dolfiny: Convenience wrappers for DOLFINx

Andreas Zilian, Michal Habera
Department of Engineering | University of Luxembourg

23 March 2021 | FEniCS'21 conference

UNIVERSITÉ DU LUXEMBOURG

Contents

Motivation | why `dolfiny`?

DOLFINx advantages

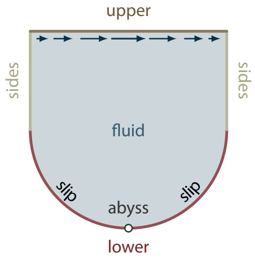
- access to low level interfaces, light core
- more flexible, more explicit
- demands bottleneck-awareness

DOLFINx shortcomings (today)

- user code often quite verbose
- increased complexity to end-user
- consistent approach to extensions?

`dolfiny`: Python, collection of wrappers, extensions + new functionality

Mesh and MeshTags | challenge: problem setup



- Gmsh to DOLFINx
- merge named non-overlapping MeshTags
- sub-classed `XDMFFile`

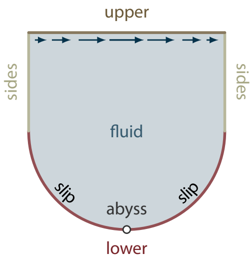
```
import mesh_cavity_gmshapi as mg

# Create the geometry/mesh of the cavity using Gmsh Python API
gmsh, tdim = mg.mesh_cavity_gmshapi()  # see demos in dolfiny repo
```

```
import dolfiny.mesh

# Get mesh and meshtags
mesh, mts = dolfiny.mesh.gmsh_to_dolfin(gmsh, tdim, prune_z=True)

# Get merged MeshTags for each codimension
subdomains, subdomains_keys = dolfiny.mesh.merge_meshtags(mts, tdim - 0)
interfaces, interfaces_keys = dolfiny.mesh.merge_meshtags(mts, tdim - 1)
markpoints, markpoints_keys = dolfiny.mesh.merge_meshtags(mts, tdim - 2)

# Define shorthands for labelled tags
fluid = subdomains_keys["fluid"]
upper = interfaces_keys["upper"]
lower = interfaces_keys["lower"]
sides = interfaces_keys["sides"]
abyss = markpoints_keys["abyss"]
```

```
import dolfiny.io

# Create output XDMF file
ofile = dolfiny.io.XDMFFile(comm, f"{name}.xdmf", "w")

# Write mesh, meshtags
ofile.write_mesh_meshtags(mesh, mts)
```

Restriction | challenge: weak interface constraints

```python
import dolfinx, ufl

# Function spaces, functions and test functions
V, P, L = # ... e.g. dolfinx.VectorFunctionSpace(mesh, ("CG", 2))
v, p, λ = # ... e.g. dolfinx.Function(V, name="v")
δv, δp, δλ =  # ... e.g. ufl.TestFunction(V)

# Integration measures
dx = ufl.Measure("dx", domain=mesh, subdomain_data=subdomains)
ds = ufl.Measure("ds", domain=mesh, subdomain_data=interfaces)

# Non-Newtonian fluid, with mu = mu(D)
D = ufl.sym(ufl.grad(v))
T = 2 * mu(D) * D - p * ufl.Identity(2)

# Weak form (as one-form), with n = ufl.FacetNormal(mesh)
f = ufl.inner(ufl.grad(δv), T) * dx + δp * ufl.div(v) * dx \
    + ufl.dot(δv, n) * λ * ds(lower) + δλ * ufl.dot(v, n) * ds(lower)
```



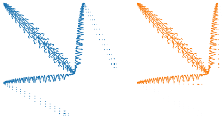

```python
import dolfiny.function

# Locate dofs: restriction
rdofsV = dolfiny.mesh.locate_dofs_topological(V, subdomains, fluid)
rdofsV = dolfiny.function.unroll_dofs(rdofsV, V.dofmap.bs)
rdofsP = dolfiny.mesh.locate_dofs_topological(P, subdomains, fluid)
rdofsL = dolfiny.mesh.locate_dofs_topological(L, interfaces, lower)

import dolfiny.restriction

# Set up restriction
r_fspaces, r_dofs = [V, P, L], [rdofsV, rdofsP, rdofsL]
restriction = dolfiny.restriction.Restriction(r_fspaces, r_dofs)
```

- restrict full function space to subset of dofs
- discrete/algebraic approach

- `SNESBlockProblem` interfaces to PETSc SNES
- custom block-wise convergence monitors
- allows MatNest and AIJ/BAIJ as matrix storage layout
- supports restrictions

```python
# Define state as (ordered) list of functions
m, δm = [v, p, λ], [δv, δp, δλ]

# Overall form (as list of forms)
F = dolfiny.function.extract_blocks(f, δm)

import dolfiny.snesblockproblem

# Create nonlinear problem: SNES
problem = dolfiny.snesblockproblem.SNESBlockProblem(F, m, restriction)

# Set/update boundary conditions
problem.bcs = # ...

# Solve nonlinear problem
problem.solve()
```

```
...
### SNES iteration 4
# sub  0 |x|=1.052e+01 |dx|=2.515e-02 |r|=1.903e-03 (v)
# sub  1 |x|=1.350e+02 |dx|=7.764e-01 |r|=2.130e-17 (p)
# sub  2 |x|=4.723e+02 |dx|=2.210e+00 |r|=2.095e-18 (λ)
# all     |x|=4.913e+02 |dx|=2.342e+00 |r|=1.903e-03

### SNES iteration 5
# sub  0 |x|=1.051e+01 |dx|=9.610e-04 |r|=3.658e-06 (v)
# sub  1 |x|=1.350e+02 |dx|=1.336e-02 |r|=1.890e-17 (p)
# sub  2 |x|=4.723e+02 |dx|=6.682e-03 |r|=1.713e-18 (λ)
# all     |x|=4.913e+02 |dx|=1.497e-02 |r|=3.658e-06
...
```



- `Restriction` uses `PETSc.Mat.createSubMatrix()`

Interpolation | challenge: expression evaluation

- interpolate UFL expressions into functions
- supports arbitrary cell-wise UFL expressions: into CG/DG (FFCx/numba)
- supports linear combination of functions for arbitrary function spaces

Cell-wise

```
import dolfiny.interpolation

# Function space for interpolated stress tensor
S = dolfinx.TensorFunctionSpace(mesh, ("DG", 1),\
                                 symmetry=True)
# Function
s = dolfinx.Function(S)

# Interpolate UFL expression T
dolfiny.interpolation.interpolate(T, s)
```

Facet-wise (soon)

```
import dolfiny.interpolation

# Function space for interpolated stress vector
S = dolfinx.VectorFunctionSpace(mesh, ("DG", 1))

# Function
s = dolfinx.Function(S)

# Interpolate UFL expression T * n
dolfiny.interpolation.interpolate(T * n, s)
```

Time-dependent forms | challenge: ease-of-use

```python
# Global time, Time step size
time, dt = dolfinx.Constant(mesh, 0.0), dolfinx.Constant(mesh, 0.1)

# Define functions representing 1st time derivative
vt, pt, λt = # ... e.g. dolfinx.Function(V, name="vt"), ...

# Define state as (ordered) list of functions
m, mt, δm = [v, p, λ], [vt, pt, λt], [δv, δp, δλ]

import dolfiny.odeint

# Time integrator
odeint = dolfiny.odeint.ODEInt(t=time, dt=dt, x=m, xt=mt)

# Weak form (as one-form), with time-dependent terms
f = # ... \
    + ufl.inner(δv, rho * vt + rho * ufl.grad(v) * v) * dx

# Overall form (as one-form)
f = odeint.discretise_in_time(f)

# Overall form (as list of forms)
F = dolfiny.function.extract_blocks(f, δm)

# Create nonlinear problem: SNES
problem = dolfiny.snesblockproblem.SNESBlockProblem(F, m, restriction)

# Time steps
for step in timesteps:
   odeint.stage()
   problem.solve()
   odeint.update()
```

- readability of forms
- single step methods

- stencil as expression
- expression-based state updates (interpolation)

- `ODEInt`, `ODEInt2` for 1st/2nd order in time ODEs
- based on modified generalised-α method

Summary and outlook

`https://github.com/michalhabera/dolfiny`

Now

- transfer Gmsh/DOLFINx
- `Restriction`
- `interpolation`, `projection`
- `SNESBlockProblem`, `SLEPcBlockProblem`
- `ODEInt`, `ODEInt2`
- supports MPI parallelism
- various demos

Future

- interpolation on facets
- flexible API for static condensation
- ...