ARL-TR-9334 • Oct 2021

DEVCOM ARMY RESEARCH LABORATORY

Generalization and Expansion of Topology Optimization in MATLAB for Scalable High-Resolution Design

by Benjamin S Rathman and Andrew T Gaynor

NOTICES

Disclaimers

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-TR-9334 ● OCT 2021

Generalization and Expansion of Topology Optimization in MATLAB for Scalable High-Resolution Design

by Benjamin S Rathman
Montana Technological University

Andrew T Gaynor
Weapons and Materials Research Directorate, DEVCOM Army Research Laboratory

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED (From - To)
October 2021	Technical Report	January 2020–April 2021

4. TITLE AND SUBTITLE
Generalization and Expansion of Topology Optimization in MATLAB for Scalable High-Resolution Design

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)
Benjamin S Rathman and Andrew T Gaynor

5d. PROJECT NUMBER
AH80

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
DEVCOM Army Research Laboratory
ATTN: FCDD-RLW-MB
Aberdeen Proving Ground, MD 21005-5066

8. PERFORMING ORGANIZATION REPORT NUMBER
ARL-TR-9334

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution is unlimited.

13. SUPPLEMENTARY NOTES
corresponding author's email: <andrew.t.gaynor2.civ@army.mil>.

14. ABSTRACT
Generally, topology optimization codes are designed to find a solution for a single type of problem. For educational codes, this tends to be a 2-D problem solving for minimum compliance (maximum stiffness). This report presents a single algorithm that broadens the capabilities for topology optimization in MATLAB. The presented algorithm has been designed to optimize 2- and 3-D problems from a selection of objective functions. Provisions were also added to allow for the optimization of geometrically complex components using existing mesh (STL) files. The algorithm does this while maintaining a high degree of control over the optimization process and increasing efficiency (allowing for fast and high-resolution solutions).

15. SUBJECT TERMS
topology optimization, TO, additive manufacturing, AM, resolution

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			Andrew T Gaynor
Unclassified	Unclassified	Unclassified	UU	60	19b. TELEPHONE NUMBER (Include area code) 410-306-0825

Contents

List of Figures

List of Tables

Acknowledgments

Funding for this project was provided by the US Army Combat Capabilities Development Command Army Research Laboratory under Cooperative Agreement W911NF-20-2-0202.

Benjamin Rathman wishes to thank Dr Andrew Gaynor for his help in the development of this code. Rathman also wishes to thank Dr Andelle Kudzal for the opportunity to work on this research as well as her help and advice.

1. Introduction

Topology optimization (TO) is the process of redistributing material within a design envelope using finite element analysis (FEA) in a rigorous optimization scheme to make a component more efficient. Often, this design envelope is the outer surfaces of the component itself. Often, the driving mechanism of using TO is creating a resulting component that is lighter while maintaining the original mechanical properties. Additionally, the component can be optimized to achieve a certain goal with respect to a fixed volume fraction or minimize the volume with respect to a fixed goal. These goals, known as objective functions, include maximizing stiffness (minimum compliance),[1] minimizing stress, or changing the heat transfer properties.

The combination of additive manufacturing (AM) processes and TO design algorithms provides several opportunities for component and system modernization. One example is the Army's Next Generation Combat Vehicle cross-functional team. The lightweighting of structural components in combat vehicles facilitates greater maneuverability of the vehicle, allowing Soldiers to get out of harm's way. Conversely, the same design tools can be applied to redistribute weight from the structural system to other systems, such as those of armor or lethality.

This report presents the progress that has been made toward creating a MATLAB code for TO that can be used for a wide range of optimization problems while still allowing for a high degree of control. In addition, the code was designed to have good mesh scaling properties, allowing for high-resolution design and thus the capability to approximate the resolution of AM processes more closely.

This report is organized as follows. The 99-line code and its evolutions are reviewed in Section 2. The new algorithm is described in Section 3. Several extensions and expansions that have been added to the code are described in Section 4. Section 5 goes through the process of coding in a problem for optimization. In Section 6, the performance of this algorithm is examined in comparison with its predecessors. Conclusions are given in Section 7. The relevant code is provided in Appendixes A–G.

2. The 99-Line Code and Its Evolution

In 2001, Sigmund published a 99-line MATLAB code for solving 2-D TO problems for minimum compliance.[2,3] This algorithm was designed as an educational tool for engineers to learn about the process of TO and aid in the development of the user's intuition regarding optimal designs. As a result of the program's simplicity, the user has complete control over the optimization process when compared to commercial solutions, with a few limitations. Problems are restricted to the use of the linear elastic material model for simplification of the FEA equations. Additionally, to simplify implementation, problems are relegated to a structured, rectangular mesh using linear rectangular and hexahedron elements for 2- and 3-D problems, respectively. While there is no inherent 3-D capability in the 99-line code, other codes using Sigmund's as the base have added this capability.

First, an FEA function is run to solve for the global displacement vector. The displacement vector is used to calculate compliance, ensuring the compliance is decreasing, using the following objective function:

$$\begin{aligned} \min_{\boldsymbol{\phi}} \quad & c(\boldsymbol{\phi}) = \mathbf{U}^{\mathbf{T}}\mathbf{K}\mathbf{U} \\ \text{subject to:} \quad & \mathbf{K}(\boldsymbol{\phi})\mathbf{U} = \mathbf{F} \\ & \frac{\sum_{e\in\Omega} x(\boldsymbol{\phi})_e v_e}{V} \leq f \\ & 0 \leq \phi^i \leq \phi^{\max} \quad \forall\, i \in \Omega, \end{aligned} \tag{1}$$

where c is compliance, $\boldsymbol{x}$ is the vector of elemental densities, and $\boldsymbol{\phi}$ is the vector of design variables. $\mathbf{U}$, $\mathbf{F}$, and $\mathbf{K}$ represent the global displacement vector, global force vector, and global stiffness matrix, respectively. v_e is the elemental volume, V is the total volume in the design domain, and f is the prescribed volume fraction. Finally, Ω is the design domain.

The 99-line code then uses a gradient-based optimization approach that assumes the material is homogeneous within each element. Each element is related to a design variable scaled by that element's density, where an element density of 0 implies no material density, or void, and an element density of 1 implies full material density. Since intermediate density materials are not achievable, binary solutions must be promoted. One prominent method to achieve this uses the solid isotropic material

with penalization (SIMP) interpolation method. The SIMP method (shown in Eq. 2) is used to promote a binary solution by penalizing intermediate densities.

$$E(\hat{x}_e) = E_{\text{min}} + {x_e}^p(E_0 - E_{\text{min}}), \tag{2}$$

where E, E_0, and E_{min} represent the Young's moduli of the element, solid phase, and void phase, respectively; x_e represents the element density; and p represents the penalization factor. As shown in Eq. 2, the penalization process is performed by applying an exponential function on the domain such that low-density elements are forced to a zero-density condition and high-density elements are not affected as much. For example, with no penalization ($p = 1$), an element with a density of 0.5 (50%) equates to a 50% stiffness. With a penalization factor of 3, the same element would equate to a 12.5% stiffness.

After the SIMP method is applied, the gradient is filtered using one of the several available filtering methods to ensure the minimum length scale is being achieved. The minimum length scale is a prescribed parameter that describes the minimum feature size. Any feature smaller than this parameter is purged from the gradient.

Finally, the gradient is entered into the Optimality Criteria (OC) optimizer to update the design variables. The optimizer uses a bi-sectioning algorithm to find the Lagrangian multiplier that fulfills the prescribed volume fraction.

This entire process is repeated until the convergence criteria are met. The structure of Sigmund's code relies on several functions, albeit in the same script, to find an optimized solution. Sigmund's code utilizes a main function to run the optimization, with several child functions to do various calculations for each iteration.

Over the past two decades, several updates and extensions have been made to Sigmund's 99-line code. Some significant contributions are noted as follows. In 2003, Bendsøe and Sigmund published a book, which in addition to giving an introduction to the field of TO, details some extensions and alternate objective functions for the 99-line code.[1] In 2011, Andreassen et al. published an updated version of the 99-line code, the 88-line code, which utilized some of the functions MATLAB had made available in the previous decade to make the code more computationally efficient.[4] In addition, some of the computations were made significantly more

efficient by eliminating for loops within the optimization loop. For example, Andreassen moved the calculation of indices for assembling of the global stiffness matrix outside the optimization loop. This change eliminated a double nested for loop and redundant calculations present in the 99-line code, replacing the assembly with a single efficient function call to `sparse`. In 2014, Liu and Tovar published an expanded version of Andreassen's code that allowed for solving 3-D solutions, while keeping the same framework.[5] In the following years, Liu and Tovar have posted several tutorials on their website (www.top3d.app) for adding functionality to the code. In 2020, Ferrari and Sigmund published an updated version of Andreassen's 88-line code, allowing further speedups and savings in computational cost.[6]

Ferrari's code, consisting of the files `top99neo.m` and `top3D125.m` for 2- and 3-D problems, respectively, was built to update Sigmund's code to incorporate various speedups and reduce the computational cost. By optimizing the code layout and using recent and third-party MATLAB functions, there has been a significant increase in the performance of the algorithm. These speedups included the assembly of the global stiffness matrix, the OC update, the use of MATLAB's `imfilter`, and the use of `fsparse`. Since the OC update was not used in this work, the details of this speedup are not reviewed.

In the assembly of the stiffness matrix, two major improvements were made. First, two vectors used in the assembly of Andreassen's 88-line code, referred to as `iK` and `jK`, used MATLAB's default numeric data type (double). Although this works, `iK` and `jK` can become very large and take up a substantial amount of random-access memory (RAM). To combat this issue and because the values of these vectors were integers anyway, Ferrari changed the data type of the `iK` and `jK` vectors from double to int32, reducing the RAM usage of each element from 8 to 4 bytes. The second major improvement was recognizing that both the elemental and global stiffness matrices are symmetrical. This means the optimization calculations can be done using a sparse triangular matrix instead of the full matrix.

The other speedups have also impacted the efficiency. One is the use of MATLAB's `imfilter` function, which allows the filtering operations to be completed more efficiently. The other is the use of `fsparse`, a function developed by Engbolm and Lukarski.[7] The `fsparse` function is more efficient than MATLAB's `sparse` function and is also able to work with both the double and integer data types, al-

lowing for the new global stiffness matrix assembly to be implemented.

The Ferrari code was chosen as the base function for this work because of its efficiency and the ability to solve both 2- and 3-D problems using essentially the same code.

3. The Works Algorithm

Unfortunately, due to the requirement of this implementation of TO being simple to understand, Ferrari's code sacrifices performance for readability. In an effort to further increase the performance of the program and generalize the program to solve for objectives other than minimum compliance, `topTheWorks.m` has been developed (see Appendix B). While `topTheWorks.m` has the same general implementation as the two Ferrari scripts, the structure of the code has been modified to resemble Sigmund's code more closely, utilizing multiple child functions, to allow the algorithm to be more modular and extendable.

Starting from the published version of Ferrari's code, several major structure and implementation changes were made. These changes include the implementation of the multigrid preconditioned conjugate gradients (MGCG) solver and the method of moving asymptotes (MMA) optimizer, and restructuring the system to generalize the script.

The MGCG solver[8] reduces the computational cost of assembling the displacement vector, $\mathbf{U}$. This is done by calculating the solution using a prescribed number of grid levels at subsequently higher resolutions. The calculation is first carried out on a low-resolution grid. As the grids become progressively higher resolution (in this implementation, for a 2-D problem, each element of a grid is split into four elements for the next grid), the solution is calculated using the coarser grid solution as a template. The relevant functions were placed in the file `multigrid.m` and modified to allow for the calculation of elements with different numbers of degrees of freedom (DoFs) (see Appendix F). Unfortunately, due to the way the MGCG solver is written, Ferrari's speedup utilizing the symmetry of the global stiffness matrix could not be used.

The MMA optimizer[9] has replaced the OC optimizer in this code. While the OC optimizer is good as a heuristic approach for minimum compliance algorithms, it

is limited in function. The MMA optimizer is much more well known and well used for design update. Additionally, the MMA optimizer works better for objective functions other than minimum compliance. The relevant functions were placed in the file `mma.m`.

The algorithm has also been generalized in several ways. The Ferrari code (as well as its predecessors) was designed as functions, allowing the input arguments to be changed readily, but some changes, such as the load conditions, have to be made within the function. To remedy this, a template, `templateTO.m` (see Appendix A), was created as a sister program for `topTheWorks.m`, allowing multiple optimization problems to be solved without adjusting the core functions. To accomplish this, several changes needed to be made to how the optimization algorithm was called. In addition to the standard input arguments for optimization (`nelx`, `nely`, `nelz`, `volfrac`, `penal`, and `rmin`), several other parameters were pulled out of the code and converted to input arguments. First, the assembly of the force ($\mathbf{F}$) and `fixed` vectors and the `passive` matrix, representing the applied load, supports, and domain, respectively, have been made into input arguments. Additionally, the material properties have been pulled out of the code and made into the input argument `mat`, which is a vector consisting of E_0 (Young's modulus of the solid), E_{min} (Young's modulus of the void), ν (Poisson's ratio), σ_Y (yield strength), σ_{UTS} (ultimate tensile strength), K_0 (thermal conductivity of the solid), and K_{min} (thermal conductivity of the void). Additional input arguments have been introduced to add functionality: `obj`, which defines the objective function; `resol`, which denotes the resolution of the part in elements per unit length; `ft`, which defines the filtering scheme being used; `maxit`, which defines the maximum allowable number of iterations; and `nl`, which denotes the number of grid layers used in the MGCG solver.

In addition, two more fundamental generalizations were made. The first was to expand the capability of a single code to solve both 2- and 3-D optimization problems. The second was to allow multiple objective functions to be solved.

Finally, several smaller generalizations were made. First, the assembly of the elemental stiffness matrix, $\mathbf{K}^{\mathbf{e}}$, has been generalized so that instead of this matrix being calculated from a series of constants, the matrix is now calculated based off the prescribed material properties and is no longer restricted to square elements.

This additional functionality, while more computationally expensive, is only done once and makes the code more robust. It is worth noting that the use of an actual instead of a unit Young's modulus causes the multigrid calculations to take significantly longer. Second, the code was expanded to solve for multiple load cases in the same solution. Lastly, the MMA variables were scaled to allow for the optimizer to properly work through the problem using actual material properties, instead of the unit material properties that were used in the 99-line and subsequent codes.

To facilitate the use of multiple objective functions and filtering schemes, several parts of the code were placed into child functions. This decision was made to increase readability and allow for later expansion of the code without compromising the working base function. The functions are structured as shown in Table 1.

Table 1 `topTheWorks.m` function structure

Function	Description
`templateTO.m`	Template for defining the optimization problem
`topTheWorks.m`	TO algorithm
`objective.m`	Calculates FEA solutions based on the defined objective function
`objMatrices.m`	Assembles elemental property matrices based on the defined material properties
`filtertype.m`	Applies a defined filter to the domain, ensuring adherence to the imposed minimum length scale
`multigrid.m`	Applies the MGCG solver to the problem
`mma.m`	Applies the MMA optimizer to the problem

The functions relating to solving the objective function, such as those used for FEA, were placed in the file `objective.m` (see Appendix C). The functions used for calculating the elemental stiffness matrix, constitutive matrix, strain displacement matrix, and elemental conduction matrix were placed in the function `objMatrices.m` (see Appendix D). The functions relating to the filtering scheme were placed in the file `filtertype.m` (see Appendix E).

In addition to the optimization functions, several third-party MATLAB functions were used in the creation of this algorithm. The `findND` function acts similarly to MATLAB's built-in `find` function, but extends the functionality to more dimensions (MATLAB's `find` is restricted to two dimensions).[10] This function is used in the assembly of the load and support vectors. The `vtkwrite` function converts a matrix to the VTK file format.[11] This function is used to visualize the optimized solutions in ParaView.[12]

4. Extensions

A number of extensions were made to the base code including various filtering schemes, the ability to optimize over arbitrarily complex geometries, and the ability to optimize for thermal compliance (heat conduction).

4.1 Filtering Schemes

In Andreassen's 88-line code two filtering methods were provided: sensitivity filtering and density filtering. Ferrari's code added the use of a projection-based filtering method. Each of these methods focuses on controlling the minimum length scale of the solid phase. While this functionality is very useful, it is sometimes desirable to control the minimum length scale of the void phase, especially in applications where sharp internal corners are not desired.

The Heaviside projection method (HPM) was originally developed to achieve implicit control on the minimum length scale of the solid phase using Eq. 3.[13]

$$x_{e1} = 1 - e^{-\beta\mu^e(\boldsymbol{\phi})} + \mu^e(\boldsymbol{\phi})e^{-\beta}, \tag{3}$$

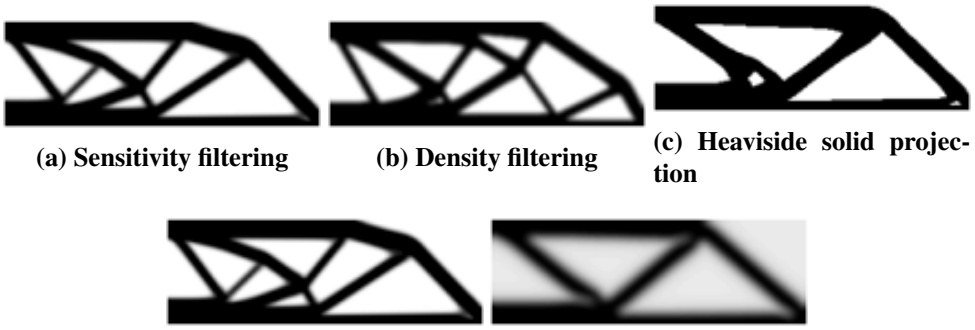
where x_{e1} represents the element density for the solid phase, $\mu^e(\phi)$ represents the projection intensity of an element, and β represents the curvature of regularization. To adjust this filtering method to project the void phase, a simple modification is made, resulting in Eq. 4.[14]

$$x_{e0} = e^{-\beta\mu^e(\boldsymbol{\phi})} - \mu^e(\boldsymbol{\phi})e^{-\beta}, \tag{4}$$

where x_{e0} represents the element density for the void phase. Controlling only the void phase can cause the resulting solution to have very thin features that are not easily manufacturable. To achieve better results while still being able to control the minimum length scale of the void phase, multiphase Heaviside projection can be used, although it is more computationally expensive. In the multiphase projection method, the minimum length scale of both the solid and void phases are controlled. Equations 3 and 4 are superposed to create the combined projection equation.

$$x_e = \frac{x_{e0} + x_{e1}}{2}. \tag{5}$$

Since the elemental projection intensities of the solid and void phases are different, the two projection equations do not cancel out. Instead, for each element, the two elemental densities are averaged. In standard Heaviside projection algorithms, beta is controlled by a continuation scheme to avoid convergence to poor local minima.[13,14] To make the HPMs more efficient, the beta continuation was eliminated using a modified Heaviside filtering scheme.[15] Examples of the sensitivity, density, and the Heaviside filtering methods are displayed in Fig. 1.



(a) Sensitivity filtering **(b) Density filtering** **(c) Heaviside solid projection**

(d) Heaviside void projection **(e) Heaviside multiphase projection**

Fig. 1 Optimal topologies for the simply supported beam with load on center problem with various filtering and projection schemes applied

To accommodate use of all the mentioned filtering methods, `filtertype.m` was developed. The second argument when calling the `filtertype` function is `ft`, which defines the filtering method being used. The values of `ft` and their associated filtering methods are listed in Table 2.

Table 2 Filtering methods

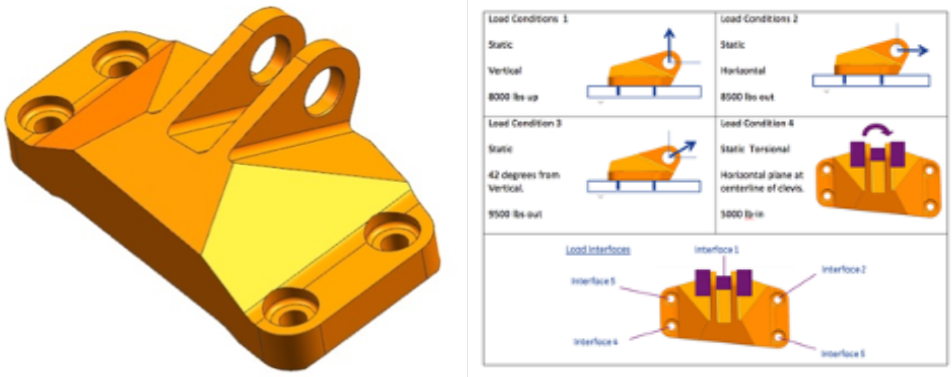
Value of `ft`	Filtering method
1	Sensitivity
2	Density
3	Heaviside projection (solid phase)
4	Heaviside projection (void phase)
5	Multiphase Heaviside projection

4.2 Complex Geometries

Since the Ferrari code and its predecessors are designed as learning tools, there is a simplifying assumption made regarding the geometries of the components being optimized. Generally, the geometries being solved take up the full domain of the matrix (a rectangle for 2-D and rectangular prism for 3-D problems). This problem can be fixed by defining passive elements (as opposed to active elements) in the problem. Passive elements are user-defined elements that the solver cannot change the density of. Elements can be defined as passive solid, passive void, or active (the default state of an element). Ever since Sigmund's code, the algorithm has had the capability for adding passive elements, but their use, at least in literature, has been relegated to examples such as passing a pipe through a beam.

The additional functionality of having complex geometries allows the user to solve optimization problems of more complex parts. Additionally, this functionality allows initial guesses to be imported into the optimizer, known as seeding. The seeding process is used to promote feature growth in a certain way. For example, a conventional finned heatsink can act as a possible seed for a heat transfer problem.

To find solutions for more complex geometries, a method has been developed using passive elements to define the design space. As a proof-of-concept problem, the GE bracket was used from GrabCAD's "GE jet engine bracket challenge".[16] The object of the challenge was to optimize a given component subjected to provided load cases (shown in Fig. 2[16]), making this problem a perfect test case.



(a) Original GE bracket geometry

(b) Design load cases

Fig. 2 Problem definition for the GE bracket challenge problem

The geometry was first imported into MATLAB using the `VOXELISE` function. This function uses the `READstl` function and is designed to create a 3-D Boolean matrix of voxels from an existing triangular-polygon mesh.[17] A "voxelized" version of the GE bracket displayed in ParaView is shown in Fig. 3.

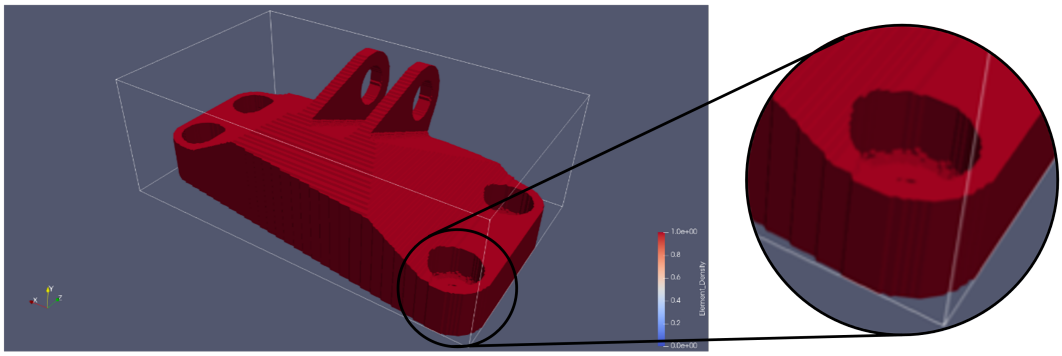


Fig. 3 "Voxelized" GE bracket (268 $\times$ 92 $\times$ 160 elements)

The geometry displayed in Fig. 3 represents the active elements for the optimization process. The "voxelized" GE bracket was then inverted to create the passive matrix. This passive matrix defined the all elements that were not in the GE bracket as passive void, allowing the complex geometry to be maintained through the optimization process.

Using a technical drawing of the GE bracket drawn in SOLIDWORKS[18] from the provided STEP file, dimensions for the load interfaces were retrieved. Through use of MATLAB's `permute` and `ipermute` functions and several if statements and for loops, the interfaces were "extruded" in MATLAB.

The load cases were defined using the same conventions as those in the Ferrari code. The four given load cases were programmed in to make sure that the solutions were coming out as expected. Finally, the combined load case was programmed in and the program run, resulting in the optimized solution in Fig. 4.

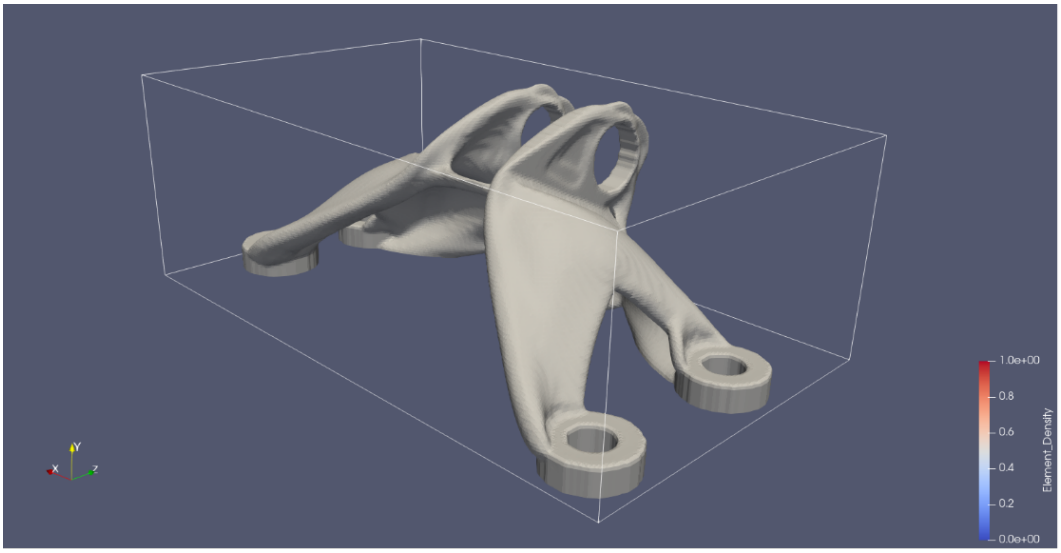


Fig. 4 Optimized GE bracket topology

4.3 Heat Transfer

As alluded to earlier, the functionality of this algorithm has been extended to multiple objective functions. In particular, the heat conduction objective function was looked at and added to the `objective` and `objMatrices` functions. The problem statement was proposed by Bejan and is as follows: "Consider a finite-size volume in which heat is being generated at every point and which is cooled through a small patch (heat sink) located on its boundary. A finite amount of high conductivity material is available. Determine the optimal distribution of material through the given volume such that the highest temperature is minimized".[19] Realistically, the solution can also be used as a heatsink from a point of high heat. This is done by using the objective function shown in Eq. 6.[20]

$$\begin{aligned} \min_{\boldsymbol{\phi}} \quad & c(\boldsymbol{\phi}) = \mathbf{U}^{\mathbf{T}}\mathbf{K}\mathbf{U} \\ \text{subject to:} \quad & \mathbf{K}(\boldsymbol{\phi})\mathbf{U} = \mathbf{F} \\ & \frac{\sum_{e\in\Omega} x(\boldsymbol{\phi})_e v_e}{V} \leq f \\ & 0 \leq \phi^i \leq \phi^{\max} \quad \forall\, i \in \Omega, \end{aligned} \tag{6}$$

where c is now the thermal compliance. $\mathbf{U}$, $\mathbf{F}$, and $\mathbf{K}$ represent the global temperature vector, global heat load vector, and global heat conduction matrix, respectively.

As shown in Eqs. 1 and 6, the system of equations for structural and thermal compliance are the same. To apply the heat conduction problem, the fixed vector is considered the heatsink ($T = 0$ condition) and all other elements have a heat flux applied, becoming part of the F vector.

The implementation of the heat conduction objective function only required a couple of modifications. The first modification was to add the assembly of the heat conduction matrix to `objMatrices`. The heat conduction matrices were taken from the 91-line MATLAB code for heat conduction[1] and the heat conduction modification for Liu and Tovar's 3-D optimization code[21] for the 2- and 3-D conditions, respectively. Next, the code was modified to generate one DoF for each element (instead of the two or three for 2- or 3-D minimum compliance, respectively). Finally, the appropriate FEA equations were added to `objective`.

To test the functionality, three load cases were programmed in: the sphere sink, the cylinder sink, and the plate sink. The cylinder sink is shown in Fig. 5.

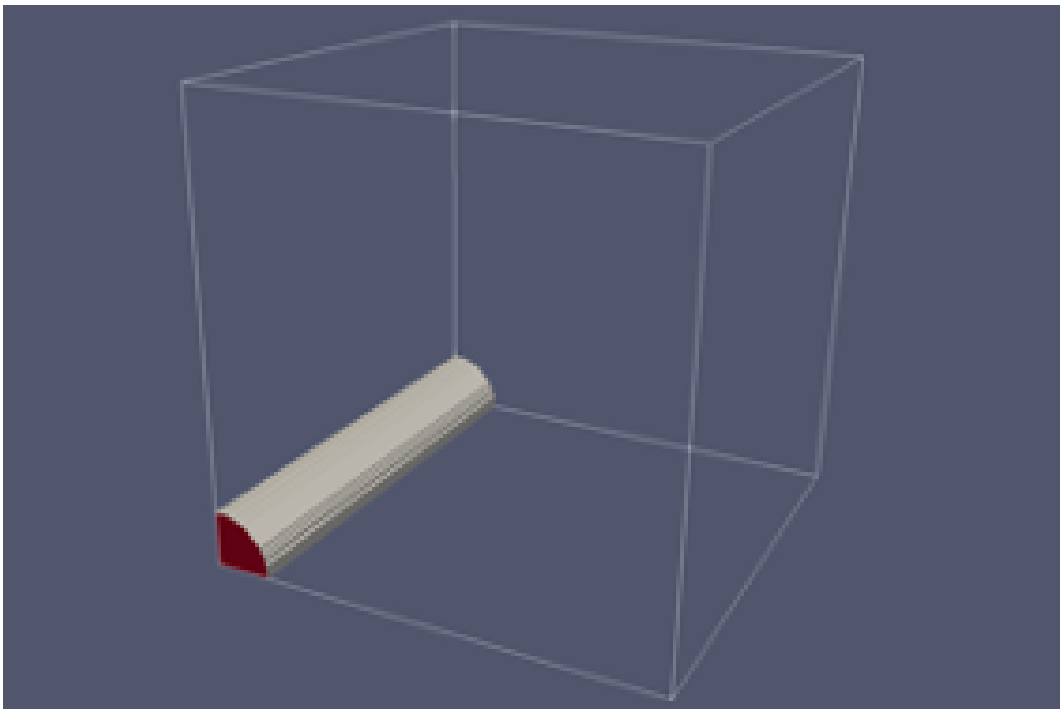


Fig. 5 Cylinder heat sink design case (1/4 domain)

Using the heat conduction objective function and a volume fraction of 0.25 (25% of the original domain), the following solution was found for the cylinder load case (Fig. 6).

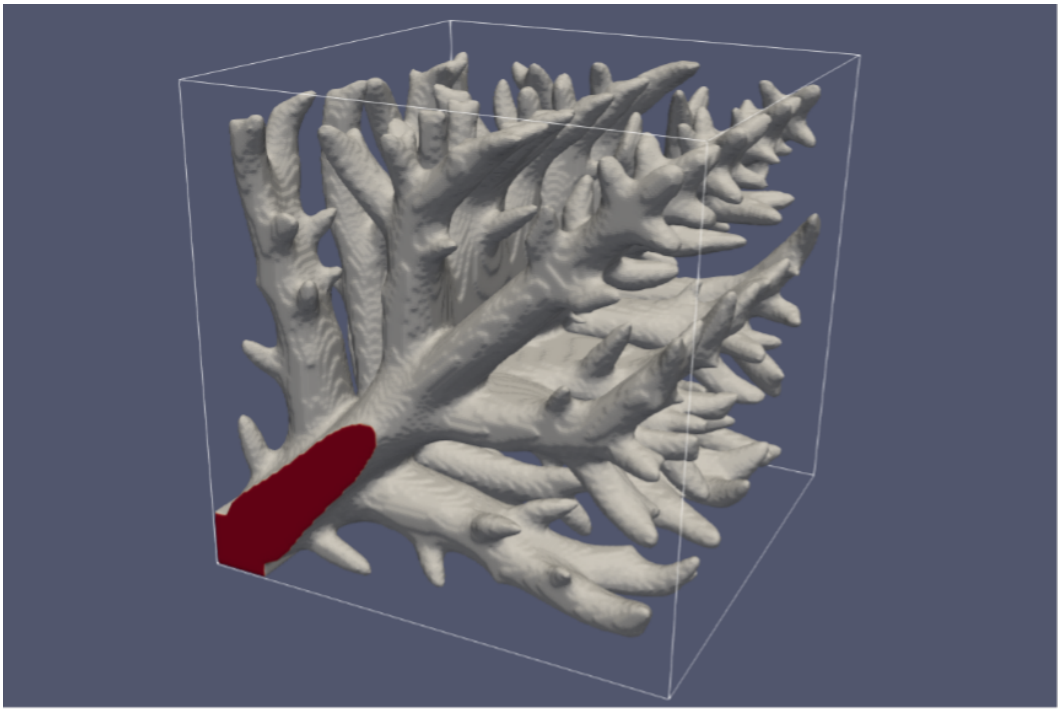


Fig. 6 Optimized solution for the cylinder heatsink

As shown in Fig. 6, the solution generated dendritic structures, which agrees with solutions in shown in literature for heat conduction optimization. There is a method of optimization that reduces the dendritic structures,[22] but this method was not applied in this research.

To allow for the use of multiple objective functions in one algorithm, `objective.m` was developed. The first argument when calling the `objective` function is `obj`, which defines the objective function being used. The values of `obj` and their associated objective functions are listed in Table 3.

Table 3 Objective functions

Value of `obj`	**Objective function**
1	Structural compliance
2	Stress (WIP)
3	Thermal compliance (heat conduction)

5. The Works Problem Setup: GE Bracket Problem Example

As an example of the process used to set up an optimization problem, the GE bracket problem is set up using `templateTO.m` (see Appendix G). First, the geometry must be imported into MATLAB. Before calling the `VOXELISE` function, the following arguments need to be defined: `x` (the number of elements in the x-direction), `y`, `z`, and the filename. For this process to be mesh independent, `x`, `y`, and `z` are defined by the following lines of code:

```
stl = 'original';    % Filename of the geometry being imported
xd = 7.029;          % x-dimension
yd = 2.461;          % y-dimension
zd = 4.262;          % z-dimension
resol = 15;          % Part resolution (ele./unit)
nl = 3;              % Number of grid levels used in TO (MGCG)
loadCase = 0;        % Load condition
radj = 2^(nl-1)*round(resol*[xd,yd,zd]/2^(nl-1));                    %
    ↪ Convert dimensions to elements
[x,y,z] = deal(radj(1),radj(2),radj(3));
```

The variables `xd`, `yd`, and `zd` represent the actual dimensions of the GE bracket in inches. The variables `resol` and `nl` (used in the calculation of `radj`) are used to adjust the number of elements in each direction. Since the coarsest layer using the MGCG solver is made up of elements from the prescribed mesh, the number of elements in each direction must be divisible by two to the power of the number of layers used in the solver minus one. Next, the filename of the geometry to be imported is defined as a string in the variable `stl` without the file extension. `VOXELISE` is called using the lines:

```
stldomain = VOXELISE(x,y,z,[stl,'.stl']);
domain = double(~stldomain);
```

A few attempts will likely be required to make sure that the coordinate system being used by the `VOXEISE` function is the same as is used by `x`, `y`, and `z`. If the coordinate systems are not the same, the aspect ratios on the part will not be correct. This error can be corrected by finding the correct permutation of `x`, `y`, and `z`. The domain is then defined by inverting the Boolean matrix that is output by `VOXELISE` because the optimization function regards values of 0 in the passive matrix as active elements, values of 1 as passive void, and values of 2 as passive solid. Finally, since the entirety of the matrix is not being used, `domainVol` is calculated as a correction factor for the prescribed volume fraction using the following statement:

```
36 domainVol = 1-mean(domain(:));
```

The next step in the formulation of the GE bracket problem is to define the loaded and supported nodes. The load and support elements were then found and defined as passive solid elements, as shown in Fig. 7.

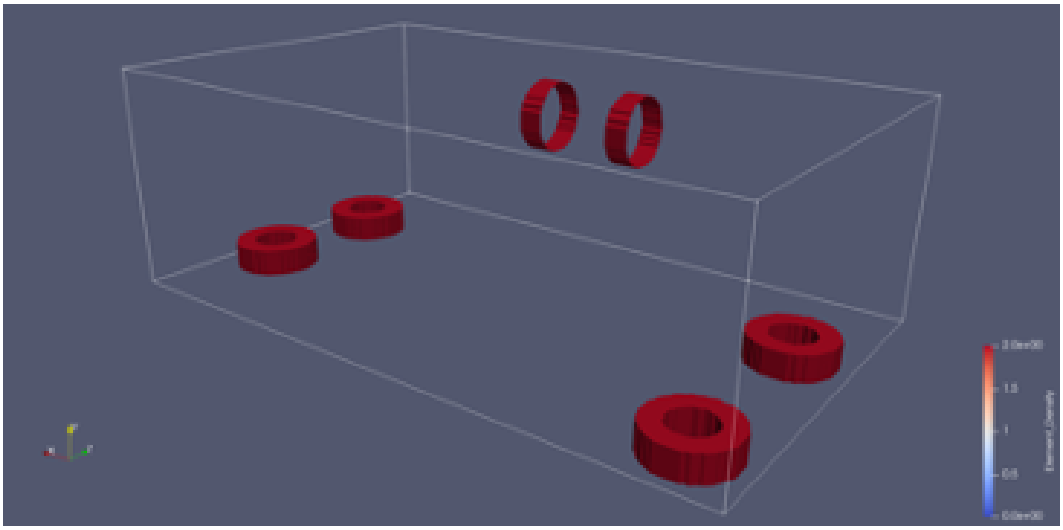


Fig. 7 Defined support and load elements

Interface 1 in Fig. 2b was only defined as the inside ring of elements while interfaces 2–5 were defined to include the points where the bolt heads would interact with the bracket. This is done by first defining the loaded and supported elements. The support elements were found using the technical drawing of the GE bracket, where the origin of the dimensions is the element with coordinates (1,1,1) in MATLAB. To make the feature positions mesh independent, the lengths were divided by the

component length. The radii of the support holes and the counterbores were also defined using this method. Lines 56–62 define these interfaces.

To define the elements that act as support elements, a cylindrical tube of support elements with an inner diameter of the hole diameter and an outer diameter of the counterbore diameter was "extruded" through the geometry. First, the domain was reoriented using MATLAB's `permute` function. The support elements were extruded using lines 63–85.

For each layer, an element is checked to make sure that the bracket is still being looked at. For each element in the layer, that element is checked to see if the element lies within any of the defined cylindrical tubes. If it does, then the element is defined as a support element, which is later redefined as passive solid. Any elements within the holes are redefined as passive void. Finally, the domain is put back into its initial orientation. After the supported elements are defined, the DoFs at those elements must be fixed and the support vector assembled using lines 87–92. Since each node has three DoFs (one for each direction), all three DoFs must be fixed for each node, achieved in line 92.

To define the loaded elements, the same process is used as was used to define the supported elements, using lines 95–130. The main difference is that the "extrusion" process is done twice. This was done so that the two sides of interface 1 can be loaded in different directions for the torque load case (load condition 4 in Fig. 2b). A simplifying assumption was made for load condition 4, allowing the torque to be applied as two linear forces (one on each half of interface 1) in opposite directions. The next step is to define the load DoFs from the loaded elements. This process is completed in three steps: the definition of the loaded DoFs for the linear forces (load conditions 1–3 in Fig. 2b), the definition of the loaded DoFs for the torque in one direction, and the definition of the loaded DoFs for torque in the other direction. This was done in lines 132–154. Then, the load vectors, $\mathbf{F}$, were assembled using lines 156–174.

In the first argument of `fsparse`, the DoFs are plugged in, but the direction is also defined. The convention is to add -1 for a force parallel to the direction along which `nelx` is defined, add 0 for `nely`, and add 1 for `nelz`. The values of `loadcase` are equivalent to the load condition numbers from the GE bracket problem definition. The `loadcase = 0` condition accounts for all of the load conditions in the

same solution. Finally, the optimization function is called using the line:

```
178 xPhys = topTheWorks(obj,mat,x,y,z,resol,volfrac*domainVol,penal,rmin,ft,...
179                     maxit,nl,passive,F,fixed);                    % Run
                        ↪ topology optimization
```

6. Performance

Since the `topTheWorks` function is based off of Ferrari's code, the performance increases inherent in the Ferrari's published code is present in this code. With the addition of the MGCG solver and the MMA optimizer, the code is even more computationally efficient. Figure 8 shows an informal comparison of the published 3-D optimization algorithms from Liu and Tovar, Ferrari, and this code using the minimum compliance objective function and a unit Young's modulus. For this testing, a desktop with an AMD Ryzen 2600, 32 GB of RAM, running Windows 10 and MATLAB R2018b was used.

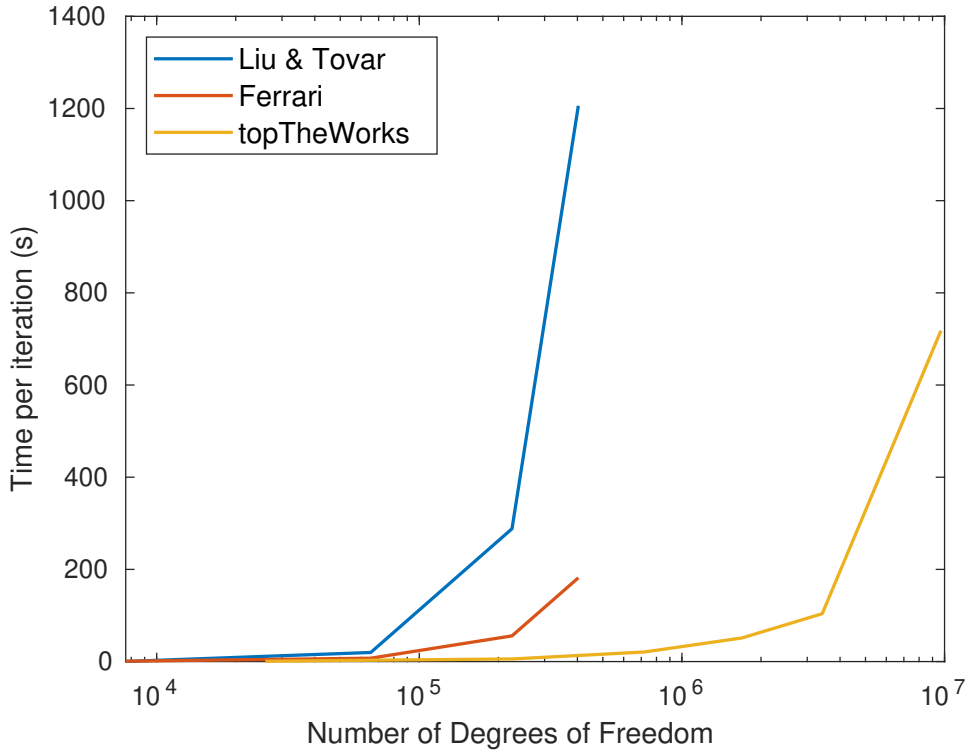


Fig. 8 Comparison of 3-D TO algorithms

The number of DoFs was increased until the algorithm became impractical to run or the algorithm failed due to memory limitations. As shown, the Liu and Tovar code became impractical to run at just over 400,000 DoFs due to the amount of time per iteration the program was taking to run. At this same point, the Ferrari code failed in the assembly of the `L` matrix. This code was able to get up to about 10 million DoFs before running into memory issues.

7. Conclusion

This report presents a MATLAB code for TO that was developed to be used on a wide array of optimization problems. Several design objectives for this code have been achieved:

1. A high degree of control over the optimization process can still be used.
2. High-resolution solutions of approximately 10 million DoFs can be found.
3. The code has been restructured and a method developed to allow for optimization problems with complex geometries to be solved.
4. The code is structured in a way allowing for modules such as different objective functions and filters to be used without modifying the optimization framework.

This code allows for fast, high-resolution solutions to be found, providing smoother solutions that can be easily manufactured using AM processes. With the readability of MATLAB, and the ability of the code to run in both local computation and high-performance computation environments, the barrier to entry for highly scalable, fast, generalized TO is reduced. Additionally, the code's ability to use different objective functions and filtering methods make it very flexible. Using this code, optimized solutions can be found and used in compliance with projects such as the lightweighting of structural components in combat vehicles, allowing for the modernization of systems.

8. References

1. Bendsøe MP, Sigmund O. Topology optimization: theory, methods, and applications. Springer; 2004.

2. Sigmund O. A 99-line topology optimization code written in MATLAB. Structural and Multidisciplinary Optimization. 2001;21:120–127.

3. MATLAB. version r2018a. The MathWorks Inc.; 2018.

4. Andreassen E, Clausen A, Schevenels M, Lazarov B, Sigmund O. Efficient topology optimization in MATLAB using 88 lines of code. Structural and Multidisciplinary Optimization. 2011;43:1–16.

5. Liu K, Tovar A. An efficient 3D topology optimization code written in MATLAB. Structural and Multidisciplinary Optimization. 2014;50:1175–1196.

6. Ferrari F, Sigmund O. A new generation 99 line MATLAB code for compliance topology optimization and its extension to 3D. Structural and Multidisciplinary Optimization. 2020;62:2211–2228.

7. Engblom S, Lukarski D. Fast matlab compatible sparse assembly on multicore computers. Parallel Computing. 2016;56:1–17.

8. Amir O, Aage N, Lazarov B. On multigrid-CG for efficient topology optimization. Structural and Multidisciplinary Optimization. 2014;48:815–829.

9. Svanberg K. The method of moving asymptotes—a new method for structural optimization. International Journal for Numerical Methods in Engineering. 1987;24(2):359–373.

10. Rik. findND. 2020. https://github.com/thrynae/findND.

11. CYY. vtkwrite. 2016. https://github.com/joe-of-all-trades/vtkwrite.

12. Kitware. ParaView, version 5.9.0. 2021. https://www.paraview.org/.

13. Guest JK, Prévost JH, Belytschko T. Achieving minimum length scale in topology optimization using nodal design variables and projection functions. International Journal for Numerical Methods in Engineering. 2004;61(2):238–254.

14. Guest JK. Topology optimization with multiple phase projection. Computer Methods in Applied Mechanics and Engineering. 2009;199(1):123–135.

15. Guest JK, Asadpoure A, Ha SH. Eliminating beta-continuation from heaviside projection and density filter algorithms. Structural and Multidisciplinary Optimization. 2011;44:443–453.

16. GrabCAD. GE jet engine bracket challenge. 2013. https://grabcad.com/challenges/ge-jet-engine-bracket-challenge.

17. Aitkenhead AH. Mesh voxelisation. 2013. https://www.mathworks.com/matlabcentral/fileexchange/27390-mesh-voxelisation.

18. Dassault Systemes. SOLIDWORKS. 2020.

19. Bejan A. Constructal-theory network of conducting paths for cooling a heat generating volume. International Journal of Heat and Mass Transfer. 1997;40(4):799–816.

20. Gersborg A, Bendsøe M, Sigmund O. Topology optimization of heat conduction problems using the finite volume method. Structural and Multidisciplinary Optimization. 2006;31:251–259.

21. Liu K, Tovar A. Heat conduction. top3d; 2014. https://www.top3d.app//tutorials/heat-conduction-top3d.

22. Yan S, Wang F, Sigmund O. On the non-optimality of tree structures for heat conduction. International Journal of Heat and Mass Transfer. 2018;122:660–680.

Appendix A. `templateTO.m`

```
%% TOPOLOGY OPTIMIZATION TEMPLATE ---------------------------------------%
% Created By:   Benjamin S. Rathman, Montana Technological University
% Program Name: templateTO.m
%% Program Description ---------------------------------------------------%
%   This program is designed to be utilized as a template for running a
%   topology optimization problem using topTheWorks.m
%% Initialize Program ----------------------------------------------------%
clear; clc;
fprintf('Program␣Running\n');
%% MATERIAL PROPERTIES ---------------------------------------------------%
scale = 1e-6;       % Material scaling factor;
E0 = 1.0;           % Young's modulus
Emin = E0*scale;    % Young's modulus of void
nu = 0.3;           % Poisson's ratio
sigmaY = 1.0;       % Yield strength
sigmaUTS = 1.0;     % Ultimate tensile strength
k0 = 1.0;           % Thermal conductivity
kmin = k0*scale;    % Thermal conductivity of void
mat = [E0,Emin,nu,sigmaY,sigmaUTS,k0,kmin];
%% PART GEOMETRY
stl = 0;            % Filename of the geometry being imported (if applicable)
xd = 100;           % x-dimension                                          % NOTE
    ↪ : If this program is going to be used
yd = 100;           % y-dimension                                          %
    ↪       to find a 2D solution, populate yd
zd = 100;           % z-dimension                                          %
    ↪       and zd, leaving xd at 0.
resol = 1;          % Part resolution (ele./unit)
nl = 3;             % Number of grid levels used in TO (MGCG)
loadCase = 0;       % Load condition
radj = 2^(nl-1)*round(resol*[xd,yd,zd]/2^(nl-1));                          %
    ↪ Convert dimensions to elements
[x,y,z] = deal(radj(1),radj(2),radj(3));
if xd == 0, x = 1; end
%% TOPOLOGY OPTIMIZATION
obj = 3;            % Objective function     1: Compliance
                    %                        2: Stress (unfinished)
                    %                        3: Heat Transfer
volfrac = 0.5;      % Prescribed volume fraction
penal = 3;          % Penalization factor
rmin0 = 3;          % Minimum length scale of void
rmin1 = 1.5;        % Minimum length scale of solid
ft = 2;             % Filtering scheme       1: Sensitivity filter
                    %                        2: Density filter
                    %                        3: Heaviside filter
                    %                        4: Heaviside void filter
                    %                        5: Multiphase Heaviside filter
maxit = 5e3;        % Maximum number of iterations
vtk = 0;            % Filename to export the solution geometry to
rmin = [rmin0*(ft == 4 || ft == 5) rmin1*~(ft == 4 && ft == 5)];
rmin(rmin == 0) = [];
%% PART LOADS & SUPPORTS
P = 1.0;            % Applied load
```

```
orient = [2,3,1];
nD = 2+double(x*y*z ~= y*z);                                                    %
    ↪ Number of dimensions
nodeNrs = int32(reshape(1:(x+1*(nD == 3))*(y+1)*(z+1),y+1,z+1,x+1*(nD == 3)));  %
    ↪ Nodes numbering
nDof = (y+1)*(z+1)*(x+1*(nD == 3))*(nD*(obj ~= 3)+1*(obj == 3));                %
    ↪ Total number of DOFs
% Supports Defined
fixed = [];                                                                     %
    ↪ Define support DOFs
% Loads Defined
lcDof = [];                                                                     %
    ↪ Define force DOFs
F = fsparse(lcDof',1,-P,[nDof,1]);                                              %
    ↪ Define force vector
% Assemble Passive Matrix
passive = [];                                                                   %
    ↪ Define passive matrix
%% RUN TOPOLOGY OPTIMIZATION
xPhys = topTheWorks(obj,mat,x,y,z,resol,volfrac,penal,rmin,ft,maxit,nl,...
                        passive,F,fixed);                                       % Run
                            ↪ topology optimization
xPhys = ipermute(xPhys,orient);
%% VTK FILE EXPORT ---------------------------------------------------------%
if vtk ~= 0
  vtkwrite([vtk,'.vtk'],'structured_points','TopOpt',xPhys);                    %
      ↪ Export solution geometry
end
%% Closing Program ---------------------------------------------------------%
fprintf('Program␣Finished\n');
```

Appendix B. `topTheWorks.m`

```
% TOPOLOGY OPTIMIZATION FOR COMPLEX DOMAINS ------------------------------%
function xPhysF = topTheWorks(obj,mat,nelx,nely,nelz,resol,volfrac,...
                              penal,rmin,ft,maxit,nl,passive,F,fixed)
% ------------------------------------------------------------------------%
% Created By:   Benjamin S. Rathman, Montana Technological University
% Program Name: topTheWorks.m
% ------------------------------------------------------------------------%
% Program Description
%   This function applies topology optimization to a part with a non-
%   standard design envelope.
% Variable Description
%   Return:
%       xPhysF      Final density distribution
%   Given:
%       obj         Objective function
%       mat         Material properties
%       nelx        Number of elements in the x-direction
%       nely        Number of elements in the y-direction
%       nelz        Number of elements in the z-direction
%       resol       Mesh density (ele./mm)
%       volfrac     Volume fraction within the design envelope
%       penal       Penalization factor
%       rmin        Minimum length scale (for 2-phase, rmin = [void solid])
%       ft          Filtering scheme
%       maxit       Maximum number of iterations
%       nl          Number of grid levels in MGCG
%       passive     Array defining the passive elements
%       F           Force vector
%       fixed       Support vector
%
% *Disclaimer*
%   The author reserves all rights but does not guarantee that the code is
%   free from errors. Furthermore, he shall not be liable in any event
%   caused by the use of the program.
% ------------------------------------------------------------------------%

% PRE.1) DISCRETIZATION FEATURES
nEl = nelx*nely*nelz;                                                   %
    ↪ Number of elements
nD = 2+double(nelx*nely*nelz ~= nely*nelz);                             %
    ↪ Number of dimensions
type = 1+double(nelx >= nelz);                                          % 2D
    ↪ stress assumption
nodeNrs = int32(reshape(1:(nely+1)*(nelz+1)*(nelx+1*(nD == 3)),...
                        nely+1,nelz+1,nelx+1*(nD == 3)));               %
                            ↪ Nodes numbering
[cMat,nDof,iK,jK] = objective.discrete(obj,nD,nodeNrs,nelx,nely,nelz);
[E0,nu,k0] = deal(mat(1),mat(3),mat(6));                                %
    ↪ Material properties
matrices = objMatrices(nD,resol,E0,nu,type,k0);                         %
    ↪ Assemble elemental matrices
% PRE.2) PREPARE FILTER
filtertype.error(ft,rmin);                                              % rmin
```

```
    ↪ configuration error
[h,Hs] = deal(cell(length(rmin),1));
for ff = 1:length(rmin)
  di = -ceil(rmin(ff))+1:ceil(rmin(ff))-1;
  if nD == 2
    [dy,dz] = meshgrid(di,di); dx = 0;
  elseif nD == 3
    [dy,dz,dx] = meshgrid(di,di,di);
  end
  hF = max(0,rmin(ff)-sqrt(dx.^2+dy.^2+dz.^2));
  HsF = imfilter(ones(nely,nelz,nelx),hF,'symmetric'); %
      ↪ Matrix of weights (filter)
  h{ff} = hF; Hs{ff} = HsF;
end
% PRE.3) INITIALIZE ITERATION
x = zeros(nEl*length(rmin),1);
act = setdiff((1:nEl)',~logical(passive)); lact = 1:length(act); %
    ↪ Define active elements
if ft == 5, act = [act;act+nEl]; end
x(act) = volfrac;
x(passive == 1) = 0; x(passive == 2) = 1; %
    ↪ Apply passive elements
xPhys = x(1:nEl);
[cK,dcK,dv0] = deal(zeros(nEl,1)); %
    ↪ Initialize vectors
U = zeros(nDof,size(F,2)); %
    ↪ Initialize displacement vector
dv0(act(lact)) = 1/nEl/volfrac; %
    ↪ Derivative of volume
[loop,beta,mu_max,change] = deal(0,0,1,1);
if ft ~= 1 && ft ~= 2, beta = 10; mu_max = 1; end %
    ↪ Initialize Heaviside parameters
% PRE.4) INITIALIZE MMA OPTIMIZER
m = 1;                    % The number of general constraints
n = length(act);          % The number of design variables x_j
xmin = zeros(n,1);        % Column vector with the lower bounds for the
    ↪ variables x_j
xmax = mu_max*ones(n,1);  % Column vector with the upper bounds for the
    ↪ variables x_j
xold1 = x(act);           % xval, one iteration ago (provided that iter>1)
xold2 = x(act);           % xval, two iterations ago (provided that iter>2)
low = ones(n,1);          % Column vector with the lower asymptotes from the
    ↪ previous iteration (provided that iter>1)
upp = ones(n,1);          % Column vector with the upper asymptotes from the
    ↪ previous iteration (provided that iter>1)
a0 = 1;                   % The constants a_0 in the term a_0*z
a = zeros(m,1);           % Column vector with the constants a_i in the terms
    ↪ a_i*z
c_MMA = 10000*ones(m,1);  % Column vector with the constants c_i in the terms
    ↪ c_i*y_i
d = zeros(m,1);           % Column vector with the constants d_i in the terms
    ↪ 0.5*d_i*(y_i)^2
% PRE.5) INITIALIZE MGCG SOLVER
Pu = cell(nl-1,1);
```

```
for mm = 1:nl-1
  Pu{mm,1} = multigrid.prepcoarse(obj,nD,nelz/2^(mm-1),nely/2^(mm-1),...
                                  nelx/2^(mm-1)); % Initialize multigrid cg
end
N = ones(nDof,1); N(fixed) = 0; Null = spdiags(N,0,nDof,nDof); % Null space elimination of supports
maxiter = 100; miniter = max(ceil(abs(2^(nl-2)-0.5-rmin/sqrt(2)))); % Maximum & minimum MGCG iterations
%% START ITERATION
while change > 1e-2 && loop < maxit
  loop = loop+1;
  % RL.1) SETUP AND SOLVE EQUILIBRIUM EQUATIONS
  K = cell(nl,1);
  sK = objective.prepfea(obj,nD,mat,matrices,xPhys,penal); % Prepare nodal stiffness matrix
  K{1,1} = fsparse(iK,jK,sK,[nDof,nDof]);
  K{1,1} = Null'*K{1,1}*Null-(Null-speye(nDof,nDof));
  for mm = 1:nl-1
    K{mm+1,1} = Pu{mm,1}'*(K{mm,1}*Pu{mm,1}); % Assemble nodal stiffness matrix
  end
  [Lfac,~] = chol(K{nl,1},'lower'); Ufac = Lfac';
  for kk = 1:size(U,2)
    [cgiters,~,U(:,kk)] = multigrid.mgcg(K,F(:,kk),U(:,kk),Lfac,Ufac,Pu,... % Assemble displacement vector
                                         nl,1,1e-6,maxiter,miniter); % for each load case
  end
  % RL.2) OBJECTIVE FUNCTION AND SENSITIVITY ANALYSIS
  [c,dc] = objective.objfunc(obj,mat,matrices,xPhys,act,lact,cK,dcK,...
                             penal,U,cMat); % Solve for the objective
  [dc,dv] = filtertype.scheme('modsens',ft,x,nelx,nely,nelz,beta,h,Hs,...
                              dc,dv0,mu_max); % Modify sensitivities
  % RL.3) METHOD OF MOVING ASYMPTOTES
  scale = 10^-floor(log(abs(c))./log(10)); % Scale objective for MMA
  xval = x(act);
  f0val = scale*c;
  df0dx = scale*dc(act);
  df0dx2 = 0*df0dx;
  fval = sum(xPhys(act(lact)))/(volfrac*n)-1-((1-volfrac)*(ft == 5));
  dfdx = dv(act)';
  dfdx2 = 0*dfdx;
  [xmma,~,~,~,~,~,~,~,~,low,upp] = mma.mmasub(m,n,loop,xval,xmin,xmax,...
  xold1,xold2,f0val,df0dx,df0dx2,fval,dfdx,dfdx2,low,upp,a0,a,c_MMA,d,beta);% Run MMA
  % Update MMA Variables
  [xnew,xPhys] = filtertype.scheme('updtMMA',ft,xmma,nelx,nely,nelz,...
                                   beta,h,Hs,xPhys,act,mu_max); % Update solution
  xPhys(passive == 1) = 0; xPhys(passive == 2) = 1; %
```

```
      ↪ Apply passive elements
129   xPhys = xPhys(:);
130   xold2 = xold1(:); xold1 = x(act);
131   change = max(abs(xnew(:)-x(:)));
132   x = xnew(:);
133   % RL.4) DISPLAY RESULTS
134   fprintf('  It.:%4i  Obj.:%6.3e  Vol.:%6.3f  ch.:%4.2e  cgIt.:%3i\n',...
135           loop,c,mean(xPhys(:)),change,cgiters);                    %
            ↪ Print iteration results
136   if nD == 2
137     imagesc(1-reshape(xPhys,nely,nelz)); colormap(gray); caxis([0 1]);
138     axis equal off; drawnow;                                        %
          ↪ Display iteration solution (2D)
139   end
140 end
141 xPhysF = reshape(xPhys,nely,nelz,nelx);
```

Appendix C. `objective.m`

```
classdef objective
  methods(Static)
    %-------------------------------------------------------------------------%
    % Created By:   Benjamin S. Rathman, Montana Technological University
    % Program Name: objective.m
    %-------------------------------------------------------------------------%
    % Program Description
    %   This function applies the chosen objective function equations to
    %   the optimization process.
    %-------------------------------------------------------------------------%

    % DISCRETIZATION FEATURES
    function [cMat,nDof,iK,jK] = discrete(obj,nD,nodeNrs,nelx,nely,nelz)
      nEl = nelx*nely*nelz;
      if obj == 1 || obj == 2 % Structural compliance or Stress
        cVec = reshape(nD*nodeNrs(1:nely,1:nelz,1:nelx)+1,nEl,1);
        if nD == 2
          cMat = cVec+int32([0,1,2*nely+[2,3,0,1],-2,-1]);
        elseif nD == 3
          cMat = cVec+int32([0,1,2,3*(nely+1)*(nelz+1)
              ↪ +[0,1,2,-3,-2,-1],-3,-2,-1,...
                3*(nely+1)+[0,1,2],3*(nely+1)*(nelz+2)+[0,1,2,-3,-2,-1],...
                3*(nely+1)+[-3,-2,-1]]);
        end
        nDof = (nely+1)*(nelz+1)*(nelx+1*(nD == 3))*nD;                    %
            ↪ Total number of DOFs
        iK = reshape(kron(cMat(1:nEl,1:(16*nD-24)),ones((16*nD-24),1,'int32'))
            ↪ ',(16*nD-24)^2*nEl,1);
        jK = reshape(kron(cMat(1:nEl,1:(16*nD-24)),ones(1,(16*nD-24),'int32'))
            ↪ ',(16*nD-24)^2*nEl,1);
      elseif obj == 3   % Thermal compliance
        cVec = reshape(nodeNrs(1:nely,1:nelz,1:nelx)+1,nEl,1);
        if nD == 2
          cMat = cVec+int32([0,nely+[1,0],-1]);
        elseif nD == 3
          cMat = cVec+int32([0,nely+[1,0],-1,(nely+1)*(nelx+1)+[0,nely+[1,0],-1]])
              ↪ ;
        end
        nDof = (nely+1)*(nelz+1)*(nelx+1*(nD == 3));
        iK = reshape(kron(cMat(1:nEl,1:(2^nD)),ones((2^nD),1,'int32'))',(2^nD)^2*
            ↪ nEl,1);
        jK = reshape(kron(cMat(1:nEl,1:(2^nD)),ones(1,(2^nD),'int32'))',(2^nD)^2*
            ↪ nEl,1);
      end
    end

    % SETUP FINITE ELEMENT ANALYSIS EQUATION
    function sK = prepfea(obj,nD,mat,matrices,xPhys,penal)
      nEl = length(xPhys(:));
      if obj == 1       % Structural compliance
        [E0,Emin] = deal(mat(1),mat(2));
        Ke = matrices{1};
        sK = reshape(Ke(:)*(Emin/E0+xPhys(:)'.^penal*(1-Emin/E0)),(16*nD-24)^2*nEl
```

```
            ↪ ,1);
47        elseif obj == 2   % Stress
48
49        elseif obj == 3   % Thermal compliance
50          [k0,kmin] = deal(mat(6),mat(7));
51          K = matrices{4};
52          sK = reshape(K(:)*(kmin/k0+xPhys(:)'.^penal*(1-kmin/k0)),(2^nD)^2*nEl,1);
53        end
54      end
55
56      % OBJECTIVE FUNCTION
57      function [c,dc] = objfunc(obj,mat,matrices,xPhys,act,lact,cK,dcK,penal,U,cMat)
58        nEl = length(xPhys(:));
59        if obj == 1       % Structural compliance
60          [E0,Emin] = deal(mat(1),mat(2));
61          Ke = matrices{1};
62          c = 0; dc = zeros(nEl,1);
63          cK(act(lact)) = Emin/E0+xPhys(act(lact)).^penal*(1-Emin/E0);
64          dcK(act(lact)) = -penal*(1-Emin/E0)*xPhys(act(lact)).^(penal-1);
65          for ii = 1:size(U,2)
66            Ui = U(:,ii);
67            c = c+sum(sum(cK.*sum(Ui(cMat)*Ke.*Ui(cMat),2)));
68            dc = dc+dcK.*sum(Ui(cMat)*Ke.*Ui(cMat),2);
69          end
70        elseif obj == 2   % Stress
71
72        elseif obj == 3   % Thermal Compliance
73          [k0,kmin] = deal(mat(6),mat(7));
74          K = matrices{4};
75          cK(act(lact)) = kmin/k0+xPhys(:)'.^penal*(1-kmin/k0);
76          dcK(act(lact)) = -penal*(1-kmin/k0)*xPhys(act(lact)).^(penal-1);
77          c = sum(sum(cK.*sum(U(cMat)*K.*U(cMat),2)));
78          dc = dcK.*sum(U(cMat)*K.*U(cMat),2);
79        end
80      end
81
82    end
83  end
```

Appendix D. `objMatrices.m`

```
function matrices = objMatrices(nD,resol,E,nu,type,k)
% ------------------------------------------------------------------------%
% Created By:   Benjamin S. Rathman, Montana Technological University
% Program Name: objMatrices.m
% ------------------------------------------------------------------------%
% Program Description
%   This function outputs property matrices given material properties.
% Variable Description
%   Return:
%       matrices    material matrices
%       |-Ke        Stiffness matrix
%       |-C         Constitutive matrix
%       |-B         Strain-displacement matrix
%       |-K         Heat Conduction matrix
%   Given:
%       nD          Number of dimensions
%       resol       Mesh density (ele./mm)
%       E           Young's Modulus
%       nu          Poisson's Ratio
%       type        2D assumption (plane stress = 1, plane strain = 2)
%       k           Thermal Conductivity
% ------------------------------------------------------------------------%

if nD == 2      % For 2D FEA
  % Constitutive Matrix
  if type == 1      % Plane Stress
    C = E/(1-nu^2)*[ 1 nu   0;
                    nu  1   0;
                     0  0 (1-nu)/2];
  elseif type == 2  % Plane Strain
    C = E/(1-2*nu)*[     1      nu/(1-nu) 0;
                    nu/(1-nu)      1      0;
                         0         0 (1-2*nu)/(2*(1-nu))];
  end
  % Strain-displacement Matrix
  [a,b] = deal(1/(2*resol)); h = 1;
  syms Ba(xi,eta)
  dNdx = (1/a)*[(eta-1),(1-eta),(1+eta),-(1+eta)];
  dNdy = (1/b)*[(xi-1),-(1+xi),(1+xi),(1-xi)];
  for ii = 1:4
    Bsym(:,:,ii) = [dNdx(ii)     0     ;
                        0      dNdy(ii);
                    dNdy(ii) dNdx(ii)];
  end
  Ba(xi,eta) = (1/4)*reshape(Bsym,3,8);
  B = double(Ba(0,0));
  % Stiffness Matrix
  Ke = double(a*b*h*int(int(Ba'*C*Ba,xi,-1,1),eta,-1,1));
  % Heat Conduction Matrix
  K = 1/6*k*[ 4 -1 -2 -1;
             -1  4 -1 -2;
             -2 -1  4 -1;
             -1 -2 -1  4];
```

```
54
55 elseif nD == 3  % For 3D FEA
56   % Constitutive Matrix
57   C = E/((1+nu)*(1-2*nu))*[1-nu   nu    nu   0 0 0;
58                            nu   1-nu   nu   0 0 0;
59                            nu    nu   1-nu  0 0 0;
60                           0 0 0 (1-2*nu)/2  0 0;
61                           0 0 0 0 (1-2*nu)/2  0;
62                           0 0 0 0 0 (1-2*nu)/2];
63   % Strain-displacement Matrix
64   [a,b,c] = deal(1/(2*resol));
65   syms Ba(xi,eta,zeta)
66   dNdx = (1/a)*[-(eta-1)*(zeta-1),(eta-1)*(zeta-1),-(1+eta)*(zeta-1),(1+eta)*(zeta
       ↪ -1),(eta-1)*(1+zeta),-(eta-1)*(1+zeta),(1+eta)*(1+zeta),-(1+eta)*(1+zeta)
       ↪ ];
67   dNdy = (1/b)*[-(xi-1)*(zeta-1),(1+xi)*(zeta-1),-(1+xi)*(zeta-1),(xi-1)*(zeta-1)
       ↪ ,(xi-1)*(1+zeta),-(1+xi)*(1+zeta),(1+xi)*(1+zeta),-(xi-1)*(1+zeta)];
68   dNdz = (1/c)*[-(xi-1)*(eta-1),(1+xi)*(eta-1),-(1+xi)*(1+eta),(xi-1)*(1+eta),(xi
       ↪ -1)*(eta-1),-(1+xi)*(eta-1),(1+xi)*(1+eta),-(xi-1)*(1+eta)];
69   for ii = 1:8
70     Bsym(:,:,ii) = [dNdx(ii)     0          0     ;
71                          0     dNdy(ii)     0     ;
72                          0        0      dNdz(ii);
73                          0     dNdz(ii) dNdy(ii);
74                     dNdz(ii)     0      dNdx(ii);
75                     dNdy(ii) dNdx(ii)     0     ];
76   end
77   Ba(xi,eta,zeta) = (1/8)*reshape(Bsym,6,24);
78   B = double(Ba(0,0,0));
79   % Stiffness Matrix
80   Ke = double(a*b*c*int(int(int(Ba'*C*Ba,xi,-1,1),eta,-1,1),zeta,-1,1));
81   % Heat Conduction Matrix
82   A1  = 4*eye(2);   A2 = -eye(2);
83   A3  = fliplr(A2); A4 = -ones(2);
84   Ke1 = [A1 A2;A2 A1];
85   Ke2 = [A3 A4;A4 A3];
86   K   = 1/12*k*[Ke1 Ke2;Ke2 Ke1];
87 end
88 matrices = {Ke,C,B,K};
```

Appendix E. `filtertype.m`

```
classdef filtertype
  methods(Static)
    %-------------------------------------------------------------------------%
    % Created By:  Benjamin S. Rathman, Montana Technological University
    % Program Name: filtertype.m
    %-------------------------------------------------------------------------%
    % Program Description
    %   This function applies the chosen filtering scheme to the
    %   optimization process.
    %-------------------------------------------------------------------------%

    % FILTER VS RMIN MISMATCH ERROR
    function error(ft,rmin)
      if (ft == 1 || ft == 2 || ft == 3 || ft == 4) && length(rmin) ~= 1
        error('Variable rmin must be a scalar for this filter type.');
      elseif ft == 5 && length(rmin) ~= 2
        error(['Variable rmin must be a vector of the form ',...
               '[rmin_void rmin_solid] for this filter type.']);
      end
    end

    % FILTERING METHOD
    function varargout = scheme(type,ft,x,nelx,nely,nelz,beta,h,Hs,varargin)
      nEl = nelx*nely*nelz;
      % Define Filter
      if ft == 1 || ft == 2 || ft == 3 || ft == 4
        h = h{1}; Hs = Hs{1};
      elseif ft == 5
        [h0,h1] = deal(h{1},h{2}); [Hs0,Hs1] = deal(Hs{1},Hs{2});
      end
      % Extract Variables from 'varargin'
      switch type
        case 'modsens' % define derivatives
          [dc,dv,mu_max] = deal(varargin{1},varargin{2},varargin{3});
        case 'updtMMA' % define active elements
          [xPhysOld,act,mu_max] = deal(varargin{1},varargin{2},varargin{3});
          if ft == 5
            xPhysOld = repmat(xPhysOld,2,1);
          end
          xPhysOld(act) = x;
      end
      % Apply Filter
      if ft == 1 % sensitivity filter
        switch type
          case 'modsens'
            dc = imfilter(reshape(x.*dc,nely,nelz,nelx),h,'symmetric')./...
                 Hs./reshape(max(1e-3,x),nely,nelz,nelx);
          case 'updtMMA'
            xnew = reshape(xPhysOld,nely,nelz,nelx);
            xPhys = xnew;
        end
      elseif ft == 2 % density filter
        switch type
```

```
            case 'modsens'
              dc = imfilter(reshape(dc,nely,nelz,nelx),h,'symmetric')./Hs;
              dv = imfilter(reshape(dv,nely,nelz,nelx),h,'symmetric')./Hs;
            case 'updtMMA'
              xnew = reshape(xPhysOld,nely,nelz,nelx);
              xPhys = imfilter(xnew./Hs,h,'symmetric');
          end
        elseif ft == 3 % heaviside solid projection
          switch type
            case 'modsens'
              mu = imfilter(reshape(x,nely,nelz,nelx)./Hs,h,'symmetric');
              dx = reshape(beta*exp(-beta*mu)+exp(-beta*mu_max)/mu_max,nEl,1);
              dc = imfilter(reshape(dc.*dx,nely,nelz,nelx),h,'symmetric')./Hs;
              dv = imfilter(reshape(dv.*dx,nely,nelz,nelx),h,'symmetric')./Hs;
            case 'updtMMA'
              xnew = reshape(xPhysOld,nely,nelz,nelx);
              mu = imfilter(xnew./Hs,h,'symmetric');
              xPhys = 1-exp(-beta*mu)+(mu/mu_max)*exp(-beta*mu_max);
          end
        elseif ft == 4 % heaviside void projection
          switch type
            case 'modsens'
              mu = imfilter(reshape(x,nely,nelz,nelx)./Hs,h,'symmetric');
              dx = reshape(-beta*exp(-beta*mu)-exp(-beta*mu_max)/mu_max,nEl,1);
              dc = imfilter(reshape(dc.*dx,nely,nelz,nelx),h,'symmetric')./Hs;
              dv = imfilter(reshape(dv.*dx,nely,nelz,nelx),h,'symmetric')./Hs;
            case 'updtMMA'
              xnew = reshape(xPhysOld,nely,nelz,nelx);
              mu = imfilter(xnew./Hs,h,'symmetric');
              xPhys = exp(-beta*mu)-(mu/mu_max)*exp(-beta*mu_max);
          end
        elseif ft == 5 % heaviside multi-phase projection
          switch type
            case 'modsens'
              mu0 = imfilter(reshape(x(1:nEl),nely,nelz,nelx)./Hs0,h0,'symmetric');
              dx0 = reshape((-beta*exp(-beta*mu0)-exp(-beta*mu_max)/mu_max)/2,nEl,1)
                  ↪ ;
              dc0 = imfilter(reshape(dc.*dx0,nely,nelz,nelx),h0,'symmetric')./Hs0;
              dv0 = imfilter(reshape(dv.*dx0,nely,nelz,nelx),h0,'symmetric')./Hs0;
              mu1 = imfilter(reshape(x(nEl+1:end),nely,nelz,nelx)./Hs1,h1,'symmetric
                  ↪ ');
              dx1 = reshape((beta*exp(-beta*mu1)+exp(-beta*mu_max)/mu_max)/2,nEl,1);
              dc1 = imfilter(reshape(dc.*dx1,nely,nelz,nelx),h1,'symmetric')./Hs1;
              dv1 = imfilter(reshape(dv.*dx1,nely,nelz,nelx),h1,'symmetric')./Hs1;
              dc = [dc0 dc1]; dv = [dv0 dv1];
            case 'updtMMA'
              xnew0 = reshape(xPhysOld(1:nEl),nely,nelz,nelx);
              mu0 = imfilter(xnew0./Hs0,h0,'symmetric');
              xPhys0 = exp(-beta*mu0)-(mu0/mu_max)*exp(-beta*mu_max);
              xnew1 = reshape(xPhysOld(nEl+1:end),nely,nelz,nelx);
              mu1 = imfilter(xnew1./Hs1,h1,'symmetric');
              xPhys1 = 1-exp(-beta*mu1)+(mu1/mu_max)*exp(-beta*mu_max);
              xnew = [xnew0 xnew1];
              xPhys = (xPhys0+xPhys1)/2;
```

```
106           end
107         end
108         % Insert Variables into 'varargout'
109         switch type
110           case 'modsens'
111             [varargout{1},varargout{2}] = deal(dc,dv);
112           case 'updtMMA'
113             [varargout{1},varargout{2}] = deal(xnew,xPhys);
114         end
115       end
116
117     end
118   end
```

Appendix F. `multigrid.m`

```
classdef multigrid
  methods(Static)

    % PREPARE MG PROLONGATION OPERATOR
    function [Pu] = prepcoarse(obj,nD,nex,ney,nez)
      % Assemble state variable prolongation
      maxnum = nex*ney*nez*20;
      iP = zeros(maxnum,1); jP = zeros(maxnum,1); sP = zeros(maxnum,1);
      nexc = nex/2; neyc = ney/2; nezc = nez/2;
      % Weights for fixed distances to neighbors on a structured grid
      if nD == 2
        vals = [1,0.5,0.25];
      elseif nD == 3
        vals = [1,0.5,0.25,0.125];
      end
      cc = 0;
      for nx = 1:nexc+1
        for ny = 1:neyc+1
          for nz = 1:nezc+1
            if nD == 2
              col = (nx-1)*(neyc+1)+ny;
            elseif nD == 3
              col = (nz-1)*((nexc+1)*(neyc+1))+(nx-1)*(neyc+1)+ny;
            end
            % Coordinate on fine grid
            nx1 = nx*2-1; ny1 = ny*2-1; nz1 = nz*2-1;
            % Loop over fine nodes within the rectangular domain
            for ii = max(nx1-1,1):min(nx1+1,nex+1)
              for jj = max(ny1-1,1):min(ny1+1,ney+1)
                for kk = max(nz1-1,1):min(nz1+1,nez+1)
                  if nD == 2
                    row = (ii-1)*(ney+1)+jj;
                    % Based on squared dist assign weights: 1.0 0.5 0.25
                    ind = 1+((nx1-ii)^2+(ny1-jj)^2);
                    if obj == 1 || obj == 2
                      cc = cc+1; iP(cc) = 2*row-1; jP(cc) = 2*col-1; sP(cc) = vals
                          ↪ (ind);
                      cc = cc+1; iP(cc) = 2*row;   jP(cc) = 2*col;   sP(cc) = vals
                          ↪ (ind);
                    elseif obj == 3
                      cc = cc+1; iP(cc) = row; jP(cc) = col; sP(cc) = vals(ind);
                    end
                  elseif nD == 3
                    row = (kk-1)*((nex+1)*(ney+1))+(ii-1)*(ney+1)+jj;
                    % Based on squared dist assign weights: 1.0 0.5 0.25 0.125
                    ind = 1+((nx1-ii)^2+(ny1-jj)^2+(nz1-kk)^2);
                    if obj == 1 || obj == 2
                      cc = cc+1; iP(cc) = 3*row-2; jP(cc) = 3*col-2; sP(cc) = vals
                          ↪ (ind);
                      cc = cc+1; iP(cc) = 3*row-1; jP(cc) = 3*col-1; sP(cc) = vals
                          ↪ (ind);
                      cc = cc+1; iP(cc) = 3*row;   jP(cc) = 3*col;   sP(cc) = vals
                          ↪ (ind);
```

```
                    elseif obj == 3
                      cc = cc+1; iP(cc) = row;    jP(cc) = col;    sP(cc) = vals(ind
                          ↪ );
                    end
                  end
                end
              end
            end
          end
        end
      end
      % Assemble matrices
      Pu = fsparse(iP(1:cc),jP(1:cc),sP(1:cc));
    end

    % MULTIGRID PRECONDITIONED CONJUGATE GRADIENTS
    function [i,relres,u] = mgcg(A,b,u,Lfac,Ufac,Pu,nl,nswp,tol,maxiter,miniter)
      r = b-A{1,1}*u;
      res0 = norm(b);
      % Jacobi smoother
      omega = 0.6;
      invD = cell(nl-1,1);
      for l = 1:nl-1
        invD{l,1}= 1./spdiags(A{l,1},0);
      end
      for i = 1:maxiter
        z = multigrid.VCycle(A,r,Lfac,Ufac,Pu,1,nl,invD,omega,nswp);
        rho = r'*z;
        if i == 1
          p = z;
        else
          beta = rho/rho_p;
          p = beta*p + z;
        end
        q = A{1,1}*p;
        dpr = p'*q;
        alpha = rho/dpr;
        u = u+alpha*p;
        r = r-alpha*q;
        rho_p = rho;
        relres = norm(r)/res0;
        if relres < tol && i >= miniter
          break
        end
      end
    end
    % COARSE GRID CORRECTION
    function [z] = VCycle(A,r,Lfac,Ufac,Pu,l,nl,invD,omega,nswp)
      z = 0*r;
      z = multigrid.smthdmpjac(z,A{l,1},r,invD{l,1},omega,nswp);
      Az = A{l,1}*z;
      d = r - Az;
      dh2 = Pu{l,1}'*d;
      if (nl == l+1)
```

```
102           vh2 = Ufac\(Lfac\dh2);
103         else
104           vh2 = multigrid.VCycle(A,dh2,Lfac,Ufac,Pu,l+1,nl,invD,omega,nswp);
105         end
106         v = Pu{l,1}*vh2;
107         z = z+v;
108         z = multigrid.smthdmpjac(z,A{l,1},r,invD{l,1},omega,nswp);
109       end
110       % DAMPED JACOBI SMOOTHER
111       function [u] = smthdmpjac(u,A,b,invD,omega,nswp)
112         for i = 1:nswp
113           u = u-omega*invD.*(A*u)+omega*invD.*b;
114         end
115       end
116
117     end
118   end
```

Appendix G. GE Bracket Problem Setup

```
%% TOPOLOGY OPTIMIZATION TEMPLATE ----------------------------------------%
% Created By:    Benjamin S. Rathman, Montana Technological University
% Program Name: GEbracket.m
% Created On:    03/23/2021
% Rev:--Modified By:----------Modified Date:----Notes:--------------------%
% r00    Benjamin S. Rathman    03/23/2021          Created template
% r01    Benjamin S. Rathman    04/02/2021          GE bracket problem
%% Program Description ---------------------------------------------------%
%   This program is designed for GE bracket TO
%% Initialize Program ----------------------------------------------------%
clear; clc;
fprintf('Program␣Running\n');
%% MATERIAL PROPERTIES ---------------------------------------------------%
scale = 1e-6;         % Material scaling factor;
E0 = 16.51e6;         % Young's modulus
Emin = E0*scale;      % Young's modulus of void
nu = 0.34;            % Poisson's ratio
sigmaY = 128e3;       % Yield strength
sigmaUTS = 138e3;     % Ultimate tensile strength
k0 = 46.5;            % Thermal conductivity
kmin = k0*scale;      % Thermal conductivity of void
mat = [E0,Emin,nu,sigmaY,sigmaUTS,k0,kmin];
%% PART GEOMETRY
stl = 'original';     % Filename of the geometry being imported
xd = 7.029;           % x-dimension
yd = 2.461;           % y-dimension
zd = 4.262;           % z-dimension
resol = 15;           % Part resolution (ele./unit)
nl = 3;               % Number of grid levels used in TO (MGCG)
loadCase = 0;         % Load condition
radj = 2^(nl-1)*round(resol*[xd,yd,zd]/2^(nl-1));                        %
    ↪ Convert dimensions to elements
[x,y,z] = deal(radj(1),radj(2),radj(3));
if xd == 0, x = 1; end
stldomain = VOXELISE(x,y,z,[stl,'.stl']);
domain = double(~stldomain);
domainVol = 1-mean(domain(:));
%% TOPOLOGY OPTIMIZATION
obj = 1;              % Objective function
volfrac = 0.25;       % Prescribed volume fraction
penal = 3;            % Penalization factor
rmin0 = 3;            % Minimum length scale of void
rmin1 = 1.5;          % Minimum length scale of solid
ft = 2;               % Filtering scheme
maxit = 50;           % Maximum number of iterations
vtk = 'GEbracket';  % Filename to export the solution geometry to
rmin = [rmin0*(ft == 4 || ft == 5) rmin1*~(ft == 4 && ft == 5)];
rmin(rmin == 0) = [];
%% PART LOADS & SUPPORTS
P = 1.0;              % Applied load
orient = [2,3,1];
nD = 2+double(x*y*z ~= y*z);                                             %
    ↪ Number of dimensions
```

```
nodeNrs = int32(reshape(1:(x+1*(nD == 3))*(y+1)*(z+1),y+1,z+1,x+1*(nD == 3))); %
    ↪ Nodes numbering
nDof = (y+1)*(z+1)*(x+1*(nD == 3))*(nD*(obj ~= 3)+1*(obj == 3)); %
    ↪ Total number of DOFs
% Supports Defined
  % Support Elements Defined
  s1x = 0.600/xd;      s1z = 0.602/zd; %
      ↪ Position of support hole 1
  s2x = s1x;           s2z = 2.652/zd; %
      ↪ Position of support hole 2
  s3x = 6.429/xd;      s3z = 1.329/zd; %
      ↪ Position of support hole 3
  s4x = s3x;           s4z = 2.829/zd; %
      ↪ Position of support hole 4
  sh = [s1x,s2x,s3x,s4x,s1z,s2z,s3z,s4z];
  sRad = (0.406/xd)/2; %
      ↪ Radius of support holes
  cRad = (0.830/xd)/2; %
      ↪ Radius of counterbores
  domainS = permute(domain,[3,1,2]);
  for layerS = 1:y(end)
    domainLayer = domainS(:,:,layerS);
    if domainLayer(round(z/10),round(x/10)) == 0 %
        ↪ Define edge of hole as active elements
      for elzS = 1:z
        for elxS = 1:x
          for ii = 1:4 % For
              ↪ each support hole position
            sx = sh(ii); sz = sh(ii+4);
            if (round(sqrt((elxS-x*sx)^2+(elzS-z*sz)^2)) >= round(x*sRad+1) &&...
                round(sqrt((elxS-x*sx)^2+(elzS-z*sz)^2)) <= round(x*cRad+1))
              domainLayer(elzS,elxS) = 3; %
                  ↪ Support ele. denoted as "3"
            elseif round(sqrt((elxS-x*sx)^2+(elzS-z*sz)^2)) < round(x*sRad+1)
              domainLayer(elzS,elxS) = 1; %
                  ↪ Clear support holes of stray elements
            end
          end
        end
      end
    end
    domainS(:,:,layerS) = domainLayer;
  end
  domainS = ipermute(domainS,[3,1,2]);
  domain(domainS == 3) = 2; %
      ↪ Assign passive solid to support elements
  domainS = permute(domainS,orient);
  % Assemble Support Vector
  [iif,jf,kf] = findND(domainS == 3); %
      ↪ Coordinates
  fixed = zeros(length(iif),1,'int32');
  for ii = 1:length(iif)
    fixed(ii) = 3*nodeNrs(iif(ii),jf(ii),kf(ii));
  end
```

```
92    fixed = [fixed(:); fixed(:)-1; fixed(:)-2];                    %
          ↪ Support(Fixed) DOFs
93  % Assemble Load Vector
94    % Load Elements Defined
95    ly = 1.761/yd;       lz = 3.560/zd;                            %
          ↪ Position of load hole
96    lRad = (0.753/xd)/2;                                           %
          ↪ Radius of load hole
97    domainL = permute(domain,[2,3,1]);
98    for layerL = 1:x/2
99      domainLayer = domainL(:,:,layerL);
100     if domainLayer(end,round(z*5/6)) == 0                        %
            ↪ Define edge of hole as active elements
101       for elzL = 1:z
102         for elyL = 1:y
103           if round(sqrt((elyL-y*ly)^2+(elzL-z*lz)^2)) == round(x*lRad+1)
104             domainLayer(elyL,elzL) = -4;                         % Load
                    ↪ ele. denoted as "-4"
105           elseif round(sqrt((elyL-y*ly)^2+(elzL-z*lz)^2)) < round(x*lRad+1)
106             domainLayer(elyL,elzL) = 1;                          %
                    ↪ Clear load holes of stray elements
107           end
108         end
109       end
110     end
111     domainL(:,:,layerL) = domainLayer;
112   end
113   for layerL = x/2:x
114     domainLayer = domainL(:,:,layerL);
115     if domainLayer(end,round(z*5/6)) == 0                        %
            ↪ Define edge of hole as active elements
116       for elzL = 1:z
117         for elyL = 1:y
118           if round(sqrt((elyL-y*ly)^2+(elzL-z*lz)^2)) == round(x*lRad+1)
119             domainLayer(elyL,elzL) = 4;                          % Load
                    ↪ ele. denoted as "4"
120           elseif round(sqrt((elyL-y*ly)^2+(elzL-z*lz)^2)) < round(x*lRad+1)
121             domainLayer(elyL,elzL) = 1;                          %
                    ↪ Clear load holes of stray elements
122           end
123         end
124       end
125     end
126     domainL(:,:,layerL) = domainLayer;
127   end
128   domainL = ipermute(domainL,[2,3,1]);                           %
          ↪ Array defining loads and supports
129   domain(domainL == -4 | domainL == 4) = 2;                      %
          ↪ Assign passive solid to support elements
130   domainL = permute(domainL,orient);
131   % Define Loads
132   if loadCase == 1 || loadCase == 2 || loadCase == 3 || loadCase == 0
133     domainL(domainL == -4) = 4;
134     [il_L,jl,kl] = findND(domainL == 4);                         %
```

```
        ↪ Coordinates
    lcDof = zeros(length(il_L),1,'int32');
    for ii = 1:length(il_L)
      lcDof(ii) = 3*nodeNrs(il_L(ii),jl(ii),kl(ii));
    end
    lcDof_L = lcDof(:)-1;                                          % Load
        ↪ DOFs
  end
  if loadCase == 4 || loadCase == 0
    [il_Tn,jl,kl] = findND(domainL == -4);                         %
        ↪ Coordinates
    lcDof = zeros(length(il_Tn),1,'int32');
    for ii = 1:length(il_Tn)
      lcDof(ii) = 3*nodeNrs(il_Tn(ii),jl(ii),kl(ii));
    end
    lcDof_Tn = lcDof(:)-1;                                         % Load
        ↪ DOFs
    [il_Tp,jl,kl] = findND(domainL == 4);                          %
        ↪ Coordinates
    lcDof = zeros(length(il_Tp),1,'int32');
    for ii = 1:length(il_Tp)
      lcDof(ii) = 3*nodeNrs(il_Tp(ii),jl(ii),kl(ii));
    end
    lcDof_Tp = lcDof(:)-1;                                         % Load
        ↪ DOFs
  end
  % Assemble Load Vector                                          % x =
      ↪ -1, y = 0, z = +1
  if loadCase == 1
    F = fsparse(lcDof_L,1,8000/length(il_L),[nDof,1]);
  elseif loadCase == 2
    F = fsparse(lcDof_L+1,1,8500/length(il_L),[nDof,1]);
  elseif loadCase == 3
    F = fsparse(lcDof_L,1,9500*cosd(42)/length(il_L),[nDof,1])+...
        fsparse(lcDof_L+1,1,9500*sind(42)/length(il_L),[nDof,1]);
  elseif loadCase == 4
    F = fsparse(lcDof_Tn+1,1,(-5000/0.563)/length(il_Tn),[nDof,1])+...
        fsparse(lcDof_Tp+1,1,(5000/0.563)/length(il_Tp),[nDof,1]);
  elseif loadCase == 0
    F1 = fsparse(lcDof_L,1,8000/length(il_L),[nDof,1]);
    F2 = fsparse(lcDof_L+1,1,8500/length(il_L),[nDof,1]);
    F3 = fsparse(lcDof_L,1,9500*cosd(42)/length(il_L),[nDof,1])+...
         fsparse(lcDof_L+1,1,9500*sind(42)/length(il_L),[nDof,1]);
    F4 = fsparse(lcDof_Tn+1,1,(-5000/0.563)/length(il_Tn),[nDof,1])+...
         fsparse(lcDof_Tp+1,1,(5000/0.563)/length(il_Tp),[nDof,1]);
    F = cat(2,F1,F2,F3,F4);
  end
% Assemble Passive Matrix
passive = permute(domain,orient);                                  %
    ↪ Define passive matrix
%% RUN TOPOLOGY OPTIMIZATION
xPhys = topTheWorks(obj,mat,x,y,z,resol,volfrac*domainVol,penal,rmin,ft,...
                    maxit,nl,passive,F,fixed);                     % Run
                        ↪ topology optimization
```

```
180 xPhys = ipermute(xPhys,orient);
181 %% VTK FILE EXPORT ----------------------------------------------------%
182 if vtk ~= 0
183   vtkwrite([vtk,'.vtk'],'structured_points','TopOpt',xPhys);                %
        ↪ Export solution geometry
184 end
185 %% Closing Program ----------------------------------------------------%
186 fprintf('Program␣Finished\n');
```

List of Symbols, Abbreviations, and Acronyms

2-D	two-dimensional
3-D	three-dimensional
AM	additive manufacturing
DoF	degree of freedom
FEA	finite element analysis
MGCG	multigrid preconditioned conjugate gradients
MMA	method of moving asymptotes
OC	Optimality Criteria
RAM	random access memory
SIMP	solid isotropic material with penalization
TO	topology optimization

1 (PDF) DEFENSE TECHNICAL INFORMATION CTR DTIC OCA

1 (PDF) DEVCOM ARL
FCDD RLD DCI
TECH LIB

71 (PDF) DIR DEVCOM ARL
FCDD RLD S
H MAUPIN
M TSCHOPP
FCDD RLS E
W BENARD
FCDD RLW
J NEWILL
S KARNA
A RAWLETT
S SCHOENFELD
FCDD RLW B
C HOPPEL
R BECKER
L VARGAS-GONZALAS
A TONGE
J CAMPBELL
FCDD RLW C
D LYON
FCDD RLW D
J ROBINETTE
B MCWILLIAMS
E FORSYTH
S ISERT
FCDD RLW L
T SHEPPARD
FCDD RLW LD
R BEYER
M NUSCA
P CONROY
J SCHMIDT
Z WINGARD
FCDD RLW LE
J DESPIRITO
FCDD RLW LF
M ILG
R HALL
B NELSON
FCDD RLW LH
E KENNEDY
M MINNICINO
J O'GRADY
R SUMMERS
FCDD RLW M
E CHIN
J LASCALA
FCDD RLW MA
J SANDS
T PLAISTED
FCDD RLW MB
B LOVE
D O'BRIEN
C FOUNTZOULAS

A GAYNOR
G GAZONAS
E HERNANDEZ
Z WILSON
FCDD RLW MD
K CHO
M PEPI
A KUDZAL
J BROWN
F KELLOGG
J TAGGART-SCARFF
R ROGERS
V CHAMPAGNE
C MOCK
I NAULT
D NIKOLOV
A NARDI
M GAMMAGE
J YU
B CHEESEMAN
W ROY
S CLUFF
D VANOOSTEN
FCDD RLW ME
S SILTON
FCDD RLW MF
K DOHERTY
B BUTLER
V HAMMOND
M DUNSTAN
P GOINS
FCDD RLW MG
J LENHART
FCDD RLW PA
S BILYK
J CLAYTON
J LLOYD
C MEREDITH