Fireshape Documentation

Release 0.0.1

Alberto Paganini and Florian Wechsung

Mar 17, 2020

Contents

Welcome to the documentation for Fireshape.

CHAPTER 1

Introduction

1.1 Overview

Fireshape is a shape optimization toolbox for the finite element library Firedrake.

To set up a shape optimization problem, all you need to provide is the mesh on an initial guess, the shape functional, and the weak-form of its PDE-contraint.

Fireshape computes adjoints and assembles first and second derivatives for you using pyadjoint, and it solves the optimization problem using the rapid optimization library (ROL).

1.2 Features

Fireshape neatly distinguishes between the discretization of control and state variables. To discretize the control, you can choose among finite elements, B-splines, and the free-form approach. To discretize the state, you have access to all finite element spaces offered by Firedrake.

Fireshape relies on the mesh-deformation approach to update the geometry of the domain. By specifying the metric of the control space, you can decide whether meshes should be updated using Laplace or elasticity equations. In 2D, you can also use the elasticity equation corrected with Cauchy-Riemann terms, which generally leads to very high-quality meshes.

1.3 Where to go from here

If you are interested in using Fireshape, do not hesitate to get in contact. You can do so by using the contact forms available here or by sending an email to `a.paganini@leicester.ac.uk`.

You can find information on how to install Fireshape on the page *Installation*.

On the page *Example 1: Level Set*, we show how to solve a toy shape optimization problem.

On the page *Example 2: L2-tracking*, we show how to solve a shape optimization problem constrained to a linear boundary value problem.

On the page *Example 3: Kinetic energy dissipation in a pipe*, we show how to solve a shape optimization problem constrained to a nonlinear boundary value problem and a volume constraint.

On the page *Using ROL in Fireshape*, we give a very brief introduction to ROL.

CHAPTER 2

Installation

2.1 Requirements

Please install the finite element library Firedrake first.

```
curl -O https://raw.githubusercontent.com/firedrakeproject/firedrake/master/scripts/
↪firedrake-install
python3 firedrake-install
```

2.2 How to install Fireshape

Activate Firedrake's virtualenv first.

```
source path/to/firedrake/bin/activate
```

Then install the Rapid Optimization Library.

```
pip3 install --no-cache-dir roltrilinos rol
```

Now you are ready to install fireshape.

For users:

```
pip install git+https://github.com/fireshape/fireshape.git
```

For developers:

```
git clone git@github.com:fireshape/fireshape.git
cd fireshape
pip install -e .
```

CHAPTER 3

Example 1: Level Set

In this example, we show how to minimize the shape functional

$$\mathcal{J}(\Omega) = \int_\Omega f(\mathbf{x})\,\mathrm{d}\mathbf{x}\,.$$

where $f : \mathbb{R}^2 \to \mathbb{R}$ is a scalar function. In particular, we consider

$$f(x, y) = (x - 0.5)^2 + (y - 0.5)^2 - 0.5\,.$$

The domain that minimizes this shape functional is a disc of radius $1/\sqrt{2}$ centered at $(0.5, 0.5)$.

3.1 Implementing the shape functional

We implement the shape functional $\mathcal{J}$ in a python module named `levelsetfunctional.py`. In the code, we highlight the lines which characterize $\mathcal{J}$.

```python
import firedrake as fd
import fireshape as fs


class LevelsetFunctional(fs.ShapeObjective):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

        # physical mesh
        self.mesh_m = self.Q.mesh_m

        # global function defined in terms of physical coordinates
        x, y = fd.SpatialCoordinate(self.mesh_m)
        self.f = (x - 0.5)**2 + (y - 0.5)**2 - 0.5

    def value_form(self):
        # volume integral
        return self.f * fd.dx
```

3.2 Setting up and solving the problem

We set up the problem in the script `levelset.py`.

To set up the problem, we need to:

- construct the mesh of the initial guess (*Line 7*, a unit square with lower left corner in the origin)
- choose the discretization of the control (*Line 8*, Lagrangian finite elements of degree 1)
- choose the metric of the control space (*Line 9*, H^1-seminorm)

Then, we specify to save the mesh after each iteration in the file `domain.pvd` by setting the function `cb` appropriately (*Lines 13 and 16*).

Finally, we initialize the shape functional (*Line 16*), create a ROL optimization prolem (*Lines 19-43*), and solve it (*Line 44*).

```
1  import firedrake as fd
2  import fireshape as fs
3  import ROL
4  from levelsetfunctional import LevelsetFunctional
5
6  # setup problem
7  mesh = fd.UnitSquareMesh(30, 30)
8  Q = fs.FeControlSpace(mesh)
9  inner = fs.LaplaceInnerProduct(Q)
10 q = fs.ControlVector(Q, inner)
11
12 # save shape evolution in file domain.pvd
13 out = fd.File("domain.pvd")
14
15 # create objective functional
16 J = LevelsetFunctional(Q, cb=lambda: out.write(Q.mesh_m.coordinates))
17
18 # ROL parameters
19 params_dict = {
20     'Step': {
21         'Type': 'Line Search',
22         'Line Search': {
23             'Descent Method': {
24                 'Type': 'Quasi-Newton Step'
25             }
26         }
27     },
28     'General': {
29         'Secant': {
30             'Type': 'Limited-Memory BFGS',
31             'Maximum Storage': 25
32         }
33     },
34     'Status Test': {
35         'Gradient Tolerance': 1e-4,
36         'Step Tolerance': 1e-10,
37         'Iteration Limit': 30
38     }
39 }
40
41 params = ROL.ParameterList(params_dict, "Parameters")
```

(continues on next page)

(continued from previous page)

```
problem = ROL.OptimizationProblem(J, q)
solver = ROL.OptimizationSolver(problem, params)
solver.solve()
```

3.3 Result

Typing `python3 levelset.py` in the terminal returns:

```
Quasi-Newton Method with Limited-Memory BFGS
Line Search: Cubic Interpolation satisfying Strong Wolfe Conditions
  iter  value          gnorm          snorm          #fval     #grad     ls_#fval  ls_
↪#grad
  0     -3.333333e-01  2.426719e-01
  1     -3.765250e-01  1.121533e-01   2.426719e-01   2         2         1         0
  2     -3.886767e-01  8.040797e-02   2.525978e-01   3         3         1         0
  3     -3.919733e-01  2.331695e-02   6.622577e-02   4         4         1         0
  4     -3.926208e-01  4.331993e-03   5.628606e-02   5         5         1         0
  5     -3.926603e-01  2.906313e-03   1.239420e-02   6         6         1         0
  6     -3.926945e-01  9.456530e-04   2.100085e-02   7         7         1         0
  7     -3.926980e-01  3.102278e-04   6.952015e-03   8         8         1         0
  8     -3.926987e-01  1.778454e-04   3.840828e-03   9         9         1         0
  9     -3.926989e-01  9.001788e-05   2.672387e-03   10        10        1         0
Optimization Terminated with Status: Converged
```

We can inspect the result by opening the file `domain.pvd` with ParaView.

CHAPTER 4

Example 2: L2-tracking

In this example, we show how to minimize the shape functional

$$\mathcal{J}(\Omega) = \int_\Omega \big(u(\mathbf{x}) - u_t(\mathbf{x})\big)^2 \,\mathrm{d}\mathbf{x}\,.$$

where $u : \mathbb{R}^2 \to \mathbb{R}$ is the solution to the scalar boundary value problem

$$-\Delta u = 4 \quad \text{in}\,\Omega\,, \qquad u = 0 \quad \text{on}\,\partial\Omega$$

and $u_t : \mathbb{R}^2 \to \mathbb{R}$ is a target function. In particular, we consider

$$u_t(x, y) = 0.36 - (x - 0.5)^2 + (y - 0.5)^2 - 0.5\,.$$

Beside the empty set, the domain that minimizes this shape functional is a disc of radius 0.6 centered at $(0.5, 0.5)$.

4.1 Implementing the PDE constraint

We implement the boundary value problem that acts as PDE constraint in a python module named `L2tracking_PDEconstraint.py`. In the code, we highlight the lines which characterize the weak formulation of this boundary value problem.

```python
import firedrake as fd
from fireshape import PdeConstraint


class PoissonSolver(PdeConstraint):
    """A Poisson BVP with hom DirBC as PDE constraint."""
    def __init__(self, mesh_m):
        super().__init__()

        # Setup problem
        V = fd.FunctionSpace(mesh_m, "CG", 1)
```

(continues on next page)

(continued from previous page)

```
12
13          # Weak form of Poisson problem
14          u = fd.Function(V, name="State")
15          v = fd.TestFunction(V)
16          f = fd.Constant(4.)
17          self.F = (fd.inner(fd.grad(u), fd.grad(v)) - f * v) * fd.dx
18          self.bcs = fd.DirichletBC(V, 0., "on_boundary")
19
20          # PDE-solver parameters
21          self.params = {
22              "ksp_type": "cg",
23              "mat_type": "aij",
24              "pc_type": "hypre",
25              "pc_factor_mat_solver_package": "boomerang",
26              "ksp_rtol": 1e-11,
27              "ksp_atol": 1e-11,
28              "ksp_stol": 1e-15,
29          }
30
31          self.solution = u
32
33          # problem = fd.NonlinearVariationalProblem(
34          #     self.F, self.solution, bcs=self.bcs)
35          # self.solver = fd.NonlinearVariationalSolver(
36          #     problem, solver_parameters=self.params)
37
38      def solve(self):
39          super().solve()
40          fd.solve(self.F == 0, self.solution, bcs=self.bcs,
41                   solver_parameters=self.params)
42          # self.solver.solve()
```

Note: To solve the discretized variational problem, we use **CG** with a multigrid preconditioner (see `self.params` in *Lines 22-30*). For 2D problems, one can also use direct solvers.

4.2 Implementing the shape functional

We implement the shape functional $\mathcal{J}$ in a python module named `L2tracking_objective.py`. In the code, we highlight the lines which characterize $\mathcal{J}$.

```
import firedrake as fd
from fireshape import ShapeObjective
from L2tracking_PDEconstraint import PoissonSolver


class L2trackingObjective(ShapeObjective):
    """L2 tracking functional for Poisson problem."""

    def __init__(self, pde_solver: PoissonSolver, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.u = pde_solver.solution

```

(continues on next page)

(continued from previous page)

```python
        # target function, exact soln is disc of radius 0.6 centered at
        # (0.5,0.5)
        (x, y) = fd.SpatialCoordinate(self.Q.mesh_m)
        self.u_target = 0.36 - (x-0.5)*(x-0.5) - (y-0.5)*(y-0.5)

    def value_form(self):
        """Evaluate misfit functional."""
        u = self.u
        return (u - self.u_target)**2 * fd.dx
```

4.3 Setting up and solving the problem

We set up the problem in the script `L2tracking_main.py`.

To set up the problem, we

- construct the mesh of the initial guess (*Line 8*, a unit square with lower left corner in the origin)
- choose the discretization of the control (*Line 9*, Lagrangian finite elements of degree 1)
- choose the metric of the control space (*Line 10*, based on linear elasticity energy norm)
- initialize the PDE contraint on the physical mesh `mesh_m` (*Line 15*)
- specify to save the function u after each iteration in the file `u.pvd` by setting the function `cb` appropriately (*Lines 19 and 22*).
- initialize the shape functional (*Line 22*), and the reduce shape functional (*Line 23*),
- create a ROL optimization prolem (*Lines 26-49*), and solve it (*Line 50*).

```python
1  import firedrake as fd
2  import fireshape as fs
3  import ROL
4  from L2tracking_PDEconstraint import PoissonSolver
5  from L2tracking_objective import L2trackingObjective
6
7  # setup problem
8  mesh = fd.UnitSquareMesh(100, 100)
9  Q = fs.FeControlSpace(mesh)
10 inner = fs.ElasticityInnerProduct(Q)
11 q = fs.ControlVector(Q, inner)
12
13 # setup PDE constraint
14 mesh_m = Q.mesh_m
15 e = PoissonSolver(mesh_m)
16
17 # save state variable evolution in file u.pvd
18 e.solve()
19 out = fd.File("u.pvd")
20
21 # create PDEconstrained objective functional
22 J_ = L2trackingObjective(e, Q, cb=lambda: out.write(e.solution))
23 J = fs.ReducedObjective(J_, e)
24
25 # ROL parameters
26 params_dict = {
```

(continues on next page)

(continued from previous page)

```
    'Step': {
        'Type': 'Line Search',
        'Line Search': {
            'Descent Method': {
                'Type': 'Quasi-Newton Step'
            }
        },
    },
    'General': {
        'Secant': {
            'Type': 'Limited-Memory BFGS',
            'Maximum Storage': 10
        }
    },
    'Status Test': {
        'Gradient Tolerance': 1e-4,
        'Step Tolerance': 1e-5,
        'Iteration Limit': 15
    }
}
params = ROL.ParameterList(params_dict, "Parameters")
problem = ROL.OptimizationProblem(J, q)
solver = ROL.OptimizationSolver(problem, params)
solver.solve()
```

4.4 Result

Typing `python3 L2tracking_main.py` in the terminal returns:

```
Dogleg Trust-Region Solver with Limited-Memory BFGS Hessian Approximation
  iter  value          gnorm          snorm          delta          #fval     #grad   ␣
 ↪  tr_flag
  0     3.984416e-03   4.253880e-02                  4.253880e-02
  1     2.380406e-03   3.243635e-02   4.253880e-02   1.063470e-01   3         2       ␣
 ↪  0
  2     8.449408e-04   1.731817e-02   1.063470e-01   1.063470e-01   4         3       ␣
 ↪  0
  3     4.944712e-04   8.530990e-03   2.919058e-02   2.658675e-01   5         4       ␣
 ↪  0
  4     1.783406e-04   5.042658e-03   5.182347e-02   6.646687e-01   6         5       ␣
 ↪  0
  5     2.395524e-05   1.342447e-03   5.472223e-02   1.661672e+00   7         6       ␣
 ↪  0
  6     6.994156e-06   4.090116e-04   2.218345e-02   4.154180e+00   8         7       ␣
 ↪  0
  7     4.011765e-06   2.961338e-04   1.009231e-02   1.038545e+01   9         8       ␣
 ↪  0
  8     1.170509e-06   2.423493e-04   1.715142e-02   2.596362e+01   10        9       ␣
 ↪  0
  9     1.170509e-06   2.423493e-04   1.372960e-02   3.432399e-03   11        9       ␣
 ↪  2
  10    1.085373e-06   4.112702e-04   3.432399e-03   3.432399e-03   12        10      ␣
 ↪  0
  11    7.428449e-07   1.090433e-04   3.432399e-03   8.580998e-03   13        11      ␣
 ↪  0
```

(continues on next page)

(continued from previous page)

```
  12    6.755854e-07   8.358801e-05   2.034473e-03   2.145249e-02   14         12      ␣
↪  0
Optimization Terminated with Status: Converged
```

We can inspect the result by opening the file `u.pvd` with ParaView.

CHAPTER 5

Example 3: Kinetic energy dissipation in a pipe

In this example, we show how to minimize the shape functional

$$\mathcal{J}(\Omega) = \int_\Omega \nu \nabla \mathbf{u} : \nabla \mathbf{u} \,\mathrm{d}\mathbf{x}\,,$$

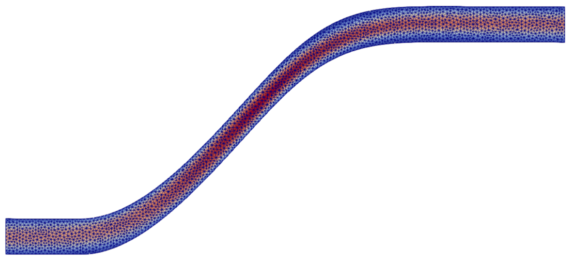
where $\mathbf{u} : \mathbb{R}^d \to \mathbb{R}$, $d = 2, 3$, is the velocity of an incompressible fluid and ν is the fluid viscosity. The fluid velocity $\mathbf{u}$ and the fluid pressure $p : \mathbb{R}^d \to \mathbb{R}$ satisfy the incompressible Navier-Stokes equations

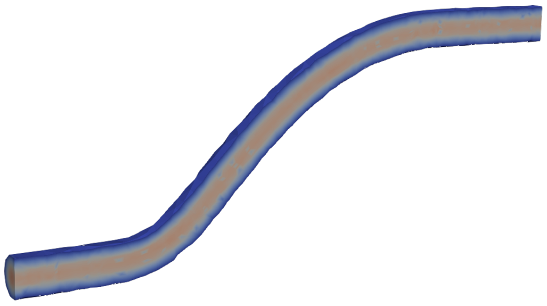
$$\begin{aligned} -\nu\Delta\mathbf{u} + \mathbf{u}\nabla\mathbf{u} + \nabla p &= 0 && \text{in } \Omega\,,\\ \operatorname{div}\mathbf{u} &= 0 && \text{in } \Omega\,,\\ \mathbf{u} &= \mathbf{g} && \text{on } \partial\Omega \setminus \Gamma\,,\\ p\mathbf{n} - \nu\nabla u \cdot \mathbf{n} &= 0 && \text{on } \Gamma\,. \end{aligned}$$

Here, $\mathbf{g}$ is given by a Poiseuille flow at the inlet and is zero on the walls of the pipe. The letter Γ denotes the outlet.

In addition to the PDE-contstraint, we enforce a volume constraint: the volume of the optimized domain should be equal to the volume of the initial domain.

5.1 Initial domain





Initial pipe design in 2D (with mesh) and 3D, and magnitude of fluid velocity.

For the 2D-example, the geometry of the initial domain is described in the following gmsh script (this has been tested with `gmsh v4.1.2`).

```
h = 1.; //width
H = 5.; //height

//lower points
Point(1) = { 0, 0, 0, 1.};
Point(2) = { 2, 0, 0, 1.}; //first spline point
Point(3) = { 4, 0, 0, 1.};
Point(4) = { 8, 6, 0, 1.};
Point(5) = {10, 6, 0, 1.};
Point(6) = {12, 6, 0, 1.}; //last spline point
Point(7) = {15, 6, 0, 1.};
//upper points
Point( 8) = {15, 7, 0, 1.};
Point( 9) = {12, 7, 0, 1.}; //first spline point
Point(10) = {10, 7, 0, 1.};
Point(11) = { 8, 7, 0, 1.};
Point(12) = { 4, 1, 0, 1.};
Point(13) = { 2, 1, 0, 1.}; //last spline point
Point(14) = { 0, 1, 0, 1.};

//edges
Line(1) = {1, 2};
BSpline(2) = {2, 3, 4, 5, 6};
Line(3) = {6, 7};
Line(4) = {7, 8};
Line(5) = {8, 9};
BSpline(6) = {9, 10, 11, 12, 13};
Line(7) = {13, 14};
Line(8) = {14,  1};

//boundary and physical curves
Curve Loop(9) = {1, 2, 3, 4, 5, 6, 7, 8};
Physical Curve("Inflow", 10) = {8};
Physical Curve("Outflow", 11) = {4};
Physical Curve("WallFixed", 12) = {1, 3, 5, 7};
Physical Curve("WallFree", 13) = {2, 6};


//domain and physical surface
Plane Surface(1) = {9};
Physical Surface("Pipe", 2) = {1};
```

The mesh can be generated typing `gmsh -2 -clscale 0.1 -format msh2 -o pipe.msh pipe2d.geo` in the terminal.

For the 3D-example, the geometry of the initial domain is described in the following gmsh script.

```
// Gmsh project created on Tue Jan 22 11:40:52 2019
SetFactory("OpenCASCADE");
Circle(1) = {0, 0, 0, 0.5, 0, 2*Pi};
Line Loop(2) = {1};
Plane Surface(1) = {2};
Point( 5) = {0, 0.0,  0.00, 1.0};
Point( 6) = {0, 0.0,  2.00, 1.0}; //first spline point
```

(continues on next page)

(continued from previous page)

```
Point( 7) = {0, 0.0,  3.00, 1.0};
Point( 8) = {0, 0.0,  4.00, 1.0};
Point( 9) = {0, 0.2,  4.75, 1.0};
Point(10) = {0, 5.0,  6.00, 1.0};
Point(11) = {0, 5.0, 10.00, 1.0};
Point(12) = {0, 5.0, 12.00, 1.0}; //last spline point
Point(13) = {0, 5.0, 15.00, 1.0};

Bezier(10) = {6, 7, 8, 9, 10, 11, 12};
Line(15) = {5, 6};
Line(16) = {12, 13};
Wire(1) = {15, 10, 16};
Extrude { Surface{1}; } Using Wire {1}
Delete{ Surface{1}; }

// When using fancy commands like "extrude", it is not quite obvious
// what the numbering of the generated lines, surfaces and volumes is
// going to be. To figure this out, we open the .geo file in the gmsh
// gui and use the Tools -> Visibility menu to find the numbers for
// each entity.

Physical Surface("Inflow", 10) = {2};
Physical Surface("Outflow", 11) = {6};
Physical Surface("WallFixed", 12) = {3, 5};
Physical Surface("WallFree", 13) = {4};
//Physical Surface("Inflow") = {2};
//Physical Surface("Outflow") = {6};
//Physical Surface("WallFixed") = {3, 5};
//Physical Surface("WallFree") = {4};
Physical Volume("PhysVol") = {1};
```

The mesh can be generated typing `gmsh -3 -clscale 0.2 -format msh2 -o pipe.msh pipe3d.geo` in the terminal (this has been tested with `gmsh v4.1.2`).

5.2 Implementing the PDE constraint

We implement the boundary value problem that acts as PDE constraint in a python module named `PDEconstraint_pipe.py`. In the code, we highlight the lines which characterize the weak formulation of this boundary value problem.

Note: The Dirichlet boundary data g depends on the dimension d (see *Lines 36-42*)

```
1  import firedrake as fd
2  from fireshape import PdeConstraint
3
4
5  class NavierStokesSolver(PdeConstraint):
6      """Incompressible Navier-Stokes as PDE constraint."""
7
8      def __init__(self, mesh_m, viscosity):
9          super().__init__()
10         self.mesh_m = mesh_m
```

(continues on next page)

(continued from previous page)

```python
        self.failed_to_solve = False  # when self.solver.solve() fail

        # Setup problem, Taylor-Hood finite elements
        self.V = fd.VectorFunctionSpace(self.mesh_m, "CG", 2) \
            * fd.FunctionSpace(self.mesh_m, "CG", 1)

        # Preallocate solution variables for state equation
        self.solution = fd.Function(self.V, name="State")
        self.testfunction = fd.TestFunction(self.V)

        # Define viscosity parameter
        self.viscosity = viscosity

        # Weak form of incompressible Navier-Stokes equations
        z = self.solution
        u, p = fd.split(z)
        test = self.testfunction
        v, q = fd.split(test)
        nu = self.viscosity  # shorten notation
        self.F = nu*fd.inner(fd.grad(u), fd.grad(v))*fd.dx - p*fd.div(v)*fd.dx\
            + fd.inner(fd.dot(fd.grad(u), u), v)*fd.dx + fd.div(u)*q*fd.dx

        # Dirichlet Boundary conditions
        X = fd.SpatialCoordinate(self.mesh_m)
        dim = self.mesh_m.topological_dimension()
        if dim == 2:
            uin = 4 * fd.as_vector([(1-X[1])*X[1], 0])
        elif dim == 3:
            rsq = X[0]**2+X[1]**2  # squared radius = 0.5**2 = 1/4
            uin = fd.as_vector([0, 0, 1-4*rsq])
        else:
            raise NotImplementedError
        self.bcs = [fd.DirichletBC(self.V.sub(0), 0., [12, 13]),
                    fd.DirichletBC(self.V.sub(0), uin, [10])]

        # PDE-solver parameters
        self.nsp = None
        self.params = {
            "snes_max_it": 10, "mat_type": "aij", "pc_type": "lu",
            "pc_factor_mat_solver_type": "superlu_dist",
            # "snes_monitor": None, "ksp_monitor": None,
        }

    def solve(self):
        super().solve()
        self.failed_to_solve = False
        u_old = self.solution.copy(deepcopy=True)
        try:
            fd.solve(self.F == 0, self.solution, bcs=self.bcs,
                     solver_parameters=self.params)
        except fd.ConvergenceError:
            self.failed_to_solve = True
            self.solution.assign(u_old)


if __name__ == "__main__":
    mesh = fd.Mesh("pipe.msh")
```

(continues on next page)

(continued from previous page)

```
    if mesh.topological_dimension() == 2:  # in 2D
        viscosity = fd.Constant(1./400.)
    elif mesh.topological_dimension() == 3:  # in 3D
        viscosity = fd.Constant(1/10.)  # simpler problem in 3D
    else:
        raise NotImplementedError
    e = NavierStokesSolver(mesh, viscosity)
    e.solve()
    print(e.failed_to_solve)
    out = fd.File("temp_PDEConstrained_u.pvd")
    out.write(e.solution.split()[0])
```

Note: The Navier-Stokes solver may fail to converge if too big an optimization step occurs in the optimization process. To address this issue, we use a trust-region algorithm as optimization solver, and we make the functional $\mathcal{J}$ return `NaN` whenever the state solver fails. This way, the trust-region method will notice that there is no improvement if the state solver fails and will thus reduce the trust-region radius.

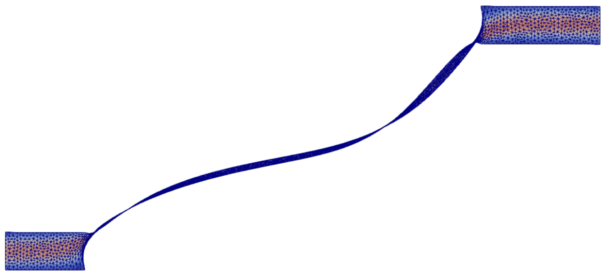


Fig. 1: If the trust-region radius is too large, the algorithm may try to evaluate the objective functional on a domain that is not feasible. In this case, the PDE-solver fails, the domain is rejected, and the trust-region radius is reduced.

5.3 Implementing the shape functional

We implement the shape functional $\mathcal{J}$ in a python file named `objective_pipe.py`. In the code, we highlight the lines which characterize $\mathcal{J}$.

```
import firedrake as fd
from fireshape import ShapeObjective
from PDEconstraint_pipe import NavierStokesSolver
import numpy as np


class PipeObjective(ShapeObjective):
    """L2 tracking functional for Poisson problem."""

    def __init__(self, pde_solver: NavierStokesSolver, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.pde_solver = pde_solver

    def value_form(self):
        """Evaluate misfit functional."""
        nu = self.pde_solver.viscosity
```

(continues on next page)

(continued from previous page)

```

        if self.pde_solver.failed_to_solve:  # return NaNs if state solve fails
            return np.nan * fd.dx(self.pde_solver.mesh_m)
        else:
            z = self.pde_solver.solution
            u, p = fd.split(z)
            return nu * fd.inner(fd.grad(u), fd.grad(u)) * fd.dx
```

5.4 Setting up and solving the problem

We set up the problem in the script `main_pipe.py`.

To set up the problem, we need to:

- load the mesh of the initial guess (*Line 9*),
- choose the discretization of the control (*Line 10*, Lagrangian finite elements of degree 1),
- choose the metric of the control space (*Line 11*, H^1-seminorm with homogeneous Dirichlet boundary conditions on fixed boundaries),
- initialize the PDE contraint on the physical mesh `mesh_m` (*Line 15-21*), choosing different viscosity parameters depending on the physical dimension *dim*
- specify to save the function $\mathbf{u}$ after each iteration in the file `solution/u.pvd` by setting the function `cb` appropriately (*Lines 24-28*),
- initialize the shape functional (*Line 31*), and the reduced shape functional (*Line 32*),
- add a regularization term to improve the mesh quality in the updated domains (*Lines 35-36*),
- specify the volume equality constraint (*Lines 39-42*)
- create a ROL optimization prolem (*Lines 45-58*), and solve it (*Line 60*). Note that the volume equality constraint is imposed in *Line 58*.

```
1  import firedrake as fd
2  import fireshape as fs
3  import fireshape.zoo as fsz
4  import ROL
5  from PDEconstraint_pipe import NavierStokesSolver
6  from objective_pipe import PipeObjective
7
8  # setup problem
9  mesh = fd.Mesh("pipe.msh")
10 Q = fs.FeControlSpace(mesh)
11 inner = fs.LaplaceInnerProduct(Q, fixed_bids=[10, 11, 12])
12 q = fs.ControlVector(Q, inner)
13
14 # setup PDE constraint
15 if mesh.topological_dimension() == 2:  # in 2D
16     viscosity = fd.Constant(1./400.)
17 elif mesh.topological_dimension() == 3:  # in 3D
18     viscosity = fd.Constant(1/10.)  # simpler problem in 3D
19 else:
20     raise NotImplementedError
21 e = NavierStokesSolver(Q.mesh_m, viscosity)
```

(continues on next page)

(continued from previous page)

```
22
23  # save state variable evolution in file u2.pvd or u3.pvd
24  if mesh.topological_dimension() == 2:  # in 2D
25      out = fd.File("solution/u2D.pvd")
26  elif mesh.topological_dimension() == 3:  # in 3D
27      out = fd.File("solution/u3D.pvd")
28
29
30  def cb():
31      return out.write(e.solution.split()[0])
32
33
34  # create PDEconstrained objective functional
35  J_ = PipeObjective(e, Q, cb=cb)
36  J = fs.ReducedObjective(J_, e)
37
38  # add regularization to improve mesh quality
39  Jq = fsz.MoYoSpectralConstraint(10, fd.Constant(0.5), Q)
40  J = J + Jq
41
42  # Set up volume constraint
43  vol = fsz.VolumeFunctional(Q)
44  initial_vol = vol.value(q, None)
45  econ = fs.EqualityConstraint([vol], target_value=[initial_vol])
46  emul = ROL.StdVector(1)
47
48  # ROL parameters
49  params_dict = {
50      'General': {'Print Verbosity': 0,  # set to 1 to understand output
51                  'Secant': {'Type': 'Limited-Memory BFGS',
52                             'Maximum Storage': 10}},
53      'Step': {'Type': 'Augmented Lagrangian',
54               'Augmented Lagrangian':
55               {'Subproblem Step Type': 'Trust Region',
56                'Print Intermediate Optimization History': False,
57                'Subproblem Iteration Limit': 10}},
58      'Status Test': {'Gradient Tolerance': 1e-2,
59                      'Step Tolerance': 1e-2,
60                      'Constraint Tolerance': 1e-1,
61                      'Iteration Limit': 10}}
62  params = ROL.ParameterList(params_dict, "Parameters")
63  problem = ROL.OptimizationProblem(J, q, econ=econ, emul=emul)
64  solver = ROL.OptimizationSolver(problem, params)
65  solver.solve()
```

Note: This problem can also be solved using Bsplines to discretize the control. For instance, one could replace *Line 10-11* with

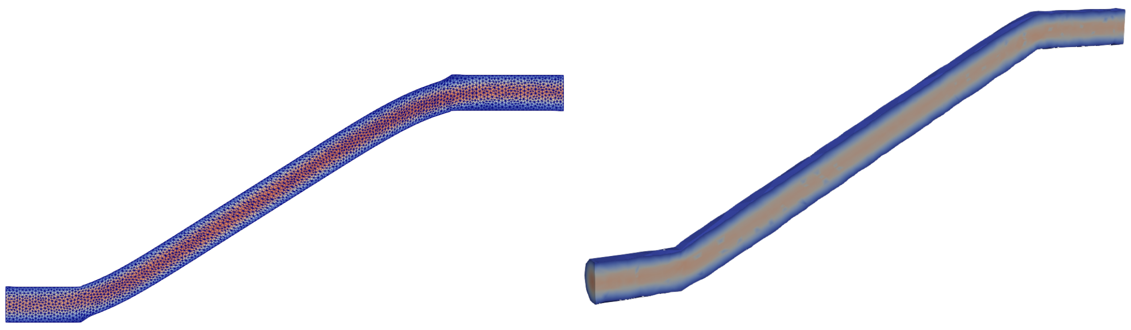
```
bbox = [(1.5, 12.), (0, 6.)]
orders = [4, 4]
levels = [4, 3]
Q = fs.BsplineControlSpace(mesh, bbox, orders, levels, boundary_regularities=[2, 0])
inner = fs.H1InnerProduct(Q)
```

In this case, the control is discretized using tensorized cubic (`order = [4, 4]`) Bsplines (roughly 2^4 in the x-direction $\times$ 2^3 in the y-direction; `levels = [4, 3]`). These Bsplines lie in the box with lower left corner $(1.5, 0)$

and upper right corner $(12., 6.)$ (`bbox = [(1.5, 12.), (0, 6.)]`). With `boundary_regularities = [2, 0]` we prescribe that the transformation vanishes for $x = 1.5$ and $x = 12$ with C^1-regularity, but it does not necessarily vanish for $y = 0$ and $y = 6$. In light of this, we do not need to specify `fixed_bids` in the inner product.

Using Bsplines to discretize the control leads to similar results.

5.5 Result



Optimized pipe design in 2D (left) and 3D (right).

For the 2D-example, typing `python3 main_pipe.py` in the terminal returns:

```
Augmented Lagrangian Solver
Subproblem Solver: Trust Region
  iter  fval           cnorm          gLnorm         snorm          penalty   feasTol␣
↪  optTol     #fval   #grad   #cval   subIter
  0     4.390787e-01   0.000000e+00   4.690997e-01                  1.00e+01  1.26e-
↪01  1.00e-02
  1     2.823114e-01   6.377197e-01   7.192107e-02   9.086354e-01   1.00e+02  1.26e-
↪01  1.00e-01  15      11      22      10
  2     3.266521e-01   9.002146e-03   2.942106e-01   4.741487e-01   1.00e+02  7.94e-
↪02  1.00e-03  23      15      32      5
  3     3.256262e-01   3.716964e-05   9.015756e-02   7.546472e-02   1.00e+02  5.01e-
↪02  1.00e-04  36      25      53      10
  4     3.251405e-01   6.469895e-06   5.961345e-02   4.412538e-02   1.00e+02  3.16e-
↪02  1.00e-04  49      35      74      10
  5     3.249782e-01   4.207885e-04   6.196224e-02   6.207929e-02   1.00e+02  2.00e-
↪02  1.00e-04  62      45      95      10
  6     3.248956e-01   1.972863e-05   8.589613e-03   9.381617e-03   1.00e+02  1.26e-
↪02  1.00e-04  75      55      116     10
Optimization Terminated with Status: Converged
```

Note: To store the terminal output in a txt file, use the bash command `python3 main_pipe.py >> output.txt`.

We can inspect the result by opening the file `u.pvd` with ParaView. We see that the difference between the volume of the initial guess and of the retrieved optimized design is roughly $2 \cdot 10^{-5}$.

For the 3D-example, typing `python3 main_pipe.py` in the terminal returns:

```
Augmented Lagrangian Solver
Subproblem Solver: Trust Region
  iter  fval            cnorm           gLnorm          snorm           penalty   feasTol␣
↪  optTol     #fval   #grad   #cval   subIter
  0     1.211321e+01   0.000000e+00   1.000000e+00                   1.50e+01  1.00e-
↪01  1.00e-02
  1     8.681223e+00   1.932438e+00   1.998650e-01   1.058693e+00   1.50e+02  8.72e-
↪02  6.65e-02  15      13      24      10
  2     1.111633e+01   2.419067e-02   3.687839e-01   1.333067e+00   1.50e+02  5.28e-
↪02  4.42e-04  23      19      36      5
  3     1.114521e+01   1.079619e-03   3.528500e-01   1.243047e-01   1.50e+02  3.20e-
↪02  1.00e-04  36      29      57      10
  4     1.113536e+01   7.521605e-04   6.820438e-02   1.229308e-01   1.50e+02  1.94e-
↪02  1.00e-04  49      39      78      10
  5     1.113373e+01   5.221684e-04   1.130481e-01   9.808759e-02   1.50e+02  1.17e-
↪02  1.00e-04  62      49      99      10
  6     1.113360e+01   3.785105e-04   5.083579e-02   7.107051e-02   1.50e+02  7.11e-
↪03  1.00e-04  75      59      120     10
  7     1.113168e+01   2.478858e-04   8.290102e-02   4.790457e-02   1.50e+02  4.30e-
↪03  1.00e-04  88      69      141     10
  8     1.113127e+01   6.115143e-05   6.973287e-02   4.889087e-02   1.50e+02  2.61e-
↪03  1.00e-04  101     79      162     10
  9     1.113080e+01   1.956210e-04   3.754803e-02   3.813478e-02   1.50e+02  1.58e-
↪03  1.00e-04  114     89      183     10
  10    1.112986e+01   5.181766e-06   3.406590e-02   2.990831e-02   1.50e+02  1.00e-
↪03  1.00e-04  127     99      204     10
Optimization Terminated with Status: Iteration Limit Exceeded
```

We can inspect the result by opening the file `u.pvd` with ParaView. We see that the difference between the volume of the initial guess and of the retrieved optimized design is roughly $5 \cdot 10^{-6}$.

CHAPTER 6

Using ROL in Fireshape

Fireshape allows solving shape optimization problems using the Rapid Optimization Library (ROL). The goal of this page is to give a very brief introduction to ROL and how to use it in Fireshape. Please note that this is not an official guide, but it's merely based on reverse engineering ROL's source code.

6.1 Basics

ROL is an optimization library that allows a complete mathematical description of the control space. More specifically, ROL allows specifying vector space operations like addition and scalar multiplication. On top of that, ROL allows specifying which norm endows the control space. Fireshape implements these methods to describe a control space of geometric transformations.

To use ROL within Fireshape, you need to declare:

- the control vector `q` and the objective function `J` (see examples in this guide),
- a dictionary `params_dict` that specifies which algorithm to employ (see below).

With these objects, you can solve the optimization problem with the following code.

```
params = ROL.ParameterList(params_dict, "Parameters")
problem = ROL.OptimizationProblem(J, q)
solver = ROL.OptimizationSolver(problem, params)
solver.solve()
```

If the optimization problem is subject to additional equality or inequality constraints, you can include these by declaring:

- the equality constraint `econ` and its multiplier `emul`
- the inequality constraint `icon`, its multiplier `imul`, and the inequality bounds `ibnd`.

Note: The variables `econ` and `icon` are of type `ROL.Constraint`, and can be instantiated usind the fireshape class `EqualityConstraint`.

The variables `emul` and `imul` are of type `ROL.StdVector`. The variable `ibnd` is of type `ROL.BoundConstraint`. For example, the following code sets the lower and the upper bound to 2 and 7, respectively.

```
lower = ROL.StdVector(1); lower[0] = 2
upper = ROL.StdVector(1); upper[0] = 7
ibnd = ROL.BoundConstraint(lower, upper)
```

With these objects, you can solve the optimization problem with the following code.

```
params = ROL.ParameterList(params_dict, "Parameters")
problem = ROL.OptimizationProblem(J, q, econ=econ, emul=emul, icon=icon, imul=imul,␣
↪ibnd=ibnd)
solver = ROL.OptimizationSolver(problem, params)
solver.solve()
```

Note: ROL allows also specifying bound constraints on the control variable, that is, constraint of the form `a<q<b`. However, such constraints are not present in shape optimization problems modelled via geometric transformations.

6.2 Choosing optimization algorithms and setting parameters

You can select the optimization algorithm and set optimization parameters by specifying the three fields: `Step`, `General`, and `Status Test` in the dictionary `params_dict`.

```
params_dict = {'Step': #set step parameters here,
                'General': #set general parameters here,
                'Status Test': #set status parameters here,
               }
```

The simplest field to set is `Status Test`. The understanding of the fields `Step` and `General` is less immediate. In this guide, we restrict ourselves to two cases: an unconstrained problem solved with a trust-region algorithm and a constrained problem solved with the augmented Lagrangian method.

Note: We prefer using a trust-region method instead of a line-search method because in the former case it is easier to deal with situations when the routine that evaluates the functional `J` fails. Such failures are usually due to failure in solving the state constraint. Among other reasons, this can happen when the control `q` is not feasible (for instance, when the underlying mesh intersects itself) or when the state constraint is nonlinear and the optimization step is too large (in which case the initial guess is not good enough).

In a nutshell, trust-region methods solve a sequence of optimization problems. In each of these, one minimizes a quadratic functional with control constraints. The idea is that the quadratic functional models the original objective functional. The control constraint limits the validity of this model to a trusted region. To construct the quadratic functional, one evaluates the original functional, its gradient, and its Hessian (or a BFGS approximation of it) in a feasible point. The minimizer to this quadratic functional is sought in a ball around that feasible point (computing this minimizer is computationally inexpensive and does not involve further evaluations of the original functional or its derivatives). Then, one evaluates the original objective functional in this minimizer and compares the *actual reduction* with the *predicted reduction*. The new control is accepted if the actual reduction is positive, that is, if there is actual reduction. Then, if there is good agreement between the actual and predicted reductions, the trust-region radius is increased. This radius is decreased if the actual reduction is not positive or the ratio between actual and predicted reductions is close to zero. From this, we understand that a safe solution to deal with failed evaluations of `J` is to store the previously computed value of `J` and, using a `try: ... except: ...` approach, return it if the new evaluation of `J` fails. This corresponds to a nonpositive actual reduction, which triggers a reduction of the trust-region radius.

Note: The following examples include all parameters that can be set for the algorithms described. However, it is not necessary to specify a field if one does not want to modify a default value.

Note: The following examples include all parameters that can be set for the algorithms described. However, it is not necessary to specify a field if one does want to modify a default value.

6.3 Setting termination criteria

This field sets the termination criteria of the algorithm. Its use is self-explanatory. We report its syntax with the default values.

```
'Status Test':{'Gradient Tolerance':1.e-6,
               'Step Tolerance':1.e-12,
               'Iteration Limit':100}
```

If the optimization problem contains equality (or inequality constraints), one can further specify the desired `Constraint Tolerance`. Its default value is `1.e-6`. In this case, an optimization algorithm has converged only if both the gradient and constraint tolerances are satisfied.

6.4 Solving an unconstrained problem with a trust-region method

To solve an unconstrained problem using a trust-region method, we can set `Step` and `General` as follows (the provided values are the default ones). To understand some of these parameters, please read the trust-region algorithm implementation.

```
'Step':{'Type':'Trust Region',
        'Trust Region':{'Initial Radius':-1, #determine initial radius with heuristics
                        'Maximum Radius':1.e8,
                        'Subproblem Solver':'Dogleg',
                        'Radius Growing Rate':2.5
                        'Step Acceptance Threshold':0.05,
                        'Radius Shrinking Threshold':0.05,
                        'Radius Growing Threshold':0.9,
                        'Radius Shrinking Rate (Negative rho)':0.0625,
                        'Radius Shrinking Rate (Positive rho)':0.25,
                        'Radius Growing Rate':2.5,
                        'Sufficient Decrease Parameter':1.e-4,
                        'Safeguard Size':100.0,
                       }
       }
'General':{'Print Verbosity':0, #set to any number >0 for increased verbosity
           'Secant':{'Type':'Limited-Memory BFGS', #BFGS-based Hessian-update in␣
↪trust-region model
                     'Maximum Storage':10
                    }
          }
```

6.5 Solving a constrained problem with an augmented Lagrangian method

To solve a problem with equality or inequality constraint using an augmented Lagrangian method, we can set `Step` and `General` as follows (the provided values are the default ones). Note that the augmented Lagrangian algorithm solves a sequence of intermediate models. These intermediate models are unconstrained optimization problems that encode constraints via penalization. To solve these unconstrained optimization problems, we use again a trust-region method based on BFGS-updates of the Hessian. The augmented Lagrangian source code is here.

```
'Step':{'Type':'Augmented Lagrangian',
        'Augmented Lagrangian':{'Use Default Initial Penalty Parameter':true,
                                'Initial Penalty Parameter':1.e1,
                                # Multiplier update parameters
                                'Use Scaled Augmented Lagrangian':false,
                                'Penalty Parameter Reciprocal Lower Bound':0.1,
                                'Penalty Parameter Growth Factor':1.e1,
                                'Maximum Penalty Parameter':1.e8,
                                # Optimality tolerance update
                                'Optimality Tolerance Update Exponent':1,
                                'Optimality Tolerance Decrease Exponent':1,
                                'Initial Optimality Tolerance':1,
                                # Feasibility tolerance update
                                'Feasibility Tolerance Update Exponent':0.1,
                                'easibility Tolerance Decrease Exponent':0.9,
                                'Initial Feasibility Tolerance':1,
                                # Subproblem information
                                'Print Intermediate Optimization History':false,
                                'Subproblem Iteration Limit':1000,
                                'Subproblem Step Type':'Trust Region',
                                # Scaling
                                'Use Default Problem Scaling':true,
                                'Objective Scaling':1.0,
                                'Constraint Scaling':1.0,
                               }
        'Trust Region':{'Initial Radius':-1, #determine initial radius with heuristics
                        'Maximum Radius':1.e8,
                        'Subproblem Solver':'Dogleg',
                        'Radius Growing Rate':2.5
                        'Step Acceptance Threshold':0.05,
                        'Radius Shrinking Threshold':0.05,
                        'Radius Growing Threshold':0.9,
                        'Radius Shrinking Rate (Negative rho)':0.0625,
                        'Radius Shrinking Rate (Positive rho)':0.25,
                        'Radius Growing Rate':2.5,
                        'Sufficient Decrease Parameter':1.e-4,
                        'Safeguard Size':100.0,
                       }
'General':{'Print Verbosity':0, #set to any number >0 for increased verbosity
           'Secant':{'Type':'Limited-Memory BFGS', #BFGS-based Hessian-update in␣
↪trust-region model
                     'Maximum Storage':10
                    }
          }
```

Note: The augmented Lagrangian default constraint, gradient, and step tolerances for the outer iterations are set to `1.e-8`. The user can set different tolerances by specifying them in `Status Test`. The gradient and step tolerances for the internal iterations are set by the augmented Lagrangian algorithm itself (based on a number of parameters, including the gradient tolerance for the outer iteration, see lines 374-375 in the source code) and cannot be modified by the user.