My first FenicsX program

This notebook gives a small introduction to FenicsX, an open-source FEM library. We use the Python API of FenicsX to solve an example PDE problem.

Learning goals

- Understand the basic structure of a FenicsX program
- Learn the basic building blocks of modern FEM software: **mesh**, **function space**, **boundary conditions**, **weak form**, and **solvers**

Poisson model

In this example, we solve the classical Poisson equation for a scalar u,

$$-\Delta u = f \text{ in } \Omega = [0,\,1] \times [0,\,1]\,, \quad u = g \text{ on } \partial\Omega \quad ,$$

where Ω is the domain of interest, f is a given scalar source term, and g is a Dirichlet boundary condition.

Structure

Below, we learn how to

1. create a mesh for a simple domain
2. pick discrete FEM function spaces (*which functions do we use to approximate the PDE solution?*)
3. define the weak form of the PDE
4. apply boundary conditions
5. assemble and solve the linear system resulting from the discretization
6. visualize the solution

Let's go!

First, we import some of the necessary python libraries

Make sure to download and unpack the library.tar.gz file in the same directory as this notebook.

```
from mpi4py import MPI
import numpy as np
import os

CMM_DIR = os.getcwd()
os.environ["PYTHONPATH"] = f"{CMM_DIR}:{os.environ.get('PYTHONPATH', '')}"

import library.plot
import library.helpers
output_dir = library.helpers.make_unique_dir(os.path.join(CMM_DIR, 'ex01'))
```

Mesh and computational domain

The basis for our discretization is a discrete approximation Ω_h of the computational domain.

Here, he have the choice between different discretization types, e.g. quadrilateral, triangular etc.

```
from dolfinx import mesh

number_elements_x = 5
number_elements_y = 1
domain = mesh.create_unit_square(
    MPI.COMM_WORLD, number_elements_x, number_elements_y, cell_type=mesh.CellType.quadrilater
)
```

The variational problem

A key step in building finite element methods is the weak form of the PDE, upon which the discrete variational problem is based.

In all cases, the weak form is obtained by multiplying the PDE with a test function v and integrating over the domain Ω.

Further operations on the resulting integral equations, such as integration by parts, can then be applied. This is often done improve the properties of the method, for example to - reduce the required regularity (*we need less derivatives of the solution to exist*) and hence allow for lower-order (=*cheaper*) finite element spaces - introduce symmetry to the (bi-)linear forms, which is favored by many linear solvers - introduce certain natural boundary conditions, e.g. the terms that appear as boundary integrals in the weak form

A first key realization is that the weak form for a given PDE is **not** unique.

A weak form for the Poisson equation

A typical variational problem for the Poisson equation is given by

Find $u \in V$ such that

$$\int_\Omega \nabla u_h \cdot \nabla v_h \, dx - \cancel{\int_{\partial\Omega} u_h \cdot \mathbf{n} v_h \, ds} = \int_\Omega f v_h \, dx$$

for all $v \in V_0$. Here $\mathbf{n}$ is the outward normal vector on the boundary $\partial\Omega$ and V_0 is the space of functions that vanish on the boundary, hence the boundary integral vanishes.

The discrete variational problem is obtained by replacing the continous domain Ω with its discrete approximation Ω_h (*our mesh*) and the continuous trial and test spaces V and V_0 with the discrete finite element spaces V_h and $V_{h,0}$, respectively.

Technical note: By default, FenicsX test functions are zero on parts of the boundary where Dirichlet conditions are applied. Since we only use Dirichlet condiitons in this example, we don't have to worry about the boundary integral term and we omit the $V_{h,0}$ notation.

Our first finite element

The term "finite element" refers to the basis functions defined on our mesh. The idea of the finite element method is to find the best approximation of the solution u in a finite-dimensional function space V_h.

We thus have to pick a finite element function space V_h on our mesh.

Typically, the choice of the function space is crucial for the accuracy and convergence of the method (*lego block analogy*)

Let's start with a simple linear polynomial function space on the quadrilateral mesh that we created (*check out what it looks like here*)

```
# used for plotting
from dolfinx.fem import functionspace, Function, Constant, form
from basix.ufl import element


mesh_element_name = domain.topology.cell_name()
""" type of mesh element, e.g. "quadrilateral" """
basis_functions_degree = 1
""" degree of the finite basis functions """

finite_element = element(
    family="CG", cell=domain.topology.cell_name(), degree=basis_functions_degree
```

```
)
discrete_fem_space = functionspace(mesh=domain, element=finite_element)
```

Lets have a look at our beautiful mesh:

```
library.plot.mesh(functionspace=discrete_fem_space, title="Mesh", figurepath=output_dir, figu
```

```
image saved in /home/is086873/CMM/ex01/mesh.png
```

```
2025-04-23 16:49:19.813 (   0.802s) [    14E0A30E5740]vtkXOpenGLRenderWindow.:1416  WARN| bac
```

```
library.plot.display_image(image_path=os.path.join(output_dir, 'mesh.png'))
```

```
<IPython.core.display.HTML object>
```

From Math to Code: the weak form and UFL

The main advantage of software like FenicsX is that it provides a way to work in a syntax that is very close to the mathematical notation of the weak form. The underlying syntax is called **UFL** (Unified Form Language) and is a domain-specific language for finite element variational forms.

Let's see how we can write the weak form in UFL.

```
from ufl import TestFunction, TrialFunction, dot, ds, dx, grad

V_h = discrete_fem_space
""" Discrete finite element space """

u_h = TrialFunction(V_h)
""" discrete trial function """
v_h = TestFunction(V_h)
""" discrete test function """

f = Constant(domain, 0.0)
""" right-hand side source term """

# Define the weak form lhs == rhs
weak_form_lhs = dot(grad(u_h), grad(v_h)) * dx
weak_form_rhs = f * v_h * dx
```

Boundary conditions

There are a couple of ways to apply boundary conditions in FenicsX. We cover the simplest way to set Dirichlet boundary conditions.

The basic idea is to tell FenicsX where we want to apply the Dirichlet boundary condition in terms of the mesh and then let the software figure out which nodes and facets of the mesh coincide with that geometrical location.

We therefore start by defining functions to tell FenicsX what we consider to be the left and right boundary of the domain ($x = 0$ and $x = 1$).

```
# return 1 on the left boundary of the unit square, e.g when x = (x,y)[0] close to 0
def left_boundary_marker(x):
    return np.isclose(x[0], 0.0)


# return 1 on the right boundary of the unit square, e.g when x = (x,y)[0] close to 1
def right_boundary_marker(x):
    return np.isclose(x[0], 1.0)
```

We now create two boundary conditions for a constant value of $g = 1$ on the left boundary and $g = 0$ on the right boundary.

```
from dolfinx.fem import dirichletbc, locate_dofs_topological
from dolfinx.mesh import locate_entities_boundary

mesh_topology_dim = domain.topology.dim
facet_geometrical_dimension = mesh_topology_dim - 1

facets_on_left_boundary = locate_entities_boundary(
    domain, facet_geometrical_dimension, left_boundary_marker
)
dofs_on_left_boundary = locate_dofs_topological(
    V_h, facet_geometrical_dimension, facets_on_left_boundary
)

value_left = Constant(domain, 1.0)
""" value on the left boundary """
bc_left = dirichletbc(value_left, dofs_on_left_boundary, V_h)

facets_on_right_boundary = locate_entities_boundary(
    domain, facet_geometrical_dimension, right_boundary_marker
)
```

```
dofs_on_right_boundary = locate_dofs_topological(
    V_h, facet_geometrical_dimension, facets_on_right_boundary
)
value_right = Constant(domain, 0.0)
""" value on the right boundary """
bc_right = dirichletbc(value_right, dofs_on_right_boundary, V_h)

dirichlet_boundary_conditions = [bc_left, bc_right]
""" list of Dirichlet boundary conditions """
```

```
' list of Dirichlet boundary conditions '
```

Assemble and solve the linear system

We start by creating a discrete function to store our solution vector. The function lives in the same approximation space as our trial function u_h and is initialized to zero:

```
discrete_solution = Function(V_h)
""" discrete solution, a function in the finite element space. Our solution vector will be st
```

```
' discrete solution, a function in the finite element space. Our solution vector will be sto
```

Then, we create a writer for the VTK file to store our results on disk.

```
from dolfinx.io import VTKFile
from library.helpers import make_unique_dir


vtk_file_abs_path_name = os.path.join(output_dir, "simulation.pvd")
vtk_writer = VTKFile(
    domain.comm, vtk_file_abs_path_name, "w+"
)
vtk_writer.write_function(discrete_solution, t=0.0)

print("Writing solution to file " + vtk_file_abs_path_name)
```

```
Writing solution to file /home/is086873/CMM/ex01/simulation.pvd
```

This is where the magic happens

We now reap the true benefit of modern FEM backends: we can assemble the linear system and solve it in a single line of code.

```
from dolfinx.fem.petsc import LinearProblem

linear_solver_options = {"ksp_type": "preonly", "pc_type": "lu"}

problem = LinearProblem(
    weak_form_lhs,
    weak_form_rhs,
    bcs=dirichlet_boundary_conditions,
    petsc_options= linear_solver_options,
)


discrete_solution = problem.solve()

# Write the solution to disk
vtk_writer.write_function(discrete_solution, t=0.5)

vtk_writer.write_function(discrete_solution, t=1.0)
vtk_writer.close()
```

Ok… something happened … I guess? Let's see if we can visualize the solution.

Visualization

There are different ways to visualize the solution. For a check of the solution from within our python script, we can visualize the solution using `pyvista`.

The result is not super pretty but we can check that the solution indeed "looks" like a linear connection between the boundary conditions

```
library.plot.scalar_field(
    function=discrete_solution,
    title="Solution",
    figurepath=output_dir,
    figurename='solution',
    show_edges=True,
)
```

```
image saved in /home/is086873/CMM/ex01/solution.png
```

```
library.plot.display_image(image_path=os.path.join(output_dir, 'solution.png'))
```

```
<IPython.core.display.HTML object>
```

Post-processing with Paraview

We can also visualize the solution using Paraview. To do that, we open the files written to the following location

```
print(vtk_file_abs_path_name)
```

```
/home/is086873/CMM/ex01/simulation.pvd
```

Verification: Comparing with the exact solution

For the simple 1D Poisson equation, we can easily compare with the analytical solution, which is just a linear connection of the boundary values, e.g.

$$u_{\text{exact}}(x, y) = 1 - x \quad \text{for } x \in [0, 1], y \in [0, 1]$$

```
from dolfinx.fem import assemble_scalar, form
from petsc4py import PETSc

def u_exact(x):
    values = np.zeros((1 , x.shape[1]), dtype=PETSc.ScalarType)
    values[0] = 1.0 - x[0]
    return values

u_ex = Function(V_h)
u_ex.interpolate(u_exact)

L2_error = form(dot(discrete_solution - u_ex, discrete_solution - u_ex) * dx)

error_L2 = np.sqrt(domain.comm.allreduce(assemble_scalar(L2_error), op=MPI.SUM))
error_max = domain.comm.allreduce(
    np.max(discrete_solution.x.petsc_vec.array - u_ex.x.petsc_vec.array), op=MPI.MAX
)

print("L2 error with respect to the analytical solution: " + str(error_L2))
print("Maximum error at the degrees of freedom: " + str(error_max))
```

```
L2 error with respect to the analytical solution: 1.4256072914824157e-16
Maximum error at the degrees of freedom: 4.440892098500626e-16
```