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

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

where Ω is the design domain, $\rho \in L^2(\Omega)$ is a function describing the material distribution, f forms the objective function, θ is the *mass fraction* (that is, the fraction of Ω 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 ρ.

Contents

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`

```
module Top88

using LinearAlgebra
using SparseArrays
using Statistics

export top88
export prepare_filter
export OC

"""
    top88(nelx, nely, volfrac, penal, rmin, ft)

A direct, naive Julia port of Andreassen et al. "Efficient topology optimization in MATLAB
using 88 lines of code." By default, this will reproduce the optimized MBB beam from Sigmund
(2001).

# Arguments
- ‘nelx::S‘: Number of elements in the horizontal direction
- ‘nely::S‘: Number of elements in the vertical direction
- ‘volfrac::T‘: Prescribed volume fraction
- ‘penal::T‘: The penalization power
- ‘rmin::T‘: Filter radius divided by the element size
- ‘ft::Bool‘: Choose between sensitivity (if true) or density filter (if false). Defaults
    to sensitivity filter.
- ‘write::Bool‘: If true, will write out iteration number, changes, and density for each
    iteration. Defaults for false.
- ‘loop_max::Int‘: Explicitly set the maximum number of iterations. Defaults to 1000.

# Returns
- ‘Matrix{T}‘: Final material distribution, represented as a matrix
"""
function top88(
    nelx::S=60,
    nely::S=20,
    volfrac::T=0.5,
    penal::T=3.0,
    rmin::T=2.0,
    ft::Bool=true,
    write::Bool=false,
    loop_max::Int=1000
) where {S <: Integer, T <: AbstractFloat}
    # Physical parameters
    E0 = 1; Emin = 1e-9; nu = 0.3;

    # Prepare finite element analysis
    A11 = [12  3 -6 -3;  3 12  3  0; -6  3 12 -3; -3  0 -3 12]
    A12 = [-6 -3  0  3; -3 -6 -3 -6;  0 -3 -6  3;  3 -6  3 -6]
    B11 = [-4  3 -2  9;  3 -4 -9  4; -2 -9 -4 -3;  9  4 -3 -4]
    B12 = [ 2 -3  4 -9; -3  2  9 -2;  4  9  2  3; -9 -2  3  2]
    KE = 1/(1-nu^2)/24*([A11 A12;A12' A11]+nu*[B11 B12;B12' B11])

    nodenrs = reshape(1:(1+nelx)*(1+nely),1+nely,1+nelx)
    edofVec = reshape(2*nodenrs[1:end-1,1:end-1].+1, nelx*nely, 1)
    edofMat = zeros(Int64, nelx*nely, 8)

    offsets = [0 1 2*nely.+[2 3 0 1] -2 -1]
    for i = 1:8
        for j = 1:nelx*nely
            edofMat[j,i]= edofVec[j] + offsets[i]
        end
    end

    iK = reshape(kron(edofMat,ones(8,1))', 64*nelx*nely,1)
    jK = reshape(kron(edofMat,ones(1,8))', 64*nelx*nely,1)

    # Loads and supports
    F = spzeros(2*(nely+1)*(nelx+1))
    F[2,1] = -1
```

```
    U = spzeros(2*(nely+1)*(nelx+1))

    fixeddofs = union(1:2:2*(nely+1), [2*(nelx+1)*(nely+1)])
    alldofs = 1:2*(nely+1)*(nelx+1)
    freedofs = setdiff(alldofs, fixeddofs)

    # Prepare the filter
    H, Hs = prepare_filter(nelx, nely, rmin)

    # Initialize iteration
    x = volfrac*ones(nely,nelx)
    xPhys = x
    loop = 0
    change = 1
    cValues = []

    # Start iteration
    while change > 0.01
        loop += 1
        # FE-Analysis
        sK = [j*((i+Emin)^penal) for i in ((E0-Emin)*xPhys[:]') for j in KE[:]]
        K = sparse(iK[:], jK[:], sK)
        K = (K+K')/2

        KK = cholesky(K[freedofs,freedofs])
        U[freedofs] = KK \ F[freedofs]

        # OLD: edM = [convert(Int64,i) for i in edofMat]
        mat = (U[edofMat]*KE).*U[edofMat]

        # Objective function and sensitivity analysis
        ce = reshape([sum(mat[i,:]) for i = 1:(size(mat)[1])],nely,nelx)
        c = sum(sum((Emin*ones(size(xPhys)).+(xPhys.^penal)*(E0-Emin)).*ce))
        push!(cValues,c)
        dc = -penal*(E0-Emin)*xPhys.^(penal-1).*ce
        dv = ones(nely,nelx)

        # Filtering/ modification of sensitivities
        if ft
            dc[:] = H*(x[:].*dc[:])./Hs./max(1e-3,maximum(x[:]))
        else
            dc[:] = H*(dc[:]./Hs)
            dv[:] = H*(dv[:]./Hs)
        end

        # Optimality criteria update of design variables and physical densities
        xnew = OC(nelx, nely, x, volfrac, dc, dv, xPhys, ft)

        change = maximum(abs.(x-xnew))
        x = xnew

        write && println("Loop = ", loop, ", Change = ", change ,", c = ", c, ", structural density = ", mean(x))
        loop >= loop_max && break
    end

    return x
end


"""
Prepare sensitivity/ density filter
"""
function prepare_filter(nelx::S, nely::S, rmin::T) where {S <: Integer, T <: AbstractFloat}
    iH = ones(nelx*nely*(2*(convert(Int64,ceil(rmin)-1))+1)^2)
    jH = ones(size(iH))
    sH = zeros(size(iH))
    k = 0
    for i1 = 1:nelx
        for j1 = 1:nely
            e1 = (i1-1)*nely+j1
            for i2 = max(i1-(ceil(rmin)-1),1):min(i1+(ceil(rmin)-1),nelx)
                for j2 = max(j1-(ceil(rmin)-1),1):min(j1+(ceil(rmin)-1),nely)
```

```
                    e2 = (i2-1)*nely+j2
                    k += 1
                    iH[k] = e1
                    jH[k] = e2
                    sH[k] = max(0,rmin-sqrt((i1-i2)^2+(j1-j2)^2))
                end
            end
        end
    end
    H = sparse(iH,jH,sH)
    Hs = [sum(H[i,:]) for i = 1:(size(H)[1])]

    return H, Hs
end

"""
Optimality criteria update
"""
function OC(
    nelx::S,
    nely,
    x,
    volfrac,
    dc::Matrix{T},
    dv,
    xPhys::Matrix{T},
    ft::Bool
) where {S <: Integer, T <: AbstractFloat}
    l1 = 0; l2 = 1e9; move = 0.2
    xnew = zeros(nely, nelx)

    while (l2-l1)/(l1+l2) > 1e-3
        lmid = 0.5*(l2+l1)
        RacBe = sqrt.(-dc./dv/lmid)
        XB = x.*RacBe

        for i = 1:nelx
            for j = 1:nely
                xji = x[j,i]
                xnew[j,i]= max(0.000,max(xji-move,min(1,min(xji+move,XB[j,i]))))
            end
        end

        if ft
            xPhys = xnew
        else
            xPhys[:] = (H*xnew[:])./Hs
        end

        if sum(xPhys[:]) > volfrac*nelx*nely
            l1 = lmid
        else
            l2 = lmid
        end
    end

    return xnew
end

end
```

. toph.jl

```
module TopH

using LinearAlgebra
using SparseArrays
using Statistics

export toph
export OC
export check
```

```
export FE

"""
    toph(nelx, nely, volfrac, penal, rmin, write, loop_max)

A direct, naive Julia port of the `toph` code listing from "Topology Optimization"
by Martin Bendsoe and Ole Sigmund.

# Arguments
- `nelx::S`: Number of elements in the horizontal direction
- `nely::S`: Number of elements in the vertical direction
- `volfrac::T`: Prescribed volume fraction
- `penal::T`: The penalization power
- `rmin::T`: Filter radius divided by the element size
- `write::Bool`: If true, will write out iteration number, changes, and density
    for each iteration. Defaults to false.
- `loop_max::Int`: Explicitly set the maximum number of iterations. Defaults to 1000.

# Returns
- `Matrix{T}`: Final material distribution, represented as a matrix.
"""
function toph(
    nelx::S,
    nely::S,
    volfrac::T,
    penal::T,
    rmin::T,
    write::Bool=false,
    loop_max::Int=100
) where {S <: Integer, T <: AbstractFloat}
    # Initialization
    x = volfrac * ones(nely,nelx)
    loop = 0
    change = 1.
    dc = zeros(nely,nelx)

    # Start iteration
    while change > 0.01
        loop += 1
        xold = x
        c = 0.

        # FE Analysis
        U = FE(nelx,nely,x,penal)

        KE = [ 2/3 -1/6 -1/3 -1/6
                -1/6  2/3 -1/6 -1/3
                -1/3 -1/6  2/3 -1/6
                -1/6 -1/3 -1/6  2/3 ]

        # Objective function/ sensitivity analysis
        for ely = 1:nely
            for elx = 1:nelx
                n1 = (nely+1)*(elx-1)+ely
                n2 = (nely+1)* elx   +ely
                Ue = U[[n1; n2; n2+1; n1+1]]

                c += (0.001+0.999*x[ely,elx]^penal)*Ue'*KE*Ue
                dc[ely,elx] = -0.999*penal*(x[ely,elx])^(penal-1)*Ue'*KE*Ue
            end
        end

        # Sensitivity filtering
        dc = check(nelx,nely,rmin,x,dc)
        # Design update by optimality criteria method
        x  = OC(nelx,nely,x,volfrac,dc)

        # Print out results if desired
        if write
            change = maximum(abs.(x-xold))
            println("Change = ", change, " c = ", c)
        end
```

```
        loop >= loop_max && break
    end

    return x
end

"""
    OC(nelx, nely, x, volfrac, dc)

Optimality criteria update

# Arguments
- `nelx::S`: Number of elements in the horizontal direction
- `nely::S`: Number of elements in the vertical direction
- `x::Matrix{T}`: Current material distribution
- `volfrac::T`: Prescribed volume fraction
- `dc::Matrix{T}`: Sensitivity filter

# Returns
- `Matrix{T}`: Updated material distribution

"""
function OC(
    nelx::S,
    nely::S,
    x::Matrix{T},
    volfrac::T,
    dc::Matrix{T}
) where {S <: Integer, T <: AbstractFloat}
    l1 = 0; l2 = 100000; move = 0.2
    xnew = zeros(nely,nelx)

    while (l2-l1) > 1e-4
        lmid = 0.5*(l2+l1)
        RacBe = sqrt.(-dc/lmid)
        XB = x .* RacBe

        for i = 1:nelx
            for j = 1:nely
                xji = x[j,i]
                xnew[j,i]= max(0.001,max(xji-move,min(1,min(xji+move,XB[j,i]))))
            end
        end

        if (sum(sum(xnew)) - volfrac*nelx*nely) > 0
            l1 = lmid
        else
            l2 = lmid
        end
    end

    return xnew
end

"""
    check(nelx, nely, rmin, x, dc)

Mesh independency filter

# Arguments
- `nelx::S`: Number of elements in the horizontal direction
- `nely::S`: Number of elements in the vertical direction
- `rmin::T`: Sensitivity filter radius divided by element size
- `x::Matrix{T}`: Current material distribution
- `dc::Matrix{T}`: Compliance derivatives

# Returns
- `Matrix{T}`: Updated dc
"""
function check(nelx::S,
    nely::S,
```

```
    rmin::T,
    x::Matrix{T},
    dc::Matrix{T}
) where {S <: Integer, T <: AbstractFloat}
    dcn=zeros(nely,nelx)

    for i = 1:nelx
        for j = 1:nely
        sum=0.0

        for k = max(i-floor(rmin),1):min(i+floor(rmin),nelx)
            for l = max(j-floor(rmin),1):min(j+floor(rmin),nely)
            l = Int64(l); k = Int64(k)
            fac = rmin-sqrt((i-k)^2+(j-l)^2)
            sum = sum+max(0,fac)
            dcn[j,i] += max(0,fac)*x[l,k]*dc[l,k]
            end
        end

        dcn[j,i] = dcn[j,i]/(x[j,i]*sum)
        end
    end

    return dcn
end

"""
    FE(nelx, nely, x, penal)

Finite element implementation

# Arguments
- `nelx::S`: Number of elements in the horizontal direction
- `nely::S`: Number of elements in the vertical direction
- `x::Matrix{T}`: Current material distribution
- `penal::T`: The penalization power

# Returns
- `Matrix{T}`: Differential equation solution U
"""
function FE(
    nelx::S,
    nely::S,
    x::Matrix{T},
    penal::T
) where {S <: Integer, T <: AbstractFloat}
    KE = [ 2/3 -1/6 -1/3 -1/6
            -1/6  2/3 -1/6 -1/3
            -1/3 -1/6  2/3 -1/6
            -1/6 -1/3 -1/6  2/3 ]

    K = spzeros((nelx+1)*(nely+1), (nelx+1)*(nely+1))
    U = zeros((nely+1)*(nelx+1))
    F = zeros((nely+1)*(nelx+1))
    for elx = 1:nelx
        for ely = 1:nely
            n1 = (nely+1)*(elx-1)+ely
            n2 = (nely+1)* elx   +ely
            edof = [n1; n2; n2+1; n1+1]
            K[edof,edof] += (0.001+0.999*x[ely,elx]^penal)*KE
        end
    end

    F .= 0.01
    fixeddofs = Int64(nely/2+1-(nely/20)):Int64(nely/2+1+(nely/20))
    alldofs = 1:(nely+1)*(nelx+1)
    freedofs = setdiff(alldofs,fixeddofs)

    U[freedofs] = K[freedofs, freedofs] \ F[freedofs]
    U[fixeddofs] .= 0

    return U
```

```
end

end
```