

# APMA2560 FINAL PROJECT: TOPOLOGY OPTIMIZATION

MATTHEW MEEKER

## 1. INTRODUCTION

Briefly, topology optimization offers a class of techniques and algorithms for discovering the optimal distribution of material within a given domain, subject to some set of physics and other constraints, which tend to form PDE-constrained optimization problems. In general, these problems take the form

$$(1) \quad \begin{aligned} \min_{\rho} \quad & F := \int_{\Omega} f(u(\rho), \rho) dV \\ \text{s.t.} \quad & \int_{\Omega} \rho dV \leq \theta V(\Omega), \\ & G_j \leq 0, \end{aligned}$$

where  $\Omega$  is the design domain,  $\rho \in L^2(\Omega)$  is a function describing the material distribution,  $f$  forms the objective function,  $\theta$  is the *mass fraction* (that is, the fraction of  $\Omega$  which may be occupied by material; this may be made physical by considering material cost constraints, etc.), and  $G_j$  describes a related set of constraints. One typically uses a finite element method to assess the design and a gradient-based optimization method to discover a solution  $\rho$ .

## CONTENTS

1. Introduction	1
2. Classical Methods and Linear Elasticity	1
3. Heat Compliance	1
Appendix A. Code Listings	1

## 2. CLASSICAL METHODS AND LINEAR ELASTICITY

### 3. HEAT COMPLIANCE

We would like to solve

## APPENDIX A. CODE LISTINGS

The following are Julia versions of top88 and toph.

. top88.jl

```

1 module Top88
2
3 using LinearAlgebra
4 using SparseArrays
5 using Statistics
6
7 export top88
8 export prepare_filter
9 export OC
10
11 """
12     top88(nelx, nely, volfrac, penal, rmin, ft)
13
14     A direct, naive Julia port of Andreassen et al. "Efficient topology optimization in MATLAB
15     using 88 lines of code." By default, this will reproduce the optimized MBB beam from Sigmund
16     (2001).
17
18     # Arguments
19     - 'nelx::S': Number of elements in the horizontal direction
20     - 'nely::S': Number of elements in the vertical direction
21     - 'volfrac::T': Prescribed volume fraction
22     - 'penal::T': The penalization power
23     - 'rmin::T': Filter radius divided by the element size
24     - 'ft::Bool': Choose between sensitivity (if true) or density filter (if false). Defaults
25       to sensitivity filter.
26     - 'write::Bool': If true, will write out iteration number, changes, and density for each
27       iteration. Defaults for false.
28     - 'loop_max::Int': Explicitly set the maximum number of iterations. Defaults to 1000.
29
30     # Returns
31     - 'Matrix{T}': Final material distribution, represented as a matrix
32     """
33 function top88(
34     nelx::S=60,
35     nely::S=20,
36     volfrac::T=0.5,
37     penal::T=3.0,
38     rmin::T=2.0,
39     ft::Bool=true,
40     write::Bool=false,
41     loop_max::Int=1000
42 ) where {S <: Integer, T <: AbstractFloat}
43     # Physical parameters
44     E0 = 1; Emin = 1e-9; nu = 0.3;
45
46     # Prepare finite element analysis
47     A11 = [12 3 -6 -3; 3 12 3 0; -6 3 12 -3; -3 0 -3 12]
48     A12 = [-6 -3 0 3; -3 -6 -3 -6; 0 -3 -6 3; 3 -6 3 -6]
49     B11 = [-4 3 -2 9; 3 -4 -9 4; -2 -9 -4 -3; 9 4 -3 -4]
50     B12 = [2 -3 4 -9; -3 2 9 -2; 4 9 2 3; -9 -2 3 2]
51     KE = 1/(1-nu^2)/24*( [A11 A12; A12' A11] + nu*[B11 B12; B12' B11] )
52
53     nodenrs = reshape(1:(1+nelx)*(1+nely), 1+nely, 1+nelx)
54     edofVec = reshape(2*nodenrs[1:end-1, 1:end-1].+1, nelx*nely, 1)
55     edofMat = zeros{Int64, nelx*nely, 8}
56
57     offsets = [0 1 2*nely.+[2 3 0 1] -2 -1]
58     for i = 1:8
59         for j = 1:nelx*nely
60             edofMat[j,i] = edofVec[j] + offsets[i]
61         end
62     end
63
64     iK = reshape(kron(edofMat, ones(8,1))', 64*nelx*nely, 1)
65     jK = reshape(kron(edofMat, ones(1,8))', 64*nelx*nely, 1)
66
67     # Loads and supports
68     F = spzeros(2*(nely+1)*(nelx+1))
69     F[2,1] = -1

```

```

70 U = spzeros(2*(nely+1)*(nelx+1))
71
72 fixeddofs = union(1:2*(nely+1), [2*(nelx+1)*(nely+1)])
73 alldofs = 1:2*(nely+1)*(nelx+1)
74 freeddofs = setdiff(alldofs, fixeddofs)
75
76 # Prepare the filter
77 H, Hs = prepare_filter(nelx, nely, rmin)
78
79 # Initialize iteration
80 x = volfrac*ones(nely,nelx)
81 xPhys = x
82 loop = 0
83 change = 1
84 cValues = []
85
86 # Start iteration
87 while change > 0.01
88     loop += 1
89     # FE-Analysis
90     sK = [j*((i+Emin)^penal) for i in ((E0-Emin)*xPhys[:]) for j in KE[:]]
91     K = sparse(iK[:], jK[:], sK)
92     K = (K+K')/2
93
94     KK = cholesky(K[freddofs,freddofs])
95     U[freddofs] = KK \ F[freddofs]
96
97     # OLD: edM = [convert(Int64,i) for i in edofMat]
98     mat = (U[edofMat]*KE).*U[edofMat]
99
100    # Objective function and sensitivity analysis
101    ce = reshape([sum(mat[i,:]) for i = 1:(size(mat)[1])],nely,nelx)
102    c = sum(sum((Emin*ones(size(xPhys)).+(xPhys.^penal)*(E0-Emin)).*ce))
103    push!(cValues,c)
104    dc = -penal*(E0-Emin)*xPhys.^(penal-1).*ce
105    dv = ones(nely,nelx)
106
107    # Filtering/ modification of sensitivities
108    if ft
109        dc[:] = H*(x[:].*dc[:])./Hs./max(1e-3,maximum(x[:]))
110    else
111        dc[:] = H*(dc[:])./Hs
112        dv[:] = H*(dv[:])./Hs
113    end
114
115    # Optimality criteria update of design variables and physical densities
116    xnew = OC(nelx, nely, x, volfrac, dc, dv, xPhys, ft)
117
118    change = maximum(abs.(x-xnew))
119    x = xnew
120
121    write && println("Loop = ", loop, ", Change = ", change, ", c = ", c, ", structural density = ", mean(x))
122    loop >= loop_max && break
123 end
124
125 return x
126 end
127
128 """
129 Prepare sensitivity/ density filter
130 """
131
132 function prepare_filter(nelx::S, nely::S, rmin::T) where {S <: Integer, T <: AbstractFloat}
133     iH = ones(nelx*nely*(2*(convert(Int64,ceil(rmin)-1)+1)^2))
134     jH = ones(size(iH))
135     sH = zeros(size(iH))
136     k = 0
137     for i1 = 1:nelx
138         for j1 = 1:nely
139             e1 = (i1-1)*nely+j1
140             for i2 = max(i1-(ceil(rmin)-1),1):min(i1+(ceil(rmin)-1),nelx)
141                 for j2 = max(j1-(ceil(rmin)-1),1):min(j1+(ceil(rmin)-1),nely)

```

```

142             e2 = (i2-1)*nely+j2
143             k += 1
144             iH[k] = e1
145             jH[k] = e2
146             sH[k] = max(0, rmin-sqrt((i1-i2)^2+(j1-j2)^2))
147         end
148     end
149 end
150 end
151 H = sparse(iH,jH,sH)
152 Hs = [sum(H[i,:]) for i = 1:(size(H)[1])]
153
154 return H, Hs
155 end
156
157 """
158 Optimality criteria update
159 """
160 function OC(
161     nelx::S,
162     nely,
163     x,
164     volfrac,
165     dc::Matrix{T},
166     dv,
167     xPhys::Matrix{T},
168     ft::Bool
169 ) where {S <: Integer, T <: AbstractFloat}
170     l1 = 0; l2 = 1e9; move = 0.2
171     xnew = zeros(nely, nelx)
172
173     while (l2-l1)/(l1+l2) > 1e-3
174         lmid = 0.5*(l2+l1)
175         RacBe = sqrt.(-dc./dv/lmid)
176         XB = x.*RacBe
177
178         for i = 1:nelx
179             for j = 1:nely
180                 xji = x[j,i]
181                 xnew[j,i] = max(0.000, max(xji-move, min(1, min(xji+move, XB[j,i]))))
182             end
183         end
184
185         if ft
186             xPhys = xnew
187         else
188             xPhys[:] = (H*xnew[:])./Hs
189         end
190
191         if sum(xPhys[:]) > volfrac*nelx*nely
192             l1 = lmid
193         else
194             l2 = lmid
195         end
196     end
197
198     return xnew
199 end
200
201 end

```

. toph.jl

```

1 module TopH
2
3 using LinearAlgebra
4 using SparseArrays
5 using Statistics
6
7 export toph
8 export OC
9 export check

```

```

10 export FE
11
12 """
13     toph(nelx, nely, volfrac, penal, rmin, write, loop_max)
14
15 A direct, naive Julia port of the 'toph' code listing from "Topology Optimization"
16 by Martin Bendsøe and Ole Sigmund.
17
18 # Arguments
19 - 'nelx::S': Number of elements in the horizontal direction
20 - 'nely::S': Number of elements in the vertical direction
21 - 'volfrac::T': Prescribed volume fraction
22 - 'penal::T': The penalization power
23 - 'rmin::T': Filter radius divided by the element size
24 - 'write::Bool': If true, will write out iteration number, changes, and density
25   for each iteration. Defaults to false.
26 - 'loop_max::Int': Explicitly set the maximum number of iterations. Defaults to 1000.
27
28 # Returns
29 - 'Matrix{T}': Final material distribution, represented as a matrix.
30 """
31 function toph(
32     nelx::S,
33     nely::S,
34     volfrac::T,
35     penal::T,
36     rmin::T,
37     write::Bool=false,
38     loop_max::Int=100
39 ) where {S <: Integer, T <: AbstractFloat}
40     # Initialization
41     x = volfrac * ones(nely, nelx)
42     loop = 0
43     change = 1.
44     dc = zeros(nely, nelx)
45
46     # Start iteration
47     while change > 0.01
48         loop += 1
49         xold = x
50         c = 0.
51
52         # FE Analysis
53         U = FE(nelx, nely, x, penal)
54
55         KE = [ 2/3 -1/6 -1/3 -1/6
56              -1/6 2/3 -1/6 -1/3
57              -1/3 -1/6 2/3 -1/6
58              -1/6 -1/3 -1/6 2/3 ]
59
60         # Objective function/ sensitivity analysis
61         for ely = 1:nely
62             for elx = 1:nelx
63                 n1 = (nely+1)*(elx-1)+ely
64                 n2 = (nely+1)* elx +ely
65                 Ue = U[[n1; n2; n2+1; n1+1]]
66
67                 c += (0.001+0.999*x[ely, elx]^penal)*Ue'*KE*Ue
68                 dc[ely, elx] = -0.999*penal*(x[ely, elx])^(penal-1)*Ue'*KE*Ue
69             end
70         end
71
72         # Sensitivity filtering
73         dc = check(nelx, nely, rmin, x, dc)
74         # Design update by optimality criteria method
75         x = OC(nelx, nely, x, volfrac, dc)
76
77         # Print out results if desired
78         if write
79             change = maximum(abs.(x-xold))
80             println("Change = ", change, " c = ", c)
81         end

```

```

82         loop >= loop_max && break
83     end
84
85
86     return x
87 end
88
89 """
90     OC(nelx, nely, x, volfrac, dc)
91
92     Optimality criteria update
93
94     # Arguments
95     - 'nelx::S': Number of elements in the horizontal direction
96     - 'nely::S': Number of elements in the vertical direction
97     - 'x::Matrix{T}': Current material distribution
98     - 'volfrac::T': Prescribed volume fraction
99     - 'dc::Matrix{T}': Sensitivity filter
100
101     # Returns
102     - 'Matrix{T}': Updated material distribution
103
104     """
105     function OC(
106         nelx::S,
107         nely::S,
108         x::Matrix{T},
109         volfrac::T,
110         dc::Matrix{T}
111     ) where {S <: Integer, T <: AbstractFloat}
112         l1 = 0; l2 = 100000; move = 0.2
113         xnew = zeros(nely, nelx)
114
115         while (l2-l1) > 1e-4
116             lmid = 0.5*(l2+l1)
117             RacBe = sqrt.(-dc/lmid)
118             XB = x .* RacBe
119
120             for i = 1:nelx
121                 for j = 1:nely
122                     xji = x[j,i]
123                     xnew[j,i] = max(0.001, max(xji-move, min(1, min(xji+move, XB[j,i]))))
124                 end
125             end
126
127             if (sum(sum(xnew)) - volfrac*nelx*nely) > 0
128                 l1 = lmid
129             else
130                 l2 = lmid
131             end
132         end
133
134         return xnew
135     end
136
137     """
138     check(nelx, nely, rmin, x, dc)
139
140     Mesh independency filter
141
142     # Arguments
143     - 'nelx::S': Number of elements in the horizontal direction
144     - 'nely::S': Number of elements in the vertical direction
145     - 'rmin::T': Sensitivity filter radius divided by element size
146     - 'x::Matrix{T}': Current material distribution
147     - 'dc::Matrix{T}': Compliance derivatives
148
149     # Returns
150     - 'Matrix{T}': Updated dc
151
152     """
153     function check(nelx::S,
154         nely::S,

```

```

154     rmin::T,
155     x::Matrix{T},
156     dc::Matrix{T}
157 ) where {S <: Integer, T <: AbstractFloat}
158     dcn=zeros(nely,nelx)
159
160     for i = 1:nelx
161         for j = 1:nely
162             sum=0.0
163
164             for k = max(i-floor(rmin),1):min(i+floor(rmin),nelx)
165                 for l = max(j-floor(rmin),1):min(j+floor(rmin),nely)
166                     l = Int64(l); k = Int64(k)
167                     fac = rmin-sqrt((i-k)^2+(j-l)^2)
168                     sum = sum+max(0,fac)
169                     dcn[j,i] += max(0,fac)*x[l,k]*dc[l,k]
170                 end
171             end
172
173             dcn[j,i] = dcn[j,i]/(x[j,i]*sum)
174         end
175     end
176
177     return dcn
178 end
179
180 """
181     FE(nelx, nely, x, penal)
182
183     Finite element implementation
184
185     # Arguments
186     - 'nelx::S': Number of elements in the horizontal direction
187     - 'nely::S': Number of elements in the vertical direction
188     - 'x::Matrix{T}': Current material distribution
189     - 'penal::T': The penalization power
190
191     # Returns
192     - 'Matrix{T}': Differential equation solution U
193     """
194     function FE(
195         nelx::S,
196         nely::S,
197         x::Matrix{T},
198         penal::T
199     ) where {S <: Integer, T <: AbstractFloat}
200         KE = [ 2/3 -1/6 -1/3 -1/6
201                -1/6 2/3 -1/6 -1/3
202                -1/3 -1/6 2/3 -1/6
203                -1/6 -1/3 -1/6 2/3 ]
204
205         K = spzeros((nelx+1)*(nely+1), (nelx+1)*(nely+1))
206         U = zeros((nely+1)*(nelx+1))
207         F = zeros((nely+1)*(nelx+1))
208         for elx = 1:nelx
209             for ely = 1:nely
210                 n1 = (nely+1)*(elx-1)+ely
211                 n2 = (nely+1)* elx +ely
212                 edof = [n1; n2; n2+1; n1+1]
213                 K[edof,edof] += (0.001+0.999*x[ely,elx]^penal)*KE
214             end
215         end
216
217         F .= 0.01
218         fixeddofs = Int64(nely/2+1-(nely/20)):Int64(nely/2+1+(nely/20))
219         alldofs = 1:(nely+1)*(nelx+1)
220         freeddofs = setdiff(alldofs,fixeddofs)
221
222         U[freedofs] = K[freedofs, freedofs] \ F[freedofs]
223         U[fixeddofs] .= 0
224
225         return U

```

226 **end**

227

228 **end**