TopOpt Documentation

Release 0.0.1-alpha.1

Zachary Ferguson

Feb 11, 2020

General

- **Free Software**: MIT License
- **Github Repository**: https://github.com/zfergus/topopt

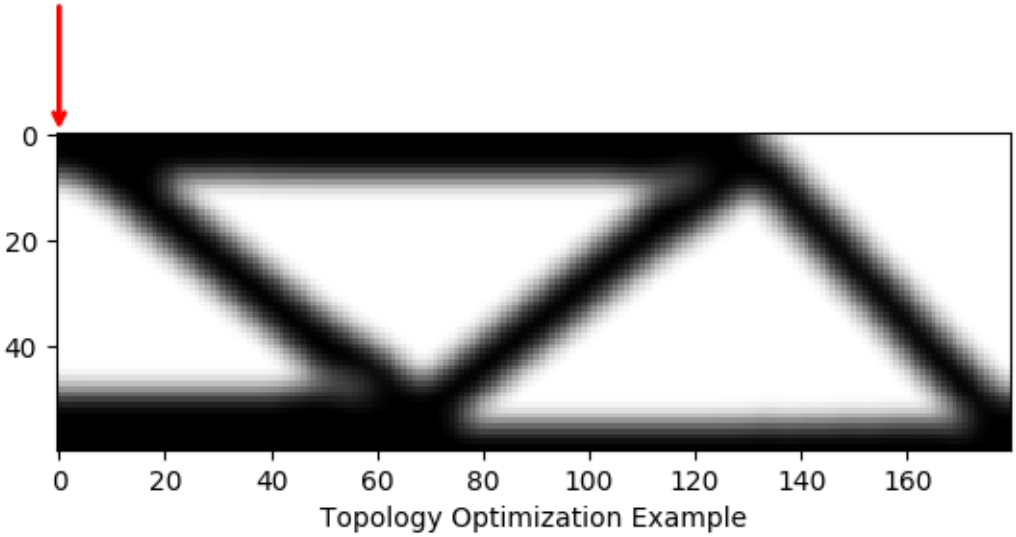
> **Warning:** This library is in early stages of development and consequently the API may change to better improve usability.

Topology optimization is a form of structure optimization where the design variable is the topology of the structure. Topological changes are achieved by optimizing the material distribution within a given design space.

TopOpt is a python library for topology optimization. TopOpt contains common design problems (e.g. minimum compliance) solved using advanced methods (e.g. Method of Moving Asymptotes (MMA)). Using TopOpt we can optimize the classic Messerschmitt–Bölkow–Blohm (MBB) beam in a few lines of code:

```
import numpy
from topopt.boundary_conditions import MBBBeamBoundaryConditions
from topopt.problems import ComplianceProblem
from topopt.solvers import TopOptSolver
from topopt.filters import DensityBasedFilter
from topopt.guis import GUI

nelx, nely = 180, 60  # Number of elements in the x and y
volfrac = 0.4  # Volume fraction for constraints
penal = 3.0  # Penalty for SIMP
rmin = 5.4  # Filter radius

# Initial solution
x = volfrac * numpy.ones(nely * nelx, dtype=float)

# Boundary conditions defining the loads and fixed points
bc = MBBBeamBoundaryConditions(nelx, nely)

# Problem to optimize given objective and constraints
problem = ComplianceProblem(bc, penal)
gui = GUI(problem, "Topology Optimization Example")
topopt_filter = DensityBasedFilter(nelx, nely, rmin)
solver = TopOptSolver(problem, volfrac, topopt_filter, gui)
x_opt = solver.optimize(x)

input("Press enter...")
```

Output:


0
20
40
0
20
40
60
80
100
120
140
160
Topology Optimization Example

CHAPTER 1

Installation

To install TopOpt, run this command in your terminal:

```
pip install topopt
```

This is the preferred method to install TopOpt, as it will always install the most recent stable release.

In case you want to install the bleeding-edge version, clone this repo:

```
git clone https://github.com/zfergus/topopt.git
```

and then run

```
cd topopt
python setup.py install
```

CHAPTER 2

Concepts

CHAPTER 3

Examples

3.1 Parameters

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Explore affects of parameters using complicance problem and MBB Beam."""
from __future__ import division

import context  # noqa

from topopt import cli


def main():
    """Explore affects of parameters using complicance problem and MBB Beam."""
    # Default input parameters
    nelx, nely, volfrac, penalty, rmin, ft = cli.parse_args()
    cli.main(nelx, nely, volfrac, penalty, rmin, ft)
    # Vary the filter radius
    for scaled_factor in [0.25, 2]:
        cli.main(nelx, nely, volfrac, penalty, scaled_factor * rmin, ft)
    # Vary the penalization power
    for scaled_factor in [0.5, 4]:
        cli.main(nelx, nely, volfrac, scaled_factor * penalty, rmin, ft)
    # Vary the discreization
    for scale_factor in [0.5, 2]:
        cli.main(int(scale_factor * nelx), int(scale_factor * nely),
                 volfrac, penalty, rmin, ft)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    main()
```

3.2 Simultaneous Point Loads

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Two simultaneous point loads."""
from __future__ import division

import numpy

import context  # noqa

from topopt.boundary_conditions import BoundaryConditions
from topopt.utils import xy_to_id

from topopt import cli


class SimultaneousLoadsBoundaryConditions(BoundaryConditions):
    """Two simultaneous point loads along the top boundary."""

    @property
    def fixed_nodes(self):
        """Return a list of fixed nodes for the problem."""
        bottom_left = 2 * xy_to_id(0, self.nely, self.nelx, self.nely)
        bottom_right = 2 * xy_to_id(self.nelx, self.nely, self.nelx, self.
↪nely)
        fixed = numpy.array(
            [bottom_left, bottom_left + 1, bottom_right, bottom_right + 1])
        return fixed

    @property
    def forces(self):
        """Return the force vector for the problem."""
        f = numpy.zeros((2 * (self.nelx + 1) * (self.nely + 1), 1))
        id1 = 2 * xy_to_id(7 * self.nelx // 20, 0, self.nelx, self.nely) + 1
        id2 = 2 * xy_to_id(13 * self.nelx // 20, 0, self.nelx, self.nely) + 1
        f[[id1, id2], 0] = -1
        return f


def main():
    """Two simultaneous point loads."""
    # Default input parameters
    nelx, nely, volfrac, penalty, rmin, ft = cli.parse_args(
        nelx=120, volfrac=0.2, rmin=1.5)
    cli.main(nelx, nely, volfrac, penalty, rmin, ft,
```

(continues on next page)

(continued from previous page)

```
            bc=SimultaneousLoadsBoundaryConditions(nelx, nely))


if __name__ == "__main__":
    main()
```

3.3 Distributed Load

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Distributed load."""
from __future__ import division

import numpy

import context  # noqa

from topopt.boundary_conditions import BoundaryConditions
from topopt.utils import xy_to_id

from topopt import cli


class DistributedLoadBoundaryConditions(BoundaryConditions):
    """Distributed load along the top boundary."""

    @property
    def fixed_nodes(self):
        """Return a list of fixed nodes for the problem."""
        bottom_left = 2 * xy_to_id(0, self.nely, self.nelx, self.nely)
        bottom_right = 2 * xy_to_id(
            self.nelx, self.nely, self.nelx, self.nely)
        fixed = numpy.array([bottom_left, bottom_left + 1,
                             bottom_right, bottom_right + 1])
        return fixed

    @property
    def forces(self):
        """Return the force vector for the problem."""
        topx_to_id = numpy.vectorize(
            lambda x: xy_to_id(x, 0, self.nelx, self.nely))
        topx = 2 * topx_to_id(numpy.arange(self.nelx + 1)) + 1
        f = numpy.zeros((2 * (self.nelx + 1) * (self.nely + 1), 1))
        f[topx, 0] = -1
        return f

    @property
    def nonuniform_forces(self):
        """Return the force vector for the problem."""
```

(continues on next page)

(continued from previous page)

```
        topx_to_id = numpy.vectorize(
            lambda x: xy_to_id(x, 0, self.nelx, self.nely))
        topx = 2 * topx_to_id(numpy.arange(self.nelx + 1)) + 1
        f = numpy.zeros((2 * (self.nelx + 1) * (self.nely + 1), 1))
        f[topx, 0] = (0.5 * numpy.cos(
            numpy.linspace(0, 2 * numpy.pi, topx.shape[0])) - 0.5)
        return f


def main():
    """Distributed load."""
    # Default input parameters
    nelx, nely, volfrac, penalty, rmin, ft = cli.parse_args(
        nelx=120, volfrac=0.2, rmin=1.2)
    cli.main(nelx, nely, volfrac, penalty, rmin, ft,
             bc=DistributedLoadBoundaryConditions(nelx, nely))


if __name__ == "__main__":
    main()
```

3.4 Multiple Loads

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Multiple loads."""
from __future__ import division

import numpy

import context  # noqa

from topopt.boundary_conditions import BoundaryConditions
from topopt.problems import ComplianceProblem
from topopt.utils import xy_to_id

from topopt import cli


class MultipleLoadsBoundaryConditions(BoundaryConditions):
    """Multiple loads applied to the top boundary."""

    @property
    def fixed_nodes(self):
        """Return a list of fixed nodes for the problem."""
        bottom_left = 2 * xy_to_id(0, self.nely, self.nelx, self.nely)
        bottom_right = 2 * xy_to_id(self.nelx, self.nely, self.nelx, self.
↪nely)
        fixed = numpy.array(
```

(continues on next page)

(continued from previous page)

```
            [bottom_left, bottom_left + 1, bottom_right, bottom_right + 1])
        return fixed

    @property
    def forces(self):
        """Return the force vector for the problem."""
        f = numpy.zeros((2 * (self.nelx + 1) * (self.nely + 1), 2))
        id1 = 2 * xy_to_id(7 * self.nelx // 20, 0, self.nelx, self.nely) + 1
        id2 = 2 * xy_to_id(13 * self.nelx // 20, 0, self.nelx, self.nely) + 1
        f[id1, 0] = -1
        f[id2, 1] = -1
        return f


def main():
    """Multiple loads."""
    # Default input parameters
    nelx, nely, volfrac, penalty, rmin, ft = cli.parse_args(
        nelx=120, volfrac=0.2, rmin=1.5)
    bc = MultipleLoadsBoundaryConditions(nelx, nely)
    problem = ComplianceProblem(bc, penalty)
    cli.main(nelx, nely, volfrac, penalty, rmin, ft, bc=bc,
             problem=problem)


if __name__ == "__main__":
    main()
```

3.5 Computing Stress

3.5.1 Distributed Load

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Multiple simultaneous point loads with stress computation."""
from __future__ import division

import context  # noqa

from topopt.guis import StressGUI
from topopt.problems import VonMisesStressProblem

from topopt import cli

from distributed_load import DistributedLoadBoundaryConditions


def main():
```

(continues on next page)

(continued from previous page)

```
    """Multiple simultaneous point loads with stress computation."""
    # Default input parameters
    nelx, nely, volfrac, penalty, rmin, ft = cli.parse_args(
        nelx=120, volfrac=0.2, rmin=1.2)
    bc = DistributedLoadBoundaryConditions(nelx, nely)
    problem = VonMisesStressProblem(nelx, nely, penalty, bc)
    gui = StressGUI(problem, title="Stresses of Distributed Load Example")
    cli.main(nelx, nely, volfrac, penalty, rmin, ft, bc=bc,
             problem=problem, gui=gui)


if __name__ == "__main__":
    main()
```

3.5.2 Multiple Load

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Multiple loads with stresses."""
from __future__ import division

import context  # noqa

from topopt.problems import VonMisesStressProblem
from topopt.guis import StressGUI

from topopt import cli

from multiple_loads_mma import MultipleLoadsBoundaryConditions


def main():
    """Multiple loads with stresses."""
    # Default input parameters
    nelx, nely, volfrac, penalty, rmin, ft = cli.parse_args(
        nelx=120, volfrac=0.2, rmin=1.5)
    bc = MultipleLoadsBoundaryConditions(nelx, nely)
    problem = VonMisesStressProblem(nelx, nely, penalty, bc)
    title = cli.title_str(nelx, nely, volfrac, rmin, penalty)
    gui = StressGUI(problem, title)
    cli.main(nelx, nely, volfrac, penalty, rmin, ft, bc=bc,
             problem=problem, gui=gui)


if __name__ == "__main__":
    main()
```

CHAPTER 4

Changelog

CHAPTER 5

Problems

Topology optimization problem to solve.

5.1 Base Problem

class topopt.problems.**Problem**(*bc*, *penalty*)

Abstract topology optimization problem.

bc

The boundary conditions for the problem.

Type *BoundaryConditions*

penalty

The SIMP penalty value.

Type float

f

The right-hand side of the FEM equation (forces).

Type numpy.ndarray

u

The variables of the FEM equation.

Type numpy.ndarray

obje

The per element objective values.

Type numpy.ndarray

__init__(*bc*, *penalty*)
Create the topology optimization problem.

Parameters

- **bc** (*BoundaryConditions*) – The boundary conditions of the problem.
- **penalty** (float) – The penalty value used to penalize fractional densities in SIMP.

compute_objective(*xPhys*, *dobj*)
Compute objective and its gradient.

Parameters

- **xPhys** (ndarray) – The design variables.
- **dobj** (ndarray) – The gradient of the objective to compute.

Returns The objective value.

Return type float

penalize_densities(*x*, *drho=None*)
Compute the penalized densties (and optionally its derivative).

Parameters

- **x** (ndarray) – The density variables to penalize.
- **drho** (Optional[ndarray]) – The derivative of the penealized densities to compute. Only set if drho is not None.

Returns The penalized densities used for SIMP.

Return type numpy.ndarray

5.2 Compliance Problem

class topopt.problems.**ComplianceProblem**(*bc*, *penalty*)
Topology optimization problem to minimize compliance.

$$\begin{aligned} \min_{\boldsymbol{\rho}} \quad & \mathbf{f}^T\mathbf{u} \\ \text{subject to :} \quad & \mathbf{K}\mathbf{u} = \mathbf{f} \\ & \sum_{e=1}^{N} v_e\rho_e \leq V_{\text{frac}}, \quad 0 < \rho_{\min \leq \rho_e \leq 1} \end{aligned}$$

where $\mathbf{f}$ are the forces, $\mathbf{u}$ are the displacements, $\mathbf{K}$ is the striffness matrix, and V is the volume.

compute_objective(*xPhys*, *dobj*)
Compute compliance and its gradient.

The objective is $\mathbf{f}^T\mathbf{u}$. The gradient of the objective is

where $\boldsymbol{\lambda} = \mathbf{u}$.

Parameters

- **xPhys** (ndarray) – The element densities.
- **dobj** (ndarray) – The gradient of compliance.

Returns The compliance value.

Return type float

5.3 Time-Harmonic Loads Problem

class topopt.problems.**HarmonicLoadsProblem**(*bc*, *penalty*)

Topology optimization problem to minimize dynamic compliance.

Replaces standard forces with undamped forced vibrations.

$$\begin{aligned} \min_{\boldsymbol{\rho}} \quad & \mathbf{f}^T\mathbf{u} \\ \text{subject to :} \quad & \mathbf{S}\mathbf{u} = \mathbf{f} \\ & \sum_{e=1}^{N} v_e\rho_e \leq V_{\text{frac}}, \quad 0 < \rho_{\min} \leq \rho_e \leq 1 \end{aligned}$$

where $\mathbf{f}$ is the amplitude of the load, $\mathbf{u}$ is the amplitude of vibration, and $\mathbf{S}$ is the system matrix (or "dynamic striffness" matrix) defined as

$$\mathbf{S} = \mathbf{K} - \omega^2\mathbf{M}$$

where ω is the angular frequency of the load, and $\mathbf{M}$ is the global mass matrix.

__init__(*bc*, *penalty*)

Create the topology optimization problem.

Parameters

- **bc** (*BoundaryConditions*) – The boundary conditions of the problem.
- **penalty** (float) – The penalty value used to penalize fractional densities in SIMP.

build_M(*xPhys*, *remove_constrained=True*)

Build the stiffness matrix for the problem.

Parameters

- **xPhys** (ndarray) – The element densisities used to build the stiffness matrix.
- **remove_constrained** (bool) – Should the constrained nodes be removed?

Returns The stiffness matrix for the mesh.

Return type scipy.sparse.coo_matrix

build_indices()
Build the index vectors for the finite element coo matrix format.

Return type None

compute_displacements(*xPhys*)
Compute the amplitude of vibration given the densities.

Compute the amplitude of vibration, $\mathbf{u}$, using linear elastic finite element analysis (solving $\mathbf{Su} = \mathbf{f}$ where $\mathbf{S} = \mathbf{K} - \omega^2\mathbf{M}$ is the system matrix and $\mathbf{f}$ is the force vector).

Parameters **xPhys** (ndarray) – The element densisities used to build the stiffness matrix.

Returns The displacements solve using linear elastic finite element analysis.

Return type numpy.ndarray

compute_objective(*xPhys*, *dobj*)
Compute compliance and its gradient.

The objective is $\mathbf{f}^T\mathbf{u}$. The gradient of the objective is

where $\boldsymbol{\lambda} = \mathbf{u}$.

Parameters

- **xPhys** (ndarray) – The element densities.
- **dobj** (ndarray) – The gradient of compliance.

Returns The compliance value.

Return type float

static lm(*nel*)
Build the element mass matrix.

$$M = \frac{1}{9\times 4n}\begin{bmatrix} 4 & 0 & 2 & 0 & 1 & 0 & 2 & 0 \\ 0 & 4 & 0 & 2 & 0 & 1 & 0 & 2 \\ 2 & 0 & 4 & 0 & 2 & 0 & 1 & 0 \\ 0 & 2 & 0 & 4 & 0 & 2 & 0 & 1 \\ 1 & 0 & 2 & 0 & 4 & 0 & 2 & 0 \\ 0 & 1 & 0 & 2 & 0 & 4 & 0 & 2 \\ 2 & 0 & 1 & 0 & 2 & 0 & 4 & 0 \\ 0 & 2 & 0 & 1 & 0 & 2 & 0 & 4 \end{bmatrix}$$

Where n is the total number of elements. The total mass is equal to unity.

Parameters **nel** (int) – The total number of elements.

Returns The element mass matrix for the material.

Return type numpy.ndarray

5.4 von Mises Stress Problem

class topopt.problems.**VonMisesStressProblem**(*nelx*, *nely*, *penalty*, *bc*, *side=1*)

Topology optimization problem to minimize stress.

Todo:

- Currently this problem minimizes compliance and computes stress on the side. This needs to be replaced to match the promise of minimizing stress.

---

static B(*side*)

Construct a strain-displacement matrix for a 2D regular grid.

$$B = \frac{1}{2s}\begin{bmatrix} 1 & 0 & -1 & 0 & -1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 & 0 & -1 & 0 & -1 \\ 1 & 1 & 1 & -1 & -1 & -1 & -1 & 1 \end{bmatrix}$$

where s is the side length of the square elements.

Todo:

- Check that this is not -B

Parameters **side** (float) – The side length of the square elements.

Returns The strain-displacement matrix for a 2D regular grid.

Return type numpy.ndarray

static E(*nu*)

Construct a constitutive matrix for a 2D regular grid.

$$E = \frac{1}{1-\nu^2}\begin{bmatrix} 1 & \nu & 0 \\ \nu & 1 & 0 \\ 0 & 0 & \frac{1-\nu}{2} \end{bmatrix}$$

Parameters **nu** – The Poisson's ratio of the material.

Returns The constitutive matrix for a 2D regular grid.

Return type numpy.ndarray

__init__(*nelx*, *nely*, *penalty*, *bc*, *side=1*)

Create the topology optimization problem.

Parameters

- **bc** – The boundary conditions of the problem.
- **penalty** – The penalty value used to penalize fractional densities in SIMP.

build_dK(*xPhys*, *remove_constrained=True*)

build_dK0(*drho_xi*, *i*, *remove_constrained=True*)

compute_objective(*xPhys*, *dobj*)
Compute compliance and its gradient.

compute_stress_objective(*xPhys*, *dobj*, *p=4*)
Compute stress objective and its gradient.

static dsigma_pow(*s11*, *s22*, *s12*, *ds11*, *ds22*, *ds12*, *p*)
Compute the gradient of the stress to the p^{th} power.

$\nabla\sigma^p = \frac{p\sigma^{p-1}}{2\sigma}\nabla(\sigma^2)$

Todo:

- Properly document what the sigma variables represent.
- Rename the sigma variables to something more readable.

Parameters

- **s11** (ndarray) – σ_{11}
- **s22** (ndarray) – σ_{22}
- **s12** (ndarray) – σ_{12}
- **ds11** (ndarray) – $\nabla\sigma_{11}$
- **ds22** (ndarray) – $\nabla\sigma_{22}$
- **ds12** (ndarray) – $\nabla\sigma_{12}$
- **p** (float) – The power (p) to raise the von Mises stress.

Returns The gradient of the von Mises stress to the p^{th} power.

Return type numpy.ndarray

static sigma_pow(*s11*, *s22*, *s12*, *p*)
Compute the von Mises stress raised to the p^{th} power.

$\sigma^p = \left(\sqrt{\sigma_{11}^2 - \sigma_{11}\sigma_{22} + \sigma_{22}^2 + 3\sigma_{12}^2}\right)^p$

Todo:

- Properly document what the sigma variables represent.
- Rename the sigma variables to something more readable.

Parameters

- **s11** (ndarray) – σ_{11}

- **s22** (ndarray) – σ_{22}
- **s12** (ndarray) – σ_{12}
- **p** (float) – The power (p) to raise the von Mises stress.

Returns The von Mises stress to the p^{th} power.

Return type numpy.ndarray

test_calculate_objective(*xPhys*, *dobj*, *p=4*, *dx=1e-06*)

Calculate the gradient of the stresses using finite differences.

Parameters

- **xPhys** (ndarray) – The element densities.
- **dobj** (ndarray) – The gradient of the stresses to compute.
- **p** (float) – The exponent for computing the softmax of the stresses.
- **dx** (float) – The difference in x values used for finite differences.

Returns The analytic objective value.

Return type float

CHAPTER 6

Solvers

Solvers to solve topology optimization problems.

Todo:

- Make TopOptSolver an abstract class
- Rename the current TopOptSolver to MMASolver(TopOptSolver)
- Create a TopOptSolver using originality criterion

6.1 Base Solver

class topopt.solvers.**TopOptSolver**(*problem*, *volfrac*, *filter*, *gui*, *maxeval=2000*, *ftol_rel=0.001*)

Solver for topology optimization problems using NLopt's MMA solver.

__init__(*problem*, *volfrac*, *filter*, *gui*, *maxeval=2000*, *ftol_rel=0.001*)

Create a solver to solve the problem.

Parameters

- **problem** (*topopt.problems.Problem*) – The topology optimization problem to solve.
- **volfrac** (*float*) – The maximum fraction of the volume to use.
- **filter** (*topopt.filters.Filter*) – A filter for the solutions to reduce artefacts.

- **gui** (*topopt.guis.GUI*) – The graphical user interface to visualize intermediate results.
- **maxeval** (*int*) – The maximum number of evaluations to perform.
- **ftol** (*float*) – A floating point tolerance for relative change.

filter_variables(*x*)
Filter the variables and impose values on passive/active variables.

Parameters **x** (ndarray) – The variables to be filtered.

Returns The filtered "physical" variables.

Return type numpy.ndarray

ftol_rel
Relative tolerance for convergence.

Type float

maxeval
Maximum number of objective evaluations (iterations).

Type int

objective_function(*x*, *dobj*)
Compute the objective value and gradient.

Parameters

- **x** (ndarray) – The design variables for which to compute the objective.
- **dobj** (ndarray) – The gradient of the objective to compute.

Returns The objective value.

Return type float

objective_function_fdiff(*x*, *dobj*, *epsilon=1e-06*)
Compute the objective value and gradient using finite differences.

Parameters

- **x** (ndarray) – The design variables for which to compute the objective.
- **dobj** (ndarray) – The gradient of the objective to compute.
- **epsilon** – Change in the finite difference to compute the gradient.

Returns The objective value.

Return type float

optimize(*x*)
Optimize the problem.

Parameters **x** (ndarray) – The initial value for the design variables.

Returns The optimal value of x found.

Return type numpy.ndarray

volume_function(*x*, *dv*)

Compute the volume constraint value and gradient.

Parameters

- **x** (ndarray) – The design variables for which to compute the volume constraint.
- **dobj** – The gradient of the volume constraint to compute.

Returns The volume constraint value.

Return type float

6.2 Specialized Solvers

CHAPTER 7

Boundary Conditions

Boundary conditions for topology optimization (forces and fixed nodes).

7.1 Base Boundary Conditions

class topopt.boundary_conditions.**BoundaryConditions**(*nelx*, *nely*)

Abstract class for boundary conditions to a topology optimization problem.

Functionalty for geting fixed nodes, forces, and passive elements.

nelx

The number of elements in the x direction.

Type int

nely

The number of elements in the y direction.

Type int

__init__(*nelx*, *nely*)

Create the boundary conditions with the size of the grid.

Parameters

- **nelx** (int) – The number of elements in the x direction.
- **nely** (int) – The number of elements in the y direction.

active_elements

Active elements to be set to full density.

Type numpy.ndarray

fixed_nodes
Fixed nodes of the problem.

Type numpy.ndarray

forces
Force vector for the problem.

Type numpy.ndarray

passive_elements
Passive elements to be set to zero density.

Type numpy.ndarray

7.2 All Boundary Conditions

This section contains all boundary conditions currently implemented in the library. It is not yet comprehensive, so it would be really nice if you can help me add more!

7.2.1 MBB Beam

class topopt.boundary_conditions.**MBBBeamBoundaryConditions**(*nelx*, *nely*)
Boundary conditions for the Messerschmitt–Bölkow–Blohm (MBB) beam.

fixed_nodes
Fixed nodes in the bottom corners.

Type numpy.ndarray

forces
Force vector in the top center.

Type numpy.ndarray

7.2.2 Cantilever

class topopt.boundary_conditions.**CantileverBoundaryConditions**(*nelx*, *nely*)
Boundary conditions for a cantilever.

fixed_nodes
Fixed nodes on the left.

Type numpy.ndarray

forces
Force vector in the middle right.

Type numpy.ndarray

7.2.3 L-Bracket

The L-bracket is a bracket in the shape of a capital "L". This domain is achieved using a passive block in the upper right corner of a square domain.

The passive block is defined by its minimum x coordinate and the maximum y coordinate.

class topopt.boundary_conditions.**LBracketBoundaryConditions**(*nelx*, *nely*, *minx*, *maxy*)

Boundary conditions for a L-shaped bracket.

__init__(*nelx*, *nely*, *minx*, *maxy*)

Create L-bracket boundary conditions with the size of the grid.

Parameters

- **nelx** (int) – The number of elements in the x direction.
- **nely** (int) – The number of elements in the y direction.
- **minx** (int) – The minimum x coordinate of the passive upper-right block.
- **maxy** (int) – The maximum y coordinate of the passive upper-right block.

Raises ValueError: *minx* and *maxy* must be indices in the grid.

fixed_nodes

Fixed nodes in the top row.

Type numpy.ndarray

forces

Force vector in the middle right.

Type numpy.ndarray

passive_elements

Passive elements in the upper right corner.

Type numpy.ndarray

7.2.4 I-Beam

The I-beam is a cross-section of a beam in the shape of a capital "I". This domain is achieved using two passive blocks in the middle left and right of a square domain.

class topopt.boundary_conditions.**IBeamBoundaryConditions**(*nelx*, *nely*)

Boundary conditions for an I-shaped beam.

fixed_nodes
Fixed nodes in the bottom row.

Type numpy.ndarray

forces
Force vector on the top row.

Type numpy.ndarray

passive_elements
Passive elements on the left and right.

Type numpy.ndarray

7.2.5 II-Beam

The II-beam is a cross-section of a beam in the shape of "II". This domain is achieved using three passive blocks in the middle left, center, and right of a square domain.

class topopt.boundary_conditions.**IIBeamBoundaryConditions**(*nelx*, *nely*)
Boundary conditions for an II-shaped beam.

passive_elements
Passives on the left, middle, and right.

Type numpy.ndarray

CHAPTER 8

Filters

Filter the solution to topology optimization.

8.1 Base Filter

class topopt.filters.**Filter**(*nelx*, *nely*, *rmin*)

Filter solutions to topology optimization to avoid checker boarding.

__init__(*nelx*, *nely*, *rmin*)

Create a filter to filter solutions.

Build (and assemble) the index+data vectors for the coo matrix format.

Parameters

- **nelx** (int) – The number of elements in the x direction.
- **nely** (int) – The number of elements in the y direction.
- **rmin** (float) – The filter radius.

filter_objective_sensitivities(*xPhys*, *dobj*)

Filter derivative of the objective.

Parameters

- **xPhys** (ndarray) – The filtered density values.
- **dobj** (ndarray) – The filtered objective sensitivities to be computed.

Return type None

filter_variables(*x*, *xPhys*)

Filter the variable of the solution to produce xPhys.

Parameters

- **x** (ndarray) – The raw density values.
- **xPhys** (ndarray) – The filtered density values to be computed

Return type None

filter_volume_sensitivities(*xPhys*, *dv*)
Filter derivative of the volume.

Parameters

- **xPhys** (ndarray) – The filtered density values.
- **dv** (ndarray) – The filtered volume sensitivities to be computed.

Return type None

8.2 Density Based Filter

class topopt.filters.**DensityBasedFilter**(*nelx*, *nely*, *rmin*)
Density based filter of solutions.

filter_objective_sensitivities(*xPhys*, *dobj*)
Filter derivative of the objective.

Parameters

- **xPhys** (ndarray) – The filtered density values.
- **dobj** (ndarray) – The filtered objective sensitivities to be computed.

Return type None

filter_variables(*x*, *xPhys*)
Filter the variable of the solution to produce xPhys.

Parameters

- **x** (ndarray) – The raw density values.
- **xPhys** (ndarray) – The filtered density values to be computed

Return type None

filter_volume_sensitivities(*xPhys*, *dv*)
Filter derivative of the volume.

Parameters

- **xPhys** (ndarray) – The filtered density values.
- **dv** (ndarray) – The filtered volume sensitivities to be computed.

Return type None

8.3 Sensitivity Based Filter

class topopt.filters.**SensitivityBasedFilter**(*nelx*, *nely*, *rmin*)

Sensitivity based filter of solutions.

filter_objective_sensitivities(*xPhys*, *dobj*)

Filter derivative of the objective.

Parameters

- **xPhys** (ndarray) – The filtered density values.
- **dobj** (ndarray) – The filtered objective sensitivities to be computed.

Return type None

filter_variables(*x*, *xPhys*)

Filter the variable of the solution to produce xPhys.

Parameters

- **x** (ndarray) – The raw density values.
- **xPhys** (ndarray) – The filtered density values to be computed

Return type None

filter_volume_sensitivities(*xPhys*, *dv*)

Filter derivative of the volume.

Parameters

- **xPhys** (ndarray) – The filtered density values.
- **dv** (ndarray) – The filtered volume sensitivities to be computed.

Return type None

CHAPTER 9

Compliant Mechanism Synthesis

The sub-package topopt.mechanisms provides topology optimization routines designed for the synthesis of compliant mechanisms.

Mechanisms: A Topology Optimization Libarary for Compliant Mechanisms Synthesis.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
"""Create a compliance mechanism that inverts displacments."""

import context  # noqa

from topopt.mechanisms.boundary_conditions import (
    DisplacementInverterBoundaryConditions,
    CrossSensitivityExampleBoundaryConditions,
    GripperBoundaryConditions)
from topopt.mechanisms.problems import MechanismSynthesisProblem
from topopt.mechanisms.solvers import MechanismSynthesisSolver
from topopt.filters import SensitivityBasedFilter, DensityBasedFilter
from topopt.guis import GUI

from topopt import cli


def main():
    """Run the example by constructing the TopOpt objects."""
    # Default input parameters
    nelx, nely, volfrac, penalty, rmin, ft = cli.parse_args(
        nelx=100, nely=100, volfrac=0.3, penalty=10, rmin=1.4)
    bc = DisplacementInverterBoundaryConditions(nelx, nely)
    # bc = GripperBoundaryConditions(nelx, nely)
    # bc = CrossSensitivityExampleBoundaryConditions(nelx, nely)
```

(continues on next page)

(continued from previous page)

```
    problem = MechanismSynthesisProblem(bc, penalty)
    title = cli.title_str(nelx, nely, volfrac, rmin, penalty)
    gui = GUI(problem, title)
    filter = [SensitivityBasedFilter, DensityBasedFilter][ft](nelx, nely,␣
↪rmin)
    solver = MechanismSynthesisSolver(problem, volfrac, filter, gui)
    cli.main(nelx, nely, volfrac, penalty, rmin, ft, solver=solver)


if __name__ == "__main__":
    main()
```

9.1 Problems

Compliant mechanism synthesis problems using topology optimization.

9.1.1 Base Problem

class topopt.mechanisms.problems.**MechanismSynthesisProblem**(*bc*, *penalty*)

Topology optimization problem to generate compliant mechanisms.

$$\max_{\boldsymbol{\rho}} \quad \{u_{\text{out}} = \mathbf{l}^T \mathbf{u}\}$$

$$\text{subject to :} \quad \mathbf{K}\mathbf{u} = \mathbf{f}_{\text{in}}$$

$$\sum_{e=1}^{N} v_e \rho_e \leq V_{\text{frac}}, \quad 0 < \rho_{\min} \leq \rho_e \leq 1, \quad e=1,\ldots,N.$$

where $\mathbf{l}$ is a vector with the value 1 at the degree(s) of freedom corresponding to the output point and with zeros at all other places.

spring_stiffnesses

The spring stiffnesses of the actuator and output displacement.

Type numpy.ndarray

Emin

The minimum stiffness of elements.

Type float

Emax

The maximum stiffness of elements.

Type float

__init__(*bc*, *penalty*)

Create the topology optimization problem.

Parameters

- **nelx** – Number of elements in the x direction.
- **nely** – Number of elements in the x direction.
- **penalty** (float) – Penalty value used to penalize fractional densities in SIMP.
- **bc** (*MechanismSynthesisBoundaryConditions*) – Boundary conditions of the problem.

build_K(*xPhys*, *remove_constrained=True*)

Build the stiffness matrix for the problem.

Parameters

- **xPhys** (ndarray) – The element densisities used to build the stiffness matrix.
- **remove_constrained** (bool) – Should the constrained nodes be removed?

Returns

Return type The stiffness matrix for the mesh.

compute_objective(*xPhys*, *dobj*)

Compute the objective and gradient of the mechanism synthesis problem.

The objective is $u_{\text{out}} = \mathrm{l}^T \mathbf{u}$ where l is a vector with the value 1 at the degree(s) of freedom corresponding to the output point and with zeros at all other places. The gradient of the objective is

where $\mathbf{K}\boldsymbol{\lambda} = -\mathrm{l}$.

Parameters

- **xPhys** (ndarray) – The density design variables.
- **dobj** (ndarray) – The gradient of the objective to compute.

Returns

Return type The objective of the compliant mechanism synthesis problem.

static lk(*E=1.0*, *nu=0.3*)

Build the element stiffness matrix.

Parameters

- **E** (float) – Young's modulus of the material.
- **nu** (float) – Poisson's ratio of the material.

Returns

Return type The element stiffness matrix for the material.

9.2 Solvers

Solve compliant mechanism synthesis problems using topology optimization.

9.2.1 Base Solver

class topopt.mechanisms.solvers.**MechanismSynthesisSolver**(*problem*, *volfrac*, *filter*, *gui*, *maxeval=2000*, *ftol=0.0001*)

Specialized solver for mechanism synthesis problems.

This solver is specially designed to create compliant mechanisms.

__init__(*problem*, *volfrac*, *filter*, *gui*, *maxeval=2000*, *ftol=0.0001*)

Create a mechanism synthesis solver to solve the problem.

Parameters

- **problem** (*topopt.problems.Problem*) – The topology optimization problem to solve.
- **volfrac** (*float*) – The maximum fraction of the volume to use.
- **filter** (*topopt.filters.Filter*) – A filter for the solutions to reduce artefacts.
- **gui** (*topopt.guis.GUI*) – The graphical user interface to visualize intermediate results.
- **maxeval** (*int*) – The maximum number of evaluations to perform.
- **ftol** (*float*) – A floating point tolerance for relative change.

objective_function(*x*, *dobj*)

Compute the objective value and gradient.

Parameters

- **x** (*numpy.ndarray*) – The design variables for which to compute the objective.
- **dobj** (*numpy.ndarray*) – The gradient of the objective to compute.

Returns The objective value.

Return type float

volume_function(*x*, *dv*)

Compute the volume constraint value and gradient.

Parameters

- **x** (*numpy.ndarray*) – The design variables for which to compute the volume constraint.

- **dobj** (*numpy.ndarray*) – The gradient of the volume constraint to compute.

Returns The volume constraint value.

Return type float

9.2.2 Specialized Solvers

9.3 Boundary Conditions

Boundary conditions for mechanism synthesis (forces and fixed nodes).

9.3.1 Mechanism Synthesis Boundary Conditions

class topopt.mechanisms.boundary_conditions.**MechanismSynthesisBoundaryConditions**(*ne*
ne

Boundary conditions for compliant mechanism synthesis.

output_displacement_mask
Mask of the output displacement.

Type numpy.ndarray

Displacement Inverter

class topopt.mechanisms.boundary_conditions.**DisplacementInverterBoundaryConditions**

Boundary conditions for a displacment inverter compliant mechanism.

fixed_nodes
Fixed bottom and top left corner nodes.

Type numpy.ndarray

forces
Middle left input force.

Type numpy.ndarray

output_displacement_mask
Middle right output displacement mask.

Type numpy.ndarray

Gripper

class topopt.mechanisms.boundary_conditions.**GripperBoundaryConditions**(*nelx*,
nely)

Boundary conditions for a gripping mechanism.

fixed_nodes
Fixed bottom and top left corner nodes.

Type numpy.ndarray

forces
Middle left input force.

Type numpy.ndarray

Cross Sensitivity

class topopt.mechanisms.boundary_conditions.**CrossSensitivityExampleBoundaryConditio**

Boundary conditions from Figure 2.19 of *Topology Optimization*.

active_elements
Active elements to be set to full density.

Type numpy.ndarray

fixed_nodes
Fixed bottom and top left corner nodes.

Type numpy.ndarray

forces
Middle left input force.

Type numpy.ndarray

output_displacement_mask
Middle right output displacement mask.

Type numpy.ndarray

CHAPTER 10

Graphical User Interfaces (GUIs)

Graphics user interfaces for topology optimization.

10.1 Base GUI

class topopt.guis.**GUI**(*problem*, *title=''*)

Graphics user interface of the topology optimization.

Draws the outputs a topology optimization problem.

__init__(*problem*, *title=''*)

Create a plot and draw the initial design.

Parameters

- **problem** (*topopt.Problem*) – problem to visualize
- **title** (*str*) – title of the plot

init_subplots()

Create the subplots.

plot_force_arrows()

Add arrows to the plot for each force.

update(*xPhys*, *title=None*)

Plot the results.

10.2 Stress GUI

class topopt.guis.**StressGUI**(*problem*, *title=''*)

Graphics user interface of the topology optimization.

Draws the output, stress, and derivative of stress of the topology optimization problem.

__init__(*problem*, *title=''*)

Create a plot and draw the initial design.

init_subplots()

Create the subplots (one for the stress and one for diff stress).

update(*xPhys*, *title=None*)

Plot the results.

CHAPTER 11

Command Line Interface

Command-line utility to run topology optimization.

topopt.cli.**create_parser**(*nelx=180*, *nely=60*, *volfrac=0.4*, *penalty=3.0*, *rmin=5.4*, *ft=1*)

Create an argument parser with the given values as defaults.

Parameters

- **nelx** (int) – The default number of elements in the x direction.
- **nely** (int) – The default number of elements in the y direction.
- **volfrac** (float) – The default fraction of the total volume to use.
- **penalty** (float) – The default penalty exponent value in SIMP.
- **rmin** (float) – The default filter radius.
- **ft** (int) –

  The default filter method to use.

  - 0: *topopt.filters.SensitivityBasedFilter*
  - 1: *topopt.filters.DensityBasedFilter*

Returns

Return type Argument parser with given defaults.

topopt.cli.**main**(*nelx*, *nely*, *volfrac*, *penalty*, *rmin*, *ft*, *gui=None*, *bc=None*, *problem=None*, *filter=None*, *solver=None*)

Run the main application of the command-line tools.

Parameters

- **nelx** (int) – The number of elements in the x direction.

- **nely** (int) – The number of elements in the y direction.
- **volfrac** (float) – The fraction of the total volume to use.
- **penalty** (float) – The penalty exponent value in SIMP.
- **rmin** (float) – The filter radius.
- **ft** (int) –

  The filter method to use.

  - 0: *topopt.filters.SensitivityBasedFilter*
  - 1: *topopt.filters.DensityBasedFilter*
- **gui** (Optional[*GUI*]) – The GUI to use.
- **bc** (Optional[*BoundaryConditions*]) – The boundary conditions to use.
- **problem** (Optional[*Problem*]) – The problem to use.
- **filter** (Optional[*Filter*]) – The filter to use.
- **solver** (Optional[*TopOptSolver*]) – The solver to use.

Return type None

topopt.cli.**parse_args**(*nelx=180*, *nely=60*, *volfrac=0.4*, *penalty=3.0*, *rmin=5.4*, *ft=1*)

Parse the system args with the given values as defaults.

Parameters

- **nelx** (int) – The default number of elements in the x direction.
- **nely** (int) – The default number of elements in the y direction.
- **volfrac** (float) – The default fraction of the total volume to use.
- **penalty** (float) – The default penalty exponent value in SIMP.
- **rmin** (float) – The default filter radius.
- **ft** (int) –

  The default filter method to use.

  - 0: *topopt.filters.SensitivityBasedFilter*
  - 1: *topopt.filters.DensityBasedFilter*

Returns

Return type Parsed command-line arguments.

topopt.cli.**title_str**(*nelx*, *nely*, *volfrac*, *rmin*, *penalty*)

Create a title string for the problem.

Parameters

- **nelx** (int) – The number of elements in the x direction.
- **nely** (int) – The number of elements in the y direction.

- **volfrac** (float) – The fraction of the total volume to use.
- **rmin** (float) – The filter radius.
- **penalty** (float) – The penalty exponent value in SIMP.

Returns

Return type Title string for the GUI.

CHAPTER 12

Development Status

TopOpt is in early stages of development and only features a limited set of finite element mesh options, optimization problems, and solvers. The following is a list of current and future features of TopOpt:

12.1 Meshes

- 2D regular grid
- **2D general mesh**
  - triangle mesh
  - quadrilateral mesh
- 3D regular grid
- **3D general mesh**
  - tetrahedron mesh
  - hexahedron mesh

12.2 Problems

- **compliance**
  - linear elasticity
  - non-linear elasticity
- stress

- thermal conductivity
- fluid flow

12.3 Solvers

- optimality criterion
- method of moving asymptotes (MMA)
- genetic algorithms

CHAPTER 13

Indices and tables

- genindex
- modindex
- search

Python Module Index

t

Index

Symbols

A

B

C

H

I

L

M

N

O

P

S

T

U

V