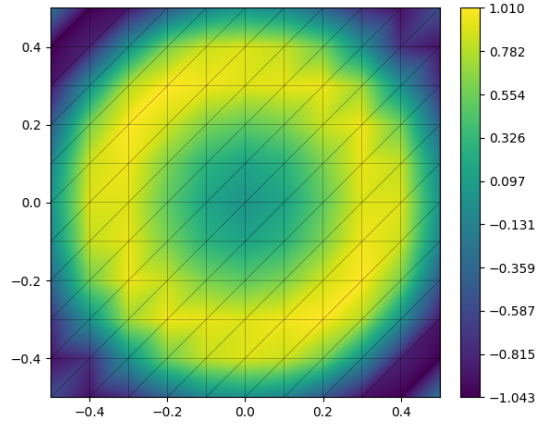
We solve $-\triangle u + u = f$ and $-\nabla u \cdot n = g_N$ in variational form:

$$\int_\Omega \nabla u \cdot \nabla v + uv = \int_\Omega fv + \int_{\partial\Omega} g_N v$$

```
from ufl import grad, div, inner, dx, pi, FacetNormal, ds
n = FacetNormal(space)
u = TrialFunction(space)
v = TestFunction(space)

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

exact = sin(4*pi*dot(x,x))
b = (-div(grad(exact)) + exact) * v * dx +
    dot(grad(exact),n) * v * ds

scheme = galerkin(a==b)

uh = space.interpolate(0, name="solution")
scheme.solve(target=uh)
```

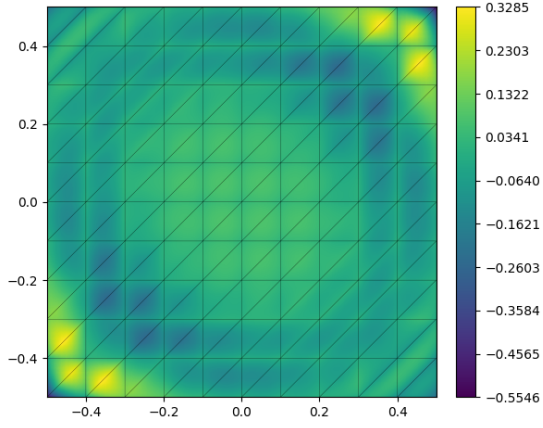
Laplace with Neumann boundary conditions

```
uh.plot()
```



uh is a grid function and can also be used in ufl expressions:

```
from dune.fem.plotting import
    plotPointData as plot
error = uh - exact # ufl expr.
plot(error, grid=view, level=3)
```



Laplace with Neumann boundary conditions

```
error_h = uflFunction( view, order=3, name="error",
          ufl=sqrt( dot(error,error) ) )
error = max( error_h(e,[1./3.,1./3.]) for e in
    view.elements)
```

For elliptic problems studying the H^1 norm is more usual

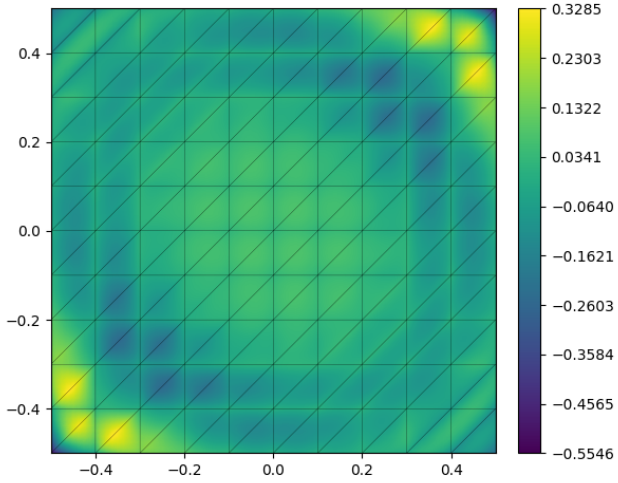
$$\|u - u_h\| := \|u - u_h\|_0 + \|\nabla(u - u_h\|_0 \,, \qquad \|v\|_0 = \sqrt{\int_\Omega v^2} \,.$$

```
from dune.fem.function import integrate
def norm(v):
    h1 = as_vector( [inner(v,v), inner(grad(v),grad(v))] )
    return np.sqrt( integrate(view,h1,order=5) )
```

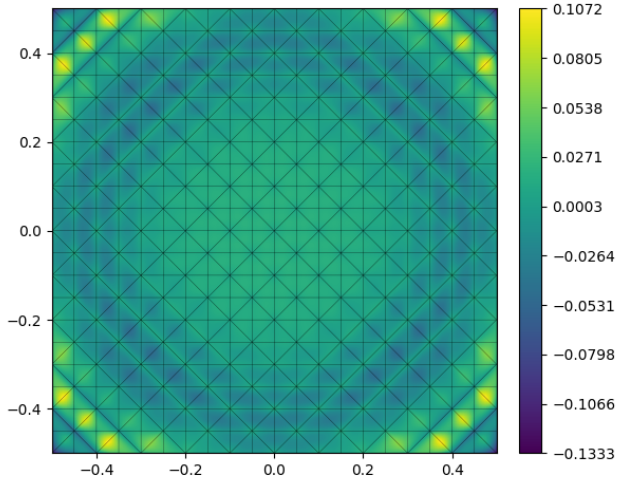
Laplace with Neumann boundary conditions

```
print(view.size(0), space.size, norm(exact-u_h) )
view.hierarchicalGrid.globalRefine(2)
scheme.solve(target=uh)
print(view.size(0), space.size, norm(exact-u_h) )
```

200 121 [0.0862595 2.7361589]



800 441 [0.0200294 1.37894141]



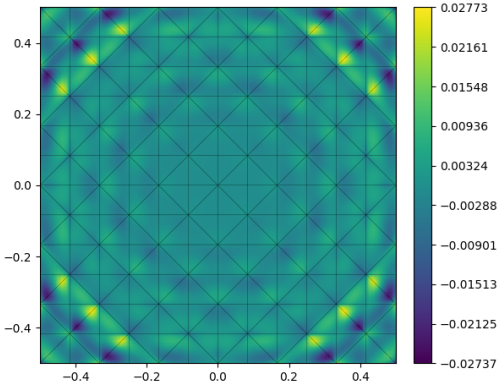
Laplace with Neumann boundary conditions

```
for order in range(1,8):
    domain = cartesianDomain([-0.5,-0.5],[0.5,0.5],[3,3])
    viewA = leafGrid(domain),
    spaceA = lagrange(viewA,order=order)
    n = FacetNormal(spaceA)
    u,v = TrialFunction(spaceA), TestFunction(spaceA)
    eqn = ( inner( grad(u), grad(v) ) * dx + dot(u,v) * dx
            == (-div(grad(exact)) + exact) * v * dx
               + dot(grad(exact),n) * v * ds )
    schemeA = galerkin(eqn, solver="cg")
    uhA = spaceA.interpolate(0, name="solution")
    for l in range(4):
        schemeA.solve(target=uhA)
        if spaceA.size > 1500: break
        viewA.hierarchicalGrid.globalRefine(2)
```

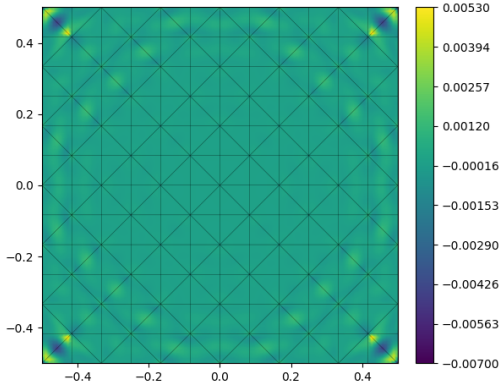
Laplace with Neumann boundary conditions

Different orders on the second level:

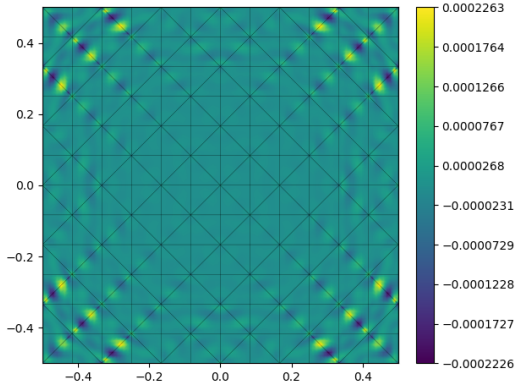
$p = 2$



$p = 3$



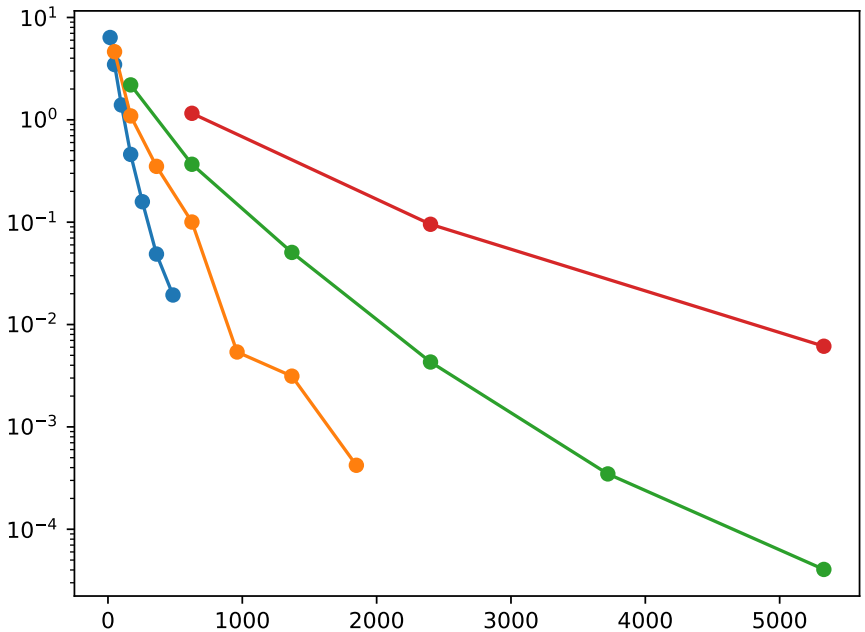
$p = 4$



From theory:

$$\|\nabla e_h\|_0 \leq Ch^p$$

Each curve is on a fixed grid (fixed h)
and varying the order p
Shown: number of dofs vs. error



A vector valued example

See example script

Heat Equation

Note: we can use grid functions within UFL expressions, so the following is the L^2 projection from above:

```
# could also use a uflFunction here
@dune.fem.function.gridFunction(view,order=2,name="sin")
def rhs(element,xLocal):
    x = elemement.geometry.toGlobal(xLocal)
    reutrn np.sin(x[0]*x[1])
equation = ( u * v * dx == rhs * v * dx )
```

More importantly we can use discrete functions in UFL expressions

```
uh_n = space.interpolate(sin(x[0]*x[1]), name="otherDF")
equation = ( u * v * dx == uh_n * v * dx )
```

This can be used to solve the heat equation:

Heat Equation

$$\partial_t u - \triangle u = 0$$

with $u(\cdot, 0) = u_0(\cdot)$ and $-\nabla u \cdot n = 0$ on the boundary.
Using an Implicit Euler method leads to (in variational form):

$$\frac{u^{n+1} - u^n}{\Delta t} - \triangle u^{n+1} = 0 \,, \quad \int_\Omega \frac{u^{n+1} - u^n}{\Delta t} v + \int_\Omega \nabla u^{n+1} \cdot \nabla v = 0$$

```
uh    = space.interpolate(initial, name='uh')
uh_n = uh.copy(name="previous")
# Note: import from dune.ufl not ufl!
from dune.ufl import Constant
dt = Constant(0.05, name="dt") # time step
a = ( dot((u-uh_n)/dt,v) + dot(grad(u),grad(v)) ) * dx
b = f*v*dx
scheme = solutionScheme(a == b, solver='cg')

time = 0
dt = 0.01 # adjust time step like this
while time <= 0.25:
    uh_n.assign(uh)
    scheme.solve(target=uh)
    time += scheme.model.dt
```

Boundary condition

- Neumann boundary conditions:
  these are added to the from using `ufl.ds`.
  To restrict the conditions to part of the boundary one can use `ufl.conditional(cond,1,0)`, e.g., for the left boundary

```
g*v * ufl.condition(x[0]<-0.499,1,0) * ds
```

- Dirichlet boundary conditions: These are so called *essential boundary conditions* that in theory have to be added to the space. We provide the information on Dirichlet boundary conditions to the scheme using the class

```
dune.ufl.DirichletBC(space,value,bndDescription)
```

Boundary condition

- Neumann boundary conditions:
  these are added to the from using `ufl.ds`.
  To restrict the conditions to part of the boundary one can use `ufl.conditional(cond,1,0)`, e.g., for the left boundary

```
g*v * ufl.condition(x[0]<-0.499,1,0) * ds
```

- Dirichlet boundary conditions: These are so called *essential boundary conditions* that in theory have to be added to the space. We provide the information on Dirichlet boundary conditions to the scheme using the class

```
dune.ufl.DirichletBC(space,value,bndDescription)
```

Dirichlet Boundary Conditions

```
from dune.ufl import DirichletBC
dbc = DirichletBC(space, exact) # this chooses the whole
    boundary
scheme = galerkin( [a==b, dbc]) # can add multiple dbc
    instances
scheme.solve(target=uh)
print( max(error_h(e,[1./3.,1./3.]) for e in
    view.elements) )
```

Tutorial on boundary conditions

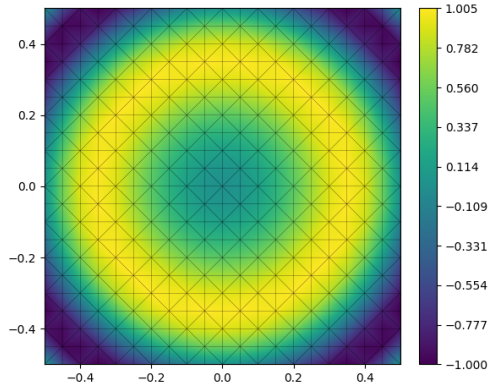
Using Numpy/Scipy

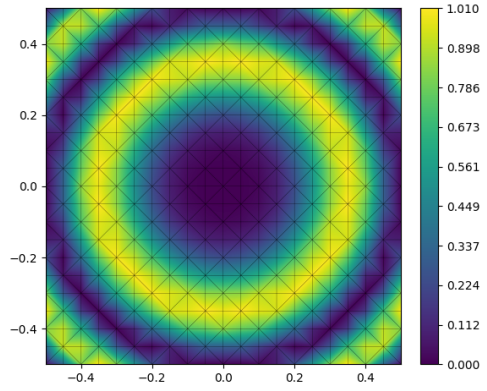
Dune-Fem provides a number of different linear algebra backends (e.g. PETSc) but it is also possible to use the underlying dof vector and sparse matrices directly in Python:

```
xVec = uh.as_numpy
```

This exposes the dof vector of u_h as a numpy vector, try for example

```
uh.plot()
xVec[:] = xVec*xVec # this changes uh's dof vector
uh.plot()
```





Using Numpy/Scipy

The linearization of a (non-linear) operator is a linear operator (a matrix):

```
import numpy as np
from scipy.sparse.linalg import spsolve as solver
from dune.fem.operator import linear as linearOperator

# obtain a linear operator by linearizing around $0$
jacobian = linearOperator(scheme)
# linearize around uh
scheme.jacobian(uh,jacobian)
xVec[:] = solver(jacobian.as_numpy, np.ones(space.size))
uh.plot()
```

Remark: `scheme.solve` uses a Newton-Krylov method by default to solve the possibly non-linear problem. There a many parameters that can be added to the scheme to customize this solver, e.g., providing tolerances, restricting the number of steps, preconditioners.

Tutorial on using different solver backends

Using Numpy/Scipy

The linearization of a (non-linear) operator is a linear operator (a matrix):

```
import numpy as np
from scipy.sparse.linalg import spsolve as solver
from dune.fem.operator import linear as linearOperator

# obtain a linear operator by linearizing around $0$
jacobian = linearOperator(scheme)
# linearize around uh
scheme.jacobian(uh,jacobian)
xVec[:] = solver(jacobian.as_numpy, np.ones(space.size))
uh.plot()
```

Remark: `scheme.solve` uses a Newton-Krylov method by default to solve the possibly non-linear problem. There a many parameters that can be added to the scheme to customize this solver, e.g., providing tolerances, restricting the number of steps, preconditioners.

Tutorial on using different solver backends

A non-linear time dependent problem

$$\partial_t u - \nabla \cdot K(\nabla u)\nabla u = f \tag{1}$$

$f = f(x,t)$ is a time dependent forcing term and the diffusion tensor is

$$K(\nabla u) = \frac{2}{1+\sqrt{1+4|\nabla u|}} \tag{2}$$

On the boundary we prescribe Neumann boundary conditions and require initial conditions $u(\cdot,0) = u_0(\cdot)$.
We solve this problem in variational form using Implicit Euler in time

$$\begin{aligned} &\int_\Omega \frac{u^{n+1}-u^n}{\Delta t}\varphi + K(\nabla u^{n+1})\nabla u^{n+1}\cdot\nabla\varphi \; dx \\ &- \int_\Omega f(x,t^n+\Delta t)\varphi \; dx - \int_{\partial\Omega} g(x,t^n+\Delta t)\varphi \; ds = 0. \end{aligned} \tag{3}$$

on a domain $\Omega = [0,1]^2$. We choose f, g so that the exact solution is

$$u(x,t) = e^{-2t}\left(\frac{1}{2}(x^2+y^2) - \frac{1}{3}(x^3-y^3)\right) + 1$$

Tutorial same model with Crank-Nicholson method

A non-linear time dependent problem

This full example is provided as template for the excercises

Excercises

Chemical Reaction: Problem Descriptions

We solve a reaction diffusion advection problem involving three species with a given velocity field. The stream function ψ for the velocity field is hereby the solution to the Laplace-equation:

$$\begin{aligned} -\triangle\psi &= 0.8\pi^2 \sin(2\pi x)\, sin(2\pi y), \quad \text{in } \Omega := [0,1]^2, \\ \psi &= 0, \quad \text{on } \partial\Omega. \end{aligned}$$

With that the velocity w is given by

$$w = (-\partial_y\psi, \partial_x\psi)$$

The three species $u = (u_0, u_1, u_2)$ satisfy the PDE

$$\partial_t u + w \cdot \nabla u - \epsilon\triangle u = R(u) + S$$

where S describes some source or sink and the reaction term is given by

$$R(u) = K\big(- u_0 u_1, -u_0 u_1, 2\, u_0 u_1 - 10\, u_2\big)$$

where K is a reaction coefficient.

Chemical Reaction: Implementation Details

- First solve the stationary elliptic problem for the velocity stream function. You can use the parameter `pointvector=w` to the vtk writer to output the vector field w. You can simply define w using ufl.
- Setup the weak form for the PDE for u using `dimRange=3` when defining the Lagrange space. All components are initialized with 0, i.e., `u_h = space.interpolate([0,0,0],name="uh")`
- Use the following values for the constants

$$K = 10, \varepsilon = 0.01, \Delta t = 0.01.$$

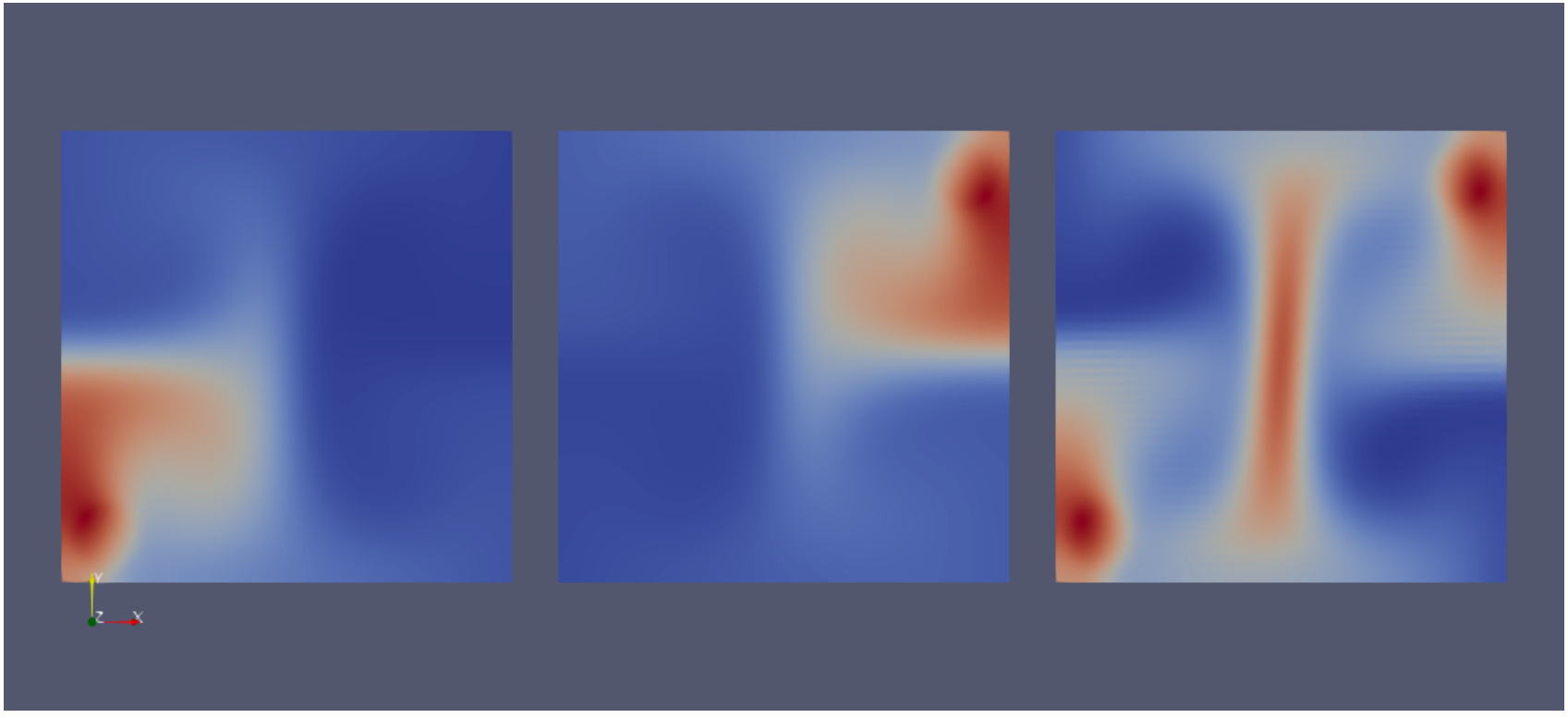
- Finally the source S is given by

$$S(x) = \big(0.1\chi_1(x), 0.1\chi_2(x), 0\big)$$

where χ_1, χ_2 are the two characteristic functions of the balls $B_R((0.1, 0.1)), B_R((0.9, 0.9))$ with radius $R = 0.075$, respectively.

- The bilinear form is not symmetric so change the solver from *cg* to *gmres*.

Chemical Reaction: Implementation Details



Heat Equation with moving Source

Problem Description

We model a heated room with a window and a moving oven by the following initial-boundary-value-problem

$$\begin{aligned} \partial_t u - K\,\Delta u &= f \quad \text{in } \Omega = (-1,1)\times(-1,1), \\ K\,\partial_\nu u &= \alpha\,(g_R - u) \quad \text{on } \Gamma_R, \\ K\,\partial_\nu u &= 0 \quad \text{on } \partial\Omega \setminus \Gamma_R, \end{aligned}$$

with the diffusion coefficient $K \equiv 1/100$, the time-dependent load function

$$f(x,\,t) = \begin{cases} \mu, & x \in B_r(P(t)) \\ 0, & \textit{else}. \end{cases}, \quad P(t) = R \begin{pmatrix} \cos(\omega\,t) \\ \sin(\omega\,t) \end{pmatrix},$$

$\mu = 8$, $r = 0.2$, $R = 0.6$, $\omega = 0.01$. The window is located at

$$\Gamma_R = (-0.5,\,0.5) \times \{1\}$$

and "connects" the outside temperature $g_R \equiv -5$ with an heat-conductivity-coefficient of $\alpha = 1.2$ to the inside. The remaining parts of the walls are considered to be ideal isolators.

Heat Equation with moving Source

Problem Description (cont.)

The weak formulation reads as follows: find $u \in H^1(\Omega)$ s.t.

$$0 = \int_\Omega K \nabla u \cdot \nabla \phi - f \, \phi \, dx + \int_{\Gamma_R} \alpha \, (u - g_R) \, \phi \, ds \quad \forall \, \phi \in H^1(\Omega).$$

or equivalently using characteristic functions for the support of f and the Robin-boundary

$$0 = \int_\Omega K \nabla u \cdot \nabla \phi - \mu \, \chi_{B_r(P(t)} \, \phi \, dx + \int_{\partial\Omega} \alpha \, (u - g_R) \, \chi_{\Gamma_R} \, \phi \, ds.$$

Tasks: Modify the provided `parabolic.py` Python program such that it discretizes the problem above. Use a uniform mesh with at least 20×20 cells and run the simulation with time-step-size $\Delta t = 0.05$ on the time interval $(0, 100]$. In each time-step compute the average temperature for a hypothetic quadratic desk with edge length 0.4 with is located in the lower left corner of the room.

Heat Equation with moving Source

Implementation Aspects

- Possible implementation of χ_{Γ_R}

```
chiWindow = ufl.conditional(abs(x[1]-1.0) < 1e-8, 1, 0)\
  * ufl.conditional(abs(x[0]) < windowWidth, 1, 0)
```

- Characteristic function for the “desk”

```
chiDesk = ufl.conditional(abs(x[0] + 0.8) < 0.2, 1, 0)\
  * ufl.conditional(abs(x[1] + 0.8) < 0.2, 1, 0)
```

- Integrating over domains

```
deskTemperature =\
  dune.fem.function.integrate(gridView, uh * chiDesk, order=1)
deskTemperature /= deskArea
```

Heat Equation with moving Source