Adaptation of Ole Sigmund's Topology Optimization Code with 99 Lines to Python

J.F.A. Madeira

May 20, 2024

1 Introduction

The original program was developed by Ole Sigmund in January 2000 and later modified for increased speed in September 2002. This code performs topology optimization using a finite element method (FEM) analysis. The goal is to optimize the material distribution within a given design space subject to specific constraints and objectives, such as minimizing compliance (maximizing stiffness) for a given volume fraction of material. For more details about the original MATLAB code by Ole Sigmund, visit the following link:

> https://www.topopt.mek.dtu.dk/apps-and-software/a-99-line-topology-optimization-code-written-in-matlab

2 Problem Formulation

The problem is formulated to find the optimal material distribution in a design space that minimizes the compliance (maximizes the stiffness) of the structure subject to a volume constraint. The optimization problem can be expressed as:

$$\begin{aligned} \underset{\mathbf{x}}{\text{minimize}} \quad & c(\mathbf{x}) = \mathbf{U}(\mathbf{x})^{\mathrm{T}}\mathbf{K}(\mathbf{x})\mathbf{U}(\mathbf{x}) \\ \text{subject to} \quad & \mathbf{K}(\mathbf{x})\mathbf{U}(\mathbf{x}) = \mathbf{F}, \\ & \sum_{e=1}^{n} x_e \leq \text{volfrac} \cdot n, \\ & 0 \leq x_e \leq 1, \quad e = 1, 2, \ldots, n, \end{aligned}$$

where $\mathbf{x}$ is the vector of design variables representing the material distribution, $\mathbf{K}$ is the global stiffness matrix, $\mathbf{U}$ is the displacement vector, and $\mathbf{F}$ is the force vector. The design variables x_e represent the relative density of material in element e.

3 Code Explanation

Below is a detailed explanation of each function in the Python code, including their purpose and functionality.

3.1 Top Level Function

The main function `top` initializes the design domain and performs the iterative optimization process.

```
    def top(nelx, nely, volfrac, penal, rmin):
    # INITIALIZE
    x = np.ones((nely, nelx)) * volfrac
    loop = 0
    change = 1.0

    # START ITERATION
    while change > 0.01:
    loop += 1
```

```
    xold = x.copy()

    # FE-ANALYSIS
    U = FE(nelx, nely, x, penal)

    # OBJECTIVE FUNCTION AND SENSITIVITY ANALYSIS
    KE = lk()
    c = 0.0
    dc = np.zeros((nely, nelx))
    for ely in range(nely):
    for elx in range(nelx):
    n1 = (nely + 1) * elx + ely
    n2 = (nely + 1) * (elx + 1) + ely
    Ue = U[[2 * n1, 2 * n1 + 1, 2 * n2, 2 * n2 + 1, 2 * n2 + 2, 2 * n2 + 3, 2 * n1 + 2,
    2 * n1 + 3]]
    c += (x[ely, elx] ** penal) * (Ue.T @ KE @ Ue).item()
    dc[ely, elx] = -penal * (x[ely, elx] ** (penal - 1)) * (Ue.T @ KE @ Ue).item()

    # FILTERING OF SENSITIVITIES
    dc = check(nelx, nely, rmin, x, dc)

    # DESIGN UPDATE BY THE OPTIMALITY CRITERIA METHOD
    x = OC(nelx, nely, x, volfrac, dc)

    # PRINT RESULTS
    change = np.max(np.abs(x - xold))
    print(f" It.: {loop:4d} Obj.: {c:10.4f} Vol.: {np.sum(x)/(nelx*nely):6.3f} ch.: {
    change:6.3f}")

    # PLOT DENSITIES
    plt.imshow(-x, cmap='gray')
    plt.axis('equal')
    plt.axis('off')
    plt.pause(1e-6)
    plt.show()
```

3.2 Optimality Criteria Function

The `OC` function updates the design variable using the optimality criteria method.

```
    def OC(nelx, nely, x, volfrac, dc):
    l1 = 0
    l2 = 100000
    move = 0.2
    while (l2 - l1) > 1e-4:
    lmid = 0.5 * (l2 + l1)
    xnew = np.maximum(0.001, np.maximum(x - move, np.minimum(1.0, np.minimum(x + move, x
     * np.sqrt(-dc / lmid)))))
    if np.sum(xnew) - volfrac * nelx * nely > 0:
    l1 = lmid
    else:
    l2 = lmid
    return xnew
```

3.3 Sensitivity Filtering Function

The `check` function applies a sensitivity filter to ensure mesh-independence.

```
    def check(nelx, nely, rmin, x, dc):
    dcn = np.zeros((nely, nelx))
    for i in range(nelx):
    for j in range(nely):
    sum = 0.0
    for k in range(max(i - int(rmin), 0), min(i + int(rmin) + 1, nelx)):
    for l in range(max(j - int(rmin), 0), min(j + int(rmin) + 1, nely)):
    fac = rmin - np.sqrt((i - k) ** 2 + (j - l) ** 2)
    sum += max(0, fac)
    dcn[j, i] += max(0, fac) * x[l, k] * dc[l, k]
    dcn[j, i] /= (x[j, i] * sum)
```

```
    return dcn

```

3.4 Finite Element Analysis Function

The `FE` function performs the finite element analysis.

```
    def FE(nelx, nely, x, penal):
    KE = lk()
    K = sp.lil_matrix((2 * (nelx + 1) * (nely + 1), 2 * (nelx + 1) * (nely + 1)))
    F = sp.lil_matrix((2 * (nely + 1) * (nelx + 1), 1))
    U = np.zeros((2 * (nely + 1) * (nelx + 1), 1))

    for elx in range(nelx):
    for ely in range(nely):
    n1 = (nely + 1) * elx + ely
    n2 = (nely + 1) * (elx + 1) + ely
    edof = np.array([2 * n1, 2 * n1 + 1, 2 * n2, 2 * n2 + 1, 2 * n2 + 2, 2 * n2 + 3, 2 *
     n1 + 2, 2 * n1 + 3])
    K[np.ix_(edof, edof)] += x[ely, elx] ** penal * KE

    # Convert K to CSR format
    K = K.tocsr()

    # DEFINE LOADS AND SUPPORTS (HALF MBB-BEAM)
    F[1, 0] = -1
    fixeddofs = np.union1d(np.arange(0, 2 * (nely + 1), 2), np.array([2 * (nelx + 1) * (
    nely + 1) - 1]))
    alldofs = np.arange(2 * (nely + 1) * (nelx + 1))
    freedofs = np.setdiff1d(alldofs, fixeddofs)

    # SOLVING
    U[freedofs, 0] = spla.spsolve(K[freedofs][:, freedofs], F[freedofs].toarray().
    flatten())
    U[fixeddofs, 0] = 0
    return U

```

3.5 Element Stiffness Matrix Function

The `lk` function computes the local stiffness matrix for each element.

```
    def lk():
    E = 1.0
    nu = 0.3
    k = np.array([1/2 - nu/6, 1/8 + nu/8, -1/4 - nu/12, -1/8 + 3*nu/8,
    -1/4 + nu/12, -1/8 - nu/8, nu/6, 1/8 - 3*nu/8])
    KE = E / (1 - nu**2) * np.array([[k[0], k[1], k[2], k[3], k[4], k[5], k[6], k[7]],
    [k[1], k[0], k[7], k[6], k[5], k[4], k[3], k[2]],
    [k[2], k[7], k[0], k[5], k[6], k[3], k[4], k[1]],
    [k[3], k[6], k[5], k[0], k[7], k[2], k[1], k[4]],
    [k[4], k[5], k[6], k[7], k[0], k[1], k[2], k[3]],
    [k[5], k[4], k[3], k[2], k[1], k[0], k[7], k[6]],
    [k[6], k[3], k[4], k[1], k[2], k[7], k[0], k[5]],
    [k[7], k[2], k[1], k[4], k[3], k[6], k[5], k[0]]])
    return KE

```

4 Detailed Explanations

4.1 Array to Scalar Conversion

In the original code, the expression `(Ue.T @ KE @ Ue)` results in an array with one element, which needs to be converted to a scalar for correct assignment. This is done using the `.item()` method:

```
    c += (x[ely, elx] ** penal) * (Ue.T @ KE @ Ue).item()
    dc[ely, elx] = -penal * (x[ely, elx] ** (penal - 1)) * (Ue.T @ KE @ Ue).item()

```

4.2 CSR Matrix Format for `spsolve`

The `spsolve` function in SciPy requires the matrix to be in CSR or CSC format. The following conversion ensures the matrix `K` is in CSR format:

```
    K = K.tocsr()

```

4.3 Result Printing Format

To print results correctly, the values must be scalars. The conversion using `.item()` ensures that `c` is a scalar:

```
    print(f" It.: {loop:4d} Obj.: {c:10.4f} Vol.: {np.sum(x)/(nelx*nely):6.3f} ch.: {
    change:6.3f}")

```

5 Explanation of `scipy.sparse` and `scipy.sparse.linalg`

`scipy.sparse` is a module in SciPy that provides functions to create and manipulate sparse matrices. Sparse matrices are memory efficient structures for large matrices that contain a lot of zero elements. Using sparse matrices can significantly reduce memory usage and increase computation speed in scientific computing tasks.

`scipy.sparse.linalg` is a module in SciPy that provides linear algebra operations for sparse matrices. This includes solvers for sparse linear systems, eigenvalue problems, and other matrix decompositions. The function `spsolve` from this module is used to solve linear systems of equations where the coefficient matrix is sparse, making it much more efficient for large-scale problems compared to dense matrix solvers.

6 Conclusion

This document details the conversion and explanation of the 99-line topology optimization code from MATLAB to Python. Each function is explained, and key issues related to array to scalar conversion, matrix format, and result printing are addressed to ensure the code functions correctly.

7 Full Python Code

Below is the complete Python code with line numbers for reference.

```
    import numpy as np
    import matplotlib.pyplot as plt
    import scipy.sparse as sp
    import scipy.sparse.linalg as spla

    def top(nelx, nely, volfrac, penal, rmin):
    # INITIALIZE
    x = np.ones((nely, nelx)) * volfrac
    loop = 0
    change = 1.0

    # START ITERATION
    while change > 0.01:
    loop += 1
    xold = x.copy()

    # FE-ANALYSIS
    U = FE(nelx, nely, x, penal)

    # OBJECTIVE FUNCTION AND SENSITIVITY ANALYSIS
    KE = lk()
    c = 0.0
    dc = np.zeros((nely, nelx))
    for ely in range(nely):
    for elx in range(nelx):
```

```
    n1 = (nely + 1) * elx + ely
    n2 = (nely + 1) * (elx + 1) + ely
    Ue = U[[2 * n1, 2 * n1 + 1, 2 * n2, 2 * n2 + 1, 2 * n2 + 2, 2 * n2 + 3, 2 * n1 + 2, 2 * n1 + 3]]
    c += (x[ely, elx] ** penal) * (Ue.T @ KE @ Ue).item()
    dc[ely, elx] = -penal * (x[ely, elx] ** (penal - 1)) * (Ue.T @ KE @ Ue).item()

    # FILTERING OF SENSITIVITIES
    dc = check(nelx, nely, rmin, x, dc)

    # DESIGN UPDATE BY THE OPTIMALITY CRITERIA METHOD
    x = OC(nelx, nely, x, volfrac, dc)

    # PRINT RESULTS
    change = np.max(np.abs(x - xold))
    print(f" It.: {loop:4d} Obj.: {c:10.4f} Vol.: {np.sum(x)/(nelx*nely):6.3f} ch.: {change:6.3f}")

    # PLOT DENSITIES
    plt.imshow(-x, cmap='gray')
    plt.axis('equal')
    plt.axis('off')
    plt.pause(1e-6)
    plt.show()

    def OC(nelx, nely, x, volfrac, dc):
    l1 = 0
    l2 = 100000
    move = 0.2
    while (l2 - l1) > 1e-4:
    lmid = 0.5 * (l2 + l1)
    xnew = np.maximum(0.001, np.maximum(x - move, np.minimum(1.0, np.minimum(x + move, x * np.sqrt(-dc / lmid)))))
    if np.sum(xnew) - volfrac * nelx * nely > 0:
    l1 = lmid
    else:
    l2 = lmid
    return xnew

    def check(nelx, nely, rmin, x, dc):
    dcn = np.zeros((nely, nelx))
    for i in range(nelx):
    for j in range(nely):
    sum = 0.0
    for k in range(max(i - int(rmin), 0), min(i + int(rmin) + 1, nelx)):
    for l in range(max(j - int(rmin), 0), min(j + int(rmin) + 1, nely)):
    fac = rmin - np.sqrt((i - k) ** 2 + (j - l) ** 2)
    sum += max(0, fac)
    dcn[j, i] += max(0, fac) * x[l, k] * dc[l, k]
    dcn[j, i] /= (x[j, i] * sum)
    return dcn

    def FE(nelx, nely, x, penal):
    KE = lk()
    K = sp.lil_matrix((2 * (nelx + 1) * (nely + 1), 2 * (nelx + 1) * (nely + 1)))
    F = sp.lil_matrix((2 * (nely + 1) * (nelx + 1), 1))
    U = np.zeros((2 * (nely + 1) * (nelx + 1), 1))

    for elx in range(nelx):
    for ely in range(nely):
    n1 = (nely + 1) * elx + ely
    n2 = (nely + 1) * (elx + 1) + ely
    edof = np.array([2 * n1, 2 * n1 + 1, 2 * n2, 2 * n2 + 1, 2 * n2 + 2, 2 * n2 + 3, 2 * n1 + 2, 2 * n1 + 3])
    K[np.ix_(edof, edof)] += x[ely, elx] ** penal * KE

    # Convert K to CSR format
    K = K.tocsr()

    # DEFINE LOADS AND SUPPORTS (HALF MBB-BEAM)
    F[1, 0] = -1
    fixeddofs = np.union1d(np.arange(0, 2 * (nely + 1), 2), np.array([2 * (nelx + 1) * (nely + 1) - 1]))
```

```
    alldofs = np.arange(2 * (nely + 1) * (nelx + 1))
    freedofs = np.setdiff1d(alldofs, fixeddofs)

    # SOLVING
    U[freedofs, 0] = spla.spsolve(K[freedofs][:, freedofs], F[freedofs].toarray().
    flatten())
    U[fixeddofs, 0] = 0
    return U

    def lk():
    E = 1.0
    nu = 0.3
    k = np.array([1/2 - nu/6, 1/8 + nu/8, -1/4 - nu/12, -1/8 + 3*nu/8,
    -1/4 + nu/12, -1/8 - nu/8, nu/6, 1/8 - 3*nu/8])
    KE = E / (1 - nu**2) * np.array([[k[0], k[1], k[2], k[3], k[4], k[5], k[6], k[7]],
    [k[1], k[0], k[7], k[6], k[5], k[4], k[3], k[2]],
    [k[2], k[7], k[0], k[5], k[6], k[3], k[4], k[1]],
    [k[3], k[6], k[5], k[0], k[7], k[2], k[1], k[4]],
    [k[4], k[5], k[6], k[7], k[0], k[1], k[2], k[3]],
    [k[5], k[4], k[3], k[2], k[1], k[0], k[7], k[6]],
    [k[6], k[3], k[4], k[1], k[2], k[7], k[0], k[5]],
    [k[7], k[2], k[1], k[4], k[3], k[6], k[5], k[0]]])
    return KE

    if __name__ == "__main__":
    top(60, 20, 0.5, 3, 1.5)

```

8 Parameter Explanations

The function `top` is called with the following parameters: `top(60, 20, 0.5, 3, 1.5)`. Each parameter has a specific role in the optimization process:

- `nelx` **(60)**: Number of elements in the x-direction (horizontal) of the mesh.
  - Defines the resolution of the mesh in the horizontal direction. A higher value leads to a finer mesh, increasing accuracy but also computational cost.
- `nely` **(20)**: Number of elements in the y-direction (vertical) of the mesh.
  - Defines the resolution of the mesh in the vertical direction. Similar to `nelx`, a higher value improves accuracy at the expense of computational resources.
- `volfrac` **(0.5)**: Volume fraction constraint.
  - Controls the fraction of the total volume of material that can be used. A value of 0.5 means only 50% of the total volume is available for the structure.
- `penal` **(3)**: Penalization factor.
  - A parameter used in the SIMP (Solid Isotropic Material with Penalization) method to penalize intermediate densities, promoting a clear 0-1 (void-solid) design.
- `rmin` **(1.5)**: Filter radius.
  - Defines the radius for the sensitivity filter, which smooths out the sensitivities to avoid numerical instabilities and ensure mesh-independence of the solution.

9 Running the Code on Google Colab

For convenience, the code is available and ready to run on Google Colab. You can access the notebook using the following link: https://github.com/aguilarmadeira/OT/blob/main/top.ipynb