인공지능과 설계: 해석 예측에서 설계 최적화까지

위상최적설계 (Topology Optimization) 실습

장인권 교수님 연구실
박사과정 고혁진

hyukjin.koh@kaist.ac.kr

Korea Advanced Institute of Science and Technology (KAIST)

Contents

Google Colab environment setting

Topology optimization using Python

▪ Introduction

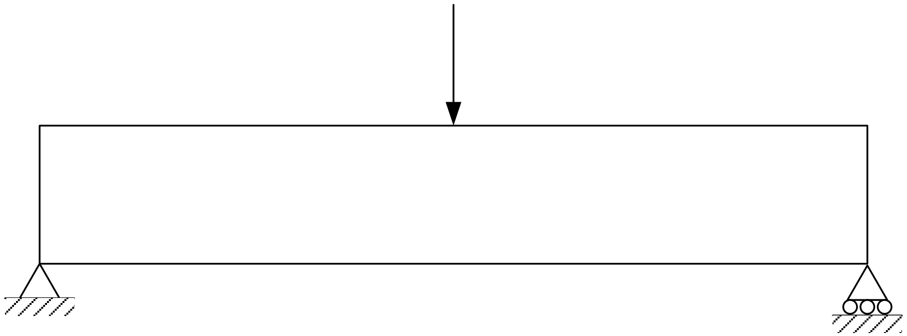
- The Python code for topology optimization problem is an open-source alternative to the 99 and 88 line MATLAB codes*.
  - Original code is provided by Topology Optimization group in DTU.
  - You can access various topology optimization apps/software in https://www.topopt.mek.dtu.dk.

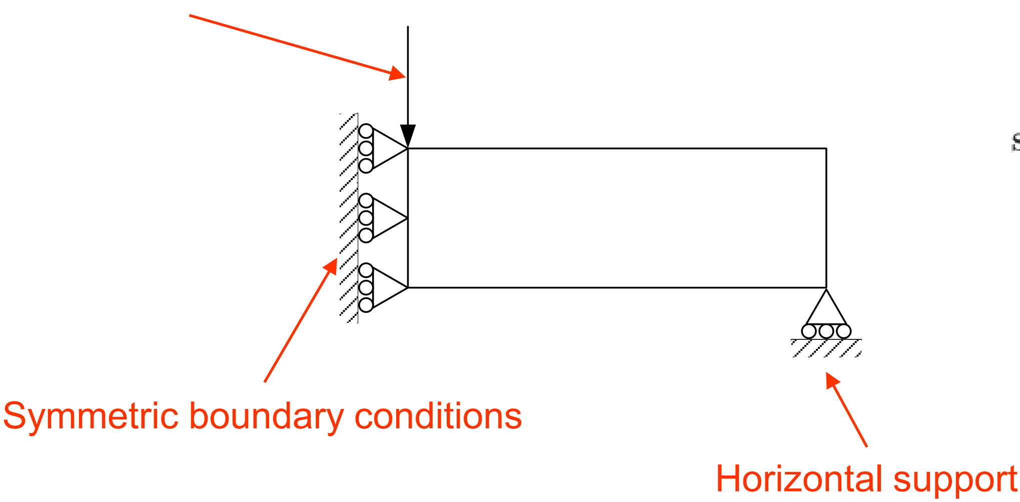
- This code is useful for students and newcomers to the field of topology optimization in the educational sense.
  - It can be extended to include more functions such as multiple load-cases, alternative mesh-independency schemes, passive areas, etc.

*O. Sigmund, "A 99 line topology optimization code written in Matlab", *Structural Multidisciplinary Optimization 21*, 120-127, 2001.
*E. Andreassen, A. Clausen, M. Schevenels, B. S. Lazarov, O. Sigmund, "Efficient topology optimization in MATLAB using 88 lines of code". *Structural Multidisciplinary Optimization 43*, 1–16, 2011.

▪ Problem formulation

- The default problem in the code is MBB-beam.
- The modified SIMP method has various scalability advantages. (e.g. additional filters)



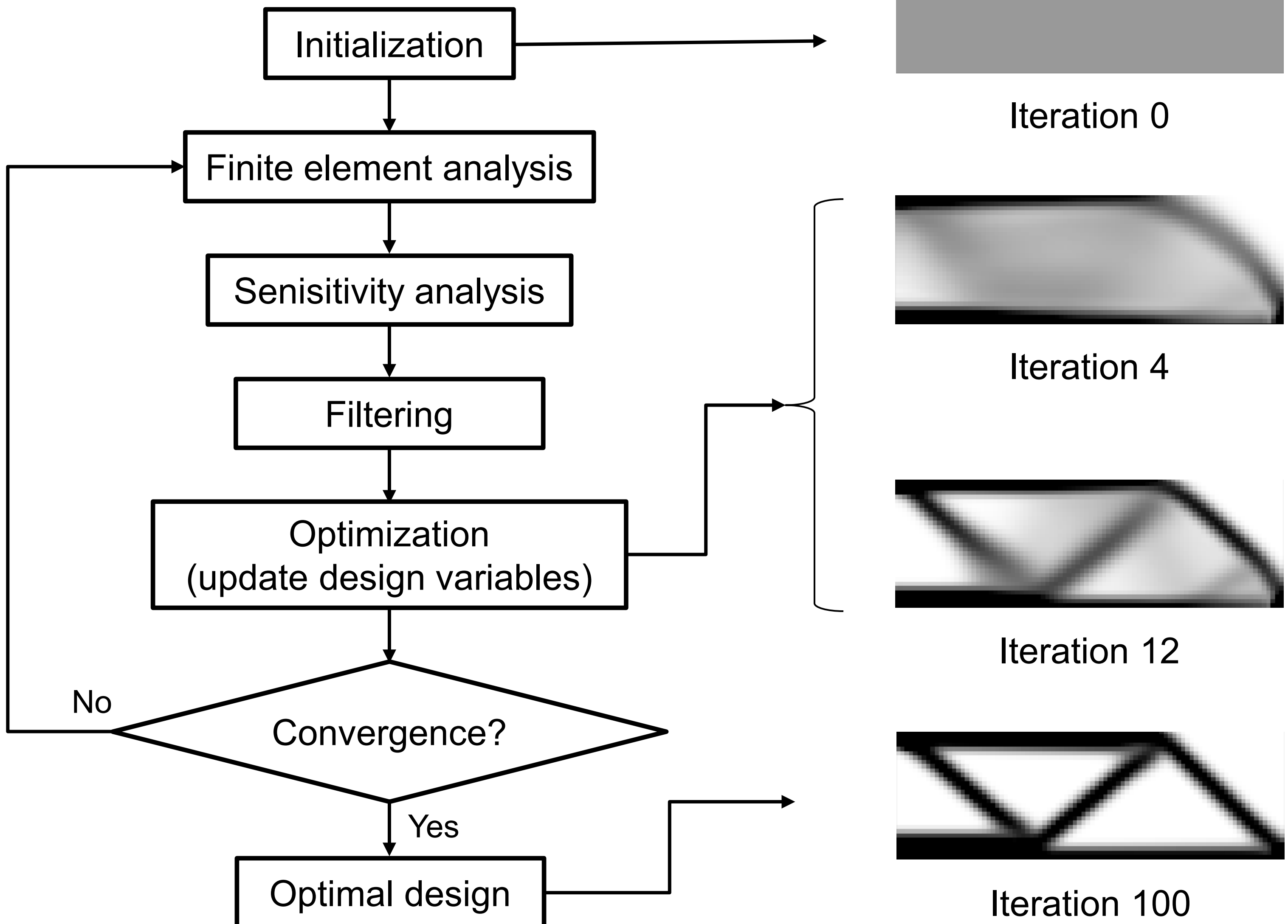


Modified SIMP; $E_{\min} = 10^{-9}$ in this practice

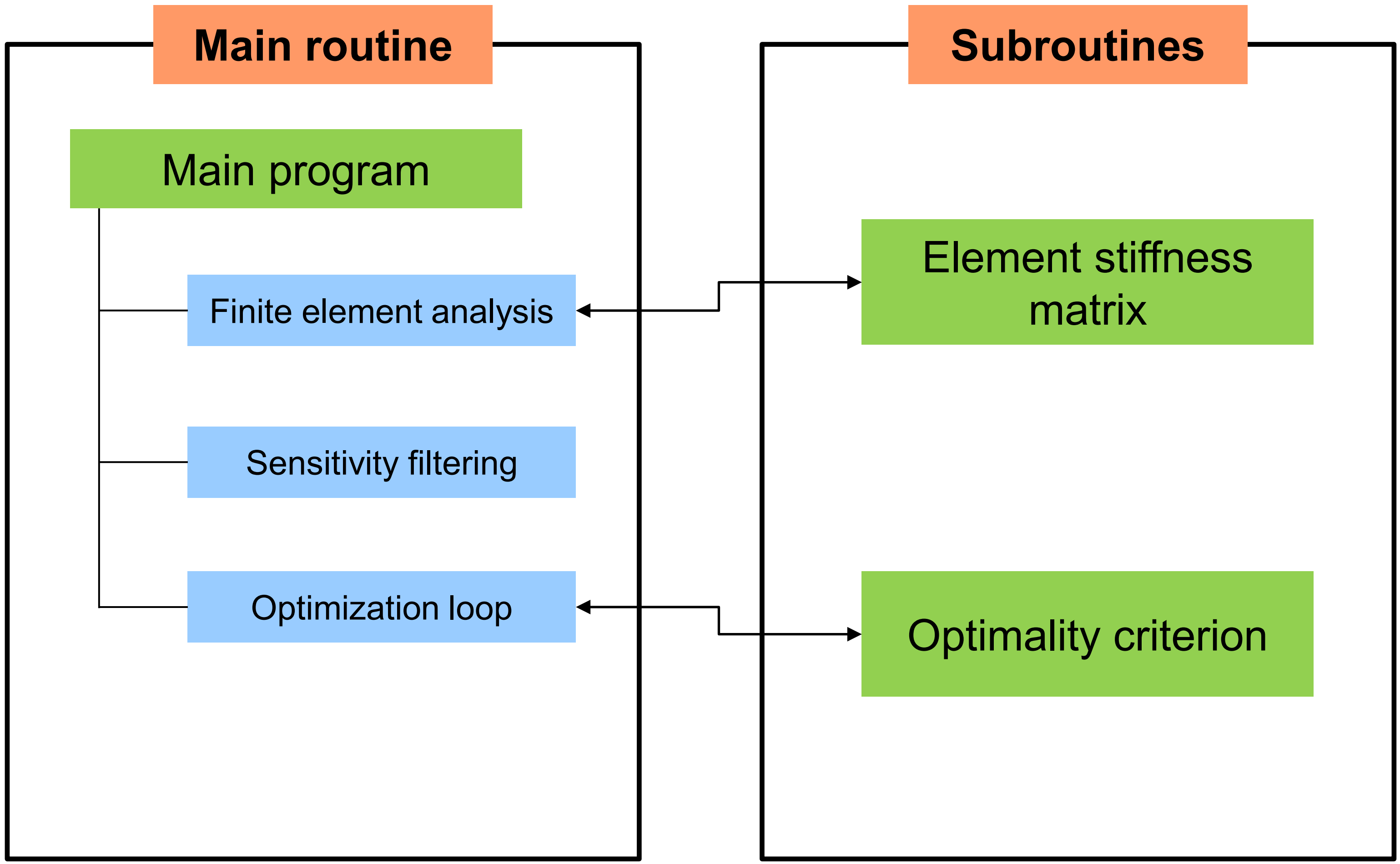
$$\boxed{E_e(x_e) = E_{\min} + x_e^p (E_0 - E_{\min}), \qquad x_e \in [0,1]}$$

$$\begin{aligned}
\min_{\mathbf{x}} : \quad & c(\mathbf{x}) = \mathbf{U}^{\mathrm{T}}\mathbf{K}\mathbf{U} = \sum_{e=1}^{N} E_e(x_e)\mathbf{u}_e^{\mathrm{T}}\mathbf{k}_0\mathbf{u}_e \\
\text{subject to:} \quad & V(\mathbf{x})/V_0 = f \\
& \mathbf{K}\mathbf{U} = \mathbf{F} \\
& \mathbf{0} \leq \mathbf{x} \leq \mathbf{1}
\end{aligned}$$

- **Structure of the code**



- **Structure of the code**

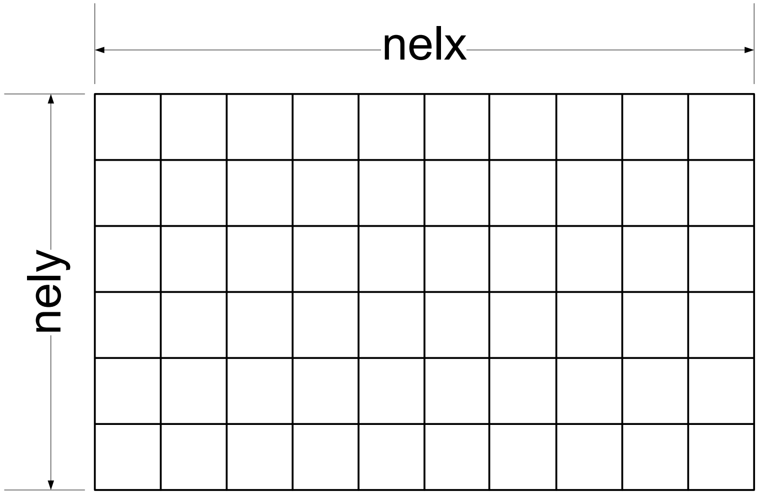


▪ Python implementation

➢ The main program is called from the Python command by
`main(nelx,nely,volfrac,penal,rmin,ft)`

`nelx`: number of elements in the horizontal direction

`nely`: number of elements in the vertical direction



▪ Python implementation

`volfrac`: volume fraction ratio for a constraint

$$\text{volfrac} = \frac{\sum_{e=1}^{N} x_e v_e}{V_0}$$

`penal`: penalization power p in SIMP method

$$E_e(x_e) = E_{\min} + x_e^p (E_0 - E_{\min}), \qquad x_e \in [0,1]$$

▪ Python implementation

`rmin`: filter radius (divided by element size)

$$\widehat{\frac{\partial c}{\partial x_e}} = \frac{1}{\max(\gamma, x_e)\sum_{i \in N_e} H_{ei}} \sum_{i \in N_e} H_{ei} x_i \frac{\partial c}{\partial x_i}$$

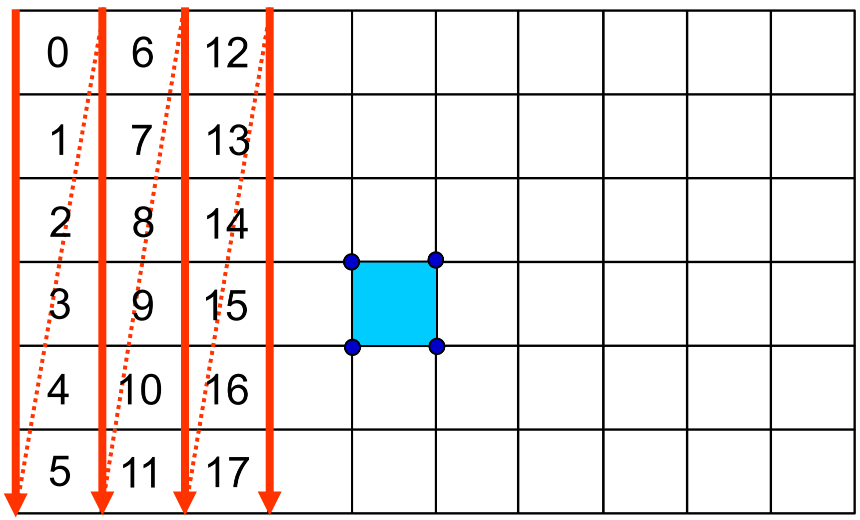
$$where \quad \widehat{H}_{if} = \max(0, \boxed{r_{\min}} - dist(e,i))$$

`ft`: sensitivity filtering (`ft = 1`) or density filtering (`ft = 2`)

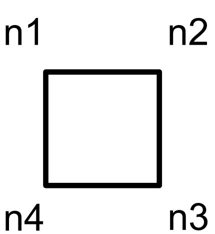
only use sensitivity filtering in this practice!

▪ Python implementation

Element, node, and nodal displacement numbering



Local number



Global number

$$n1 = (\text{nely}+1)\times \text{elx}+\text{ely}$$

$$n2 = (\text{nely}+1)\times (\text{elx}+1)+\text{ely}$$

In this case, nelx=10, nely=6, elx=4, ely=3

$\therefore$ n1=31, n2=38, n3=39, n4=32

Compliance and its sensitivty

$$c = \mathbf{U}^T\mathbf{F} = \mathbf{U}^T\mathbf{K}\mathbf{U} = \sum_{e=1}^{N} E_e(x_e)\mathbf{u}_e^T\mathbf{k}_0\mathbf{u}_e$$

$$\frac{\partial c}{\partial \rho_i} = -\mathbf{U}^T\frac{\partial \mathbf{K}}{\partial \rho_i}\mathbf{U}$$

nodal displacement vector

$$\boxed{\begin{aligned}\mathbf{u}_e &= [u_{n1,x}; u_{n1,y}; u_{n2,x}; u_{n2,y}; u_{n3,x}; u_{n3,y}; u_{n4,x}; u_{n4,y}] \\ &= [2*n1; 2*n1+1; 2*n2; 2*n2+1; \\ &\quad \cdots 2*n2+2; 2*n2+3; 2*n1+2; 2*n1+3]\end{aligned}}$$

2 DOFs per node, 8 DOFs per element

▪ Python implementation: Finite element anlaysis

FEA procedure

Step 1: Obtain stiffness matrix for each element

Step 2: Expand element stiffness matrices for assembly

Step 3: Assemble element stiffness matrices

Step 4: Build global $\mathbf{KU=F}$

Step 5: Apply boundary conditions

Step 6: Determine displacement at nodes

Step 7: Determine compliance

▪ Python implementation: Finite element anlaysis

FEA procedure

Step 1: Obtain stiffness matrix for each element

Step 2: Expand element stiffness matrices for assembly

Step 3: Assemble element stiffness matrices

Step 4: Build global $\mathbf{KU=F}$

Step 5: Apply boundary conditions

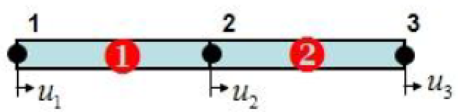
Step 6: Determine displacement at nodes

Step 7: Determine compliance

▪ Python implementation: Finite element anlaysis

Stiffness Matrix Assembly

- Two bar elements (three nodes → a total of 3DOF)



If the entire structure has n DOF, expand each $[k^E]$ to an $n \times n$ matrix.

Element 1 (with nodes 1 and 2): $\begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \end{Bmatrix} = \begin{Bmatrix} \alpha \\ \beta \end{Bmatrix} \Rightarrow \begin{bmatrix} a & b & 0 \\ c & d & 0 \\ 0 & 0 & 0 \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \end{Bmatrix} = \begin{Bmatrix} \alpha \\ \beta \\ 0 \end{Bmatrix}$

Element 2 (with nodes 2 and 3): $\begin{bmatrix} e & f \\ g & h \end{bmatrix} \begin{Bmatrix} u_2 \\ u_3 \end{Bmatrix} = \begin{Bmatrix} \beta \\ \gamma \end{Bmatrix} \Rightarrow \begin{bmatrix} 0 & 0 & 0 \\ 0 & e & f \\ 0 & g & h \end{bmatrix} \begin{Bmatrix} u_1 \\ u_2 \\ u_3 \end{Bmatrix} = \begin{Bmatrix} 0 \\ \beta \\ \gamma \end{Bmatrix}$

$$\therefore [K] = \sum_i [K_i^E] = \begin{bmatrix} a & b & 0 \\ c & d+e & f \\ 0 & g & h \end{bmatrix}$$

▪ Python implementation: Finite element anlaysis

```python
# dofs:
ndof = 2*(nelx+1)*(nely+1)
# Allocate design variables (as array), initialize and allocate sens.
x=volfrac * np.ones(nely*nelx,dtype=float)
xold=x.copy()
xPhys=x.copy()
g=0 # must be initialized to use the NGuyen/Paulino OC approach
dc=np.zeros((nely,nelx), dtype=float)
# FE: Build the index vectors for the for coo matrix format.
KE=lk()
edofMat=np.zeros((nelx*nely,8),dtype=int)
for elx in range(nelx):
  for ely in range(nely):
    el = ely+elx*nely
    n1=(nely+1)*elx+ely
    n2=(nely+1)*(elx+1)+ely
    edofMat[el,:]=np.array([2*n1+2, 2*n1+3, 2*n2+2, 2*n2+3,2*n2, 2*n2+1, 2*n1, 2*n1+1])
# Construct the index pointers for the coo format
iK = np.kron(edofMat,np.ones((8,1))).flatten()
jK = np.kron(edofMat,np.ones((1,8))).flatten()
```

2 DOFs per node

Call elelment (local) stiffness matrix subroutine

```python
# BC's and support
dofs=np.arange(2*(nelx+1)*(nely+1))
fixed=np.union1d(dofs[0:2*(nely+1):2],np.array([2*(nelx+1)*(nely+1)-1]))
free=np.setdiff1d(dofs,fixed)
# Solution and RHS vectors
f=np.zeros((ndof,1))
u=np.zeros((ndof,1))
# Set load
f[1,0]=-1
```

(Symmetric boundary condition)
+ (horizontal support)

Vertical force at node 1

▪ Python implementation: Finite element anlaysis

```python
# Setup and solve FE problem
sK=((KE.flatten()[np.newaxis]).T*(Emin+(xPhys)**penal*(Emax-Emin))).flatten(order='F')
K = coo_matrix((sK,(iK,jK)),shape=(ndof,ndof)).tocsc()
# Remove constrained dofs from matrix
K = K[free,:][:,free]
# Solve system
u[free,0]=spsolve(K,f[free,0])
# Objective and sensitivity
ce[:] = (np.dot(u[edofMat].reshape(nelx*nely,8),KE) * u[edofMat].reshape(nelx*nely,8) ).sum(1)
obj=( (Emin+xPhys**penal*(Emax-Emin))*ce ).sum()
dc[:]=(-penal*xPhys**(penal-1)*(Emax-Emin))*ce
dv[:] = np.ones(nely*nelx)
```

Global stiffness matrix **K**

Solve **KU=F**

$$c = \mathbf{U}^T\mathbf{F} = \mathbf{U}^T\mathbf{K}\mathbf{U} = \sum_{e=1}^{N} E_e(x_e)\mathbf{u}_e^T\mathbf{k}_0\mathbf{u}_e$$

$$\frac{\partial c}{\partial \rho_i} = -\mathbf{U}^T\frac{\partial \mathbf{K}}{\partial \rho_i}\mathbf{U}$$

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

Element stiffness matrix

Squre bilinear 4-node element

▪ Python implementation: Sensitivity filtering

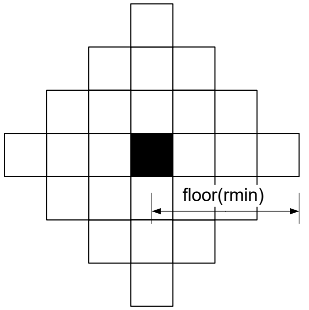
```
# Filter: Build (and assemble) the index+data vectors for the coo matrix format
nfilter=int(nelx*nely*((2*(np.ceil(rmin)-1)+1)**2))
iH = np.zeros(nfilter)
jH = np.zeros(nfilter)
sH = np.zeros(nfilter)
cc=0
for i in range(nelx):
  for j in range(nely):
    row=i*nely+j
    kk1=int(np.maximum(i-(np.ceil(rmin)-1),0))
    kk2=int(np.minimum(i+np.ceil(rmin),nelx))
    ll1=int(np.maximum(j-(np.ceil(rmin)-1),0))
    ll2=int(np.minimum(j+np.ceil(rmin),nely))
    for k in range(kk1,kk2):
      for l in range(ll1,ll2):
        col=k*nely+l
        fac=rmin-np.sqrt(((i-k)*(i-k)+(j-l)*(j-l)))
        iH[cc]=row
        jH[cc]=col
        sH[cc]=np.maximum(0.0,fac)
        cc=cc+1
```

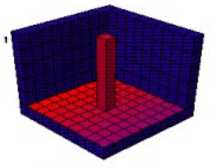
```
  # Sensitivity filtering:
  if ft==0:
    dc[:] = np.asarray((H*(x*dc))[np.newaxis].T/Hs)[:,0] / np.maximum(0.001,x)
```

$$\widehat{\frac{\partial c}{\partial x_e}} = \frac{1}{\max(\gamma, x_e)\sum_{i\in N_e} H_{ei}} \sum_{i\in N_e} H_{ei} x_i \frac{\partial c}{\partial x_i}$$

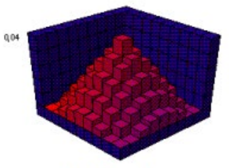
$$\text{where} \quad \widehat{H}_{ei} = \max(0, r_{\min} - dist(e,i))$$

$$\gamma = 10^{-3}, \; N_e = \{ i \in N \mid dist(e,i) \le r_{\min} \}$$





No filtering



Filtering

▪ Python implementation: Optimaility criteria

```python
    # Optimality criteria
    xold[:]=x
    (x[:],g)=oc(nelx,nely,x,volfrac,dc,dv,g)
```

Call optimality criterion subroutine

```python
# Optimality criterion
def oc(nelx,nely,x,volfrac,dc,dv,g):
  l1=0
  l2=1e9
  move=0.2
  # reshape to perform vector operations
  xnew=np.zeros(nelx*nely)
  while (l2-l1)/(l1+l2)>1e-3:
    lmid=0.5*(l2+l1)
    xnew[:]= np.maximum(0.0,np.maximum(x-move,np.minimum(1.0,np.minimum(x+move,x*np.sqrt(-dc/dv/lmid)))))
    gt=g+np.sum((dv*(xnew-x)))
    if gt>0 :
      l1=lmid
    else:
      l2=lmid
  return (xnew,gt)
```

$$
x_e^{\text{new}} = \begin{cases} \max(x_{\min}, x_e - m) \\ \quad \text{if} \quad x_e B_e^\eta \le \max(x_{\min}, x_e - m)\,, \\ x_e B_e^\eta \\ \quad \text{if} \quad \max(x_{\min}, x_e - m) < x_e B_e^\eta < \min(1, x_e + m)\,, \\ \min(1, x_e + m) \\ \quad \text{if} \quad \min(1, x_e + m) \le x_e B_e^\eta\,, \end{cases}
$$

$$
\eta = \frac{1}{2};\ m = 0.2;
$$

$$
B_e = \frac{-\dfrac{\partial c}{\partial x_e}}{\lambda \dfrac{\partial V}{\partial x_e}}
$$

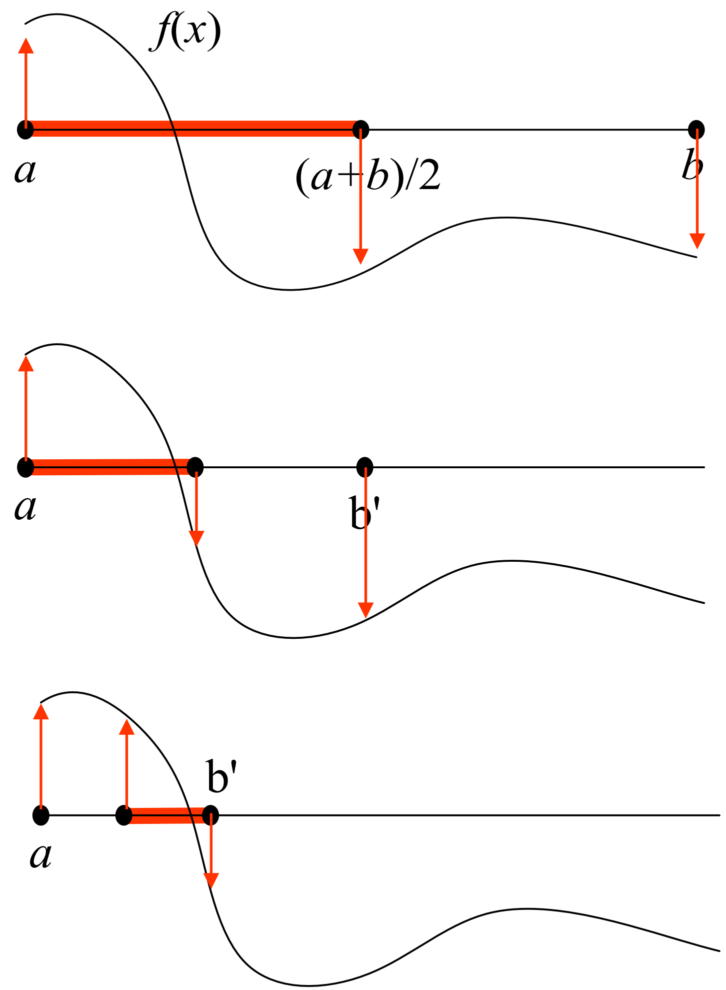
Lagrange multiplier λ must be chosen so that volume constraint is satisfied; the appropriate value can be found by bisection algorithm

KAIST

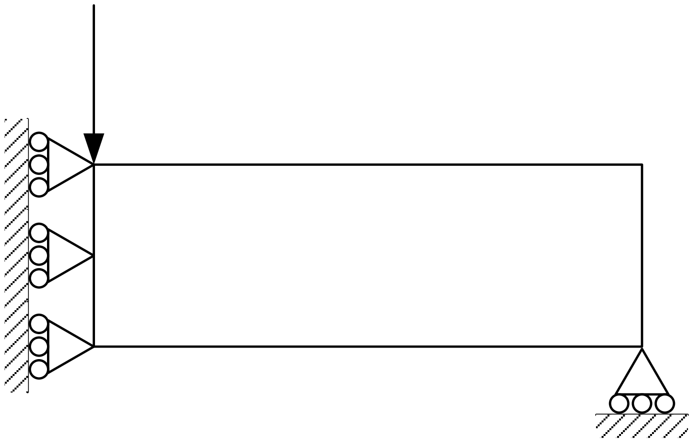
Innovative Design for Mobility Systems Laboratory

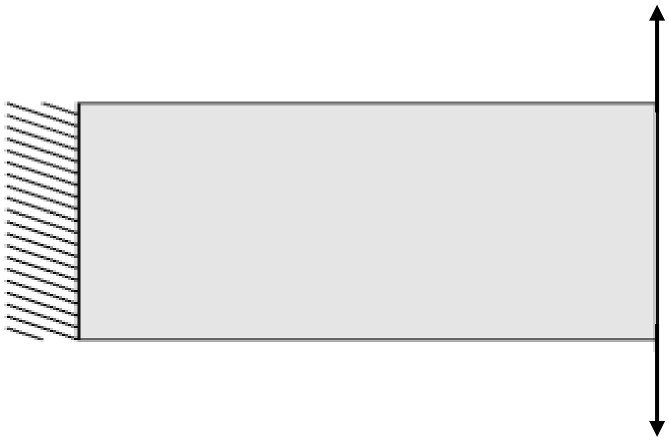
- **Python implementation: Optimaility criteria**

Bisection algorithm



- **Extensions:** **Other boundary condtions and muptiple load cases**





▪ Extensions: Other boundary condtions and muptiple load cases

Other boundary condtions

```
fixed=np.union1d(dofs[0:2*(nely+1):2],np.array([2*(nelx+1)*(nely+1)-1]))
```

↓

```
fixed=dofs[0:2*(nely+1)]
```

Multiple load cases

The objective function is now the sum of each load case,

$$c_{multiple} = \sum_{i=1}^{m} \mathbf{U}_i^{\ T} \mathbf{K} \mathbf{U}_i \quad \textit{where m means the number of load cases.}$$

First, declare force and displacement vectors to multi-column vectors and assign load values at target index.

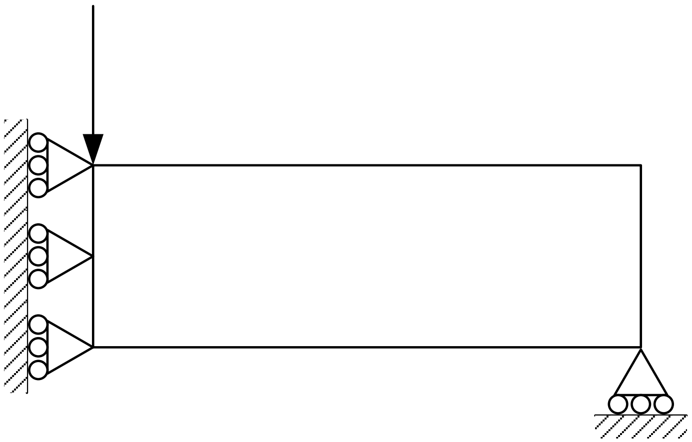
```
f=np.zeros((ndof,2))
u=np.zeros((ndof,2))
```

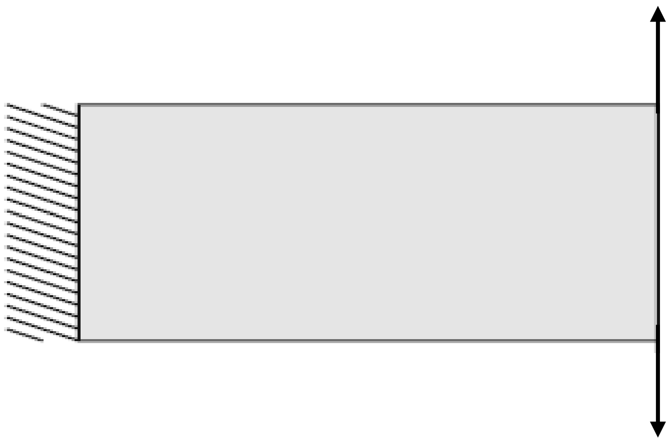
```
f[2*(nelx)*(nely+1)+1,0]=1
f[2*(nelx+1)*(nely+1)-1,1]=-1
```

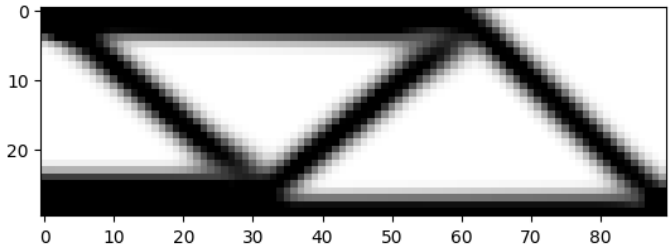
And then, change objective function and sensitivity values.

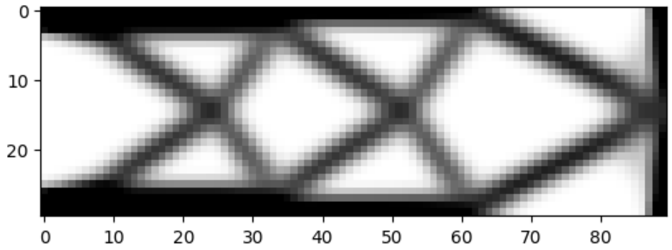
```
obj = 0
dc = np.zeros(nely*nelx)
for i in range(np.size(f,1)):
  ui = u[:,i]
  ce[:] = (np.dot(ui[edofMat].reshape(nelx*nely,8),KE) * ui[edofMat].reshape(nelx*nely,8) ).sum(1)
  obj = obj + ( (Emin+xPhys**penal*(Emax-Emin))*ce ).sum()
  dc[:] = dc + (-penal*xPhys**(penal-1)*(Emax-Emin))*ce
```

▪ Extensions: Other boundary condtions and muptiple load cases

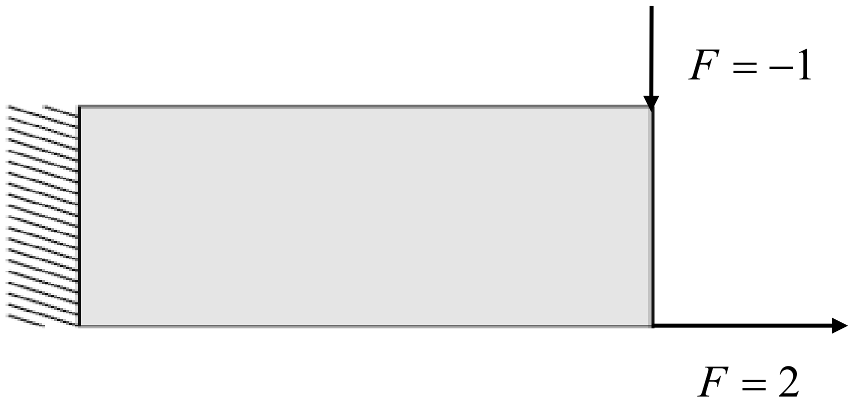






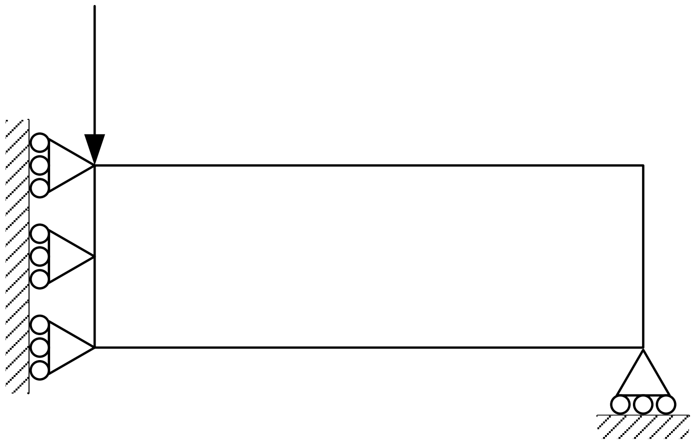


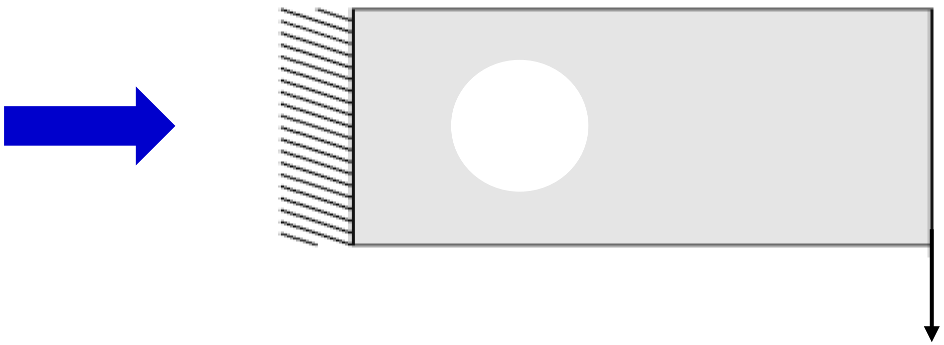
- **Exercise**



▪ Extensions: Passive element

- Center of void is `(nely/2, nelx/3)` and radius with `nely/3`





▪ Extensions: Passive element

Passive element

A `nely×nelx` matrix `passive` is defined with 0 at elements free to change, 1 at elements fixed to be void, And 2 at elements fixed to be solid.

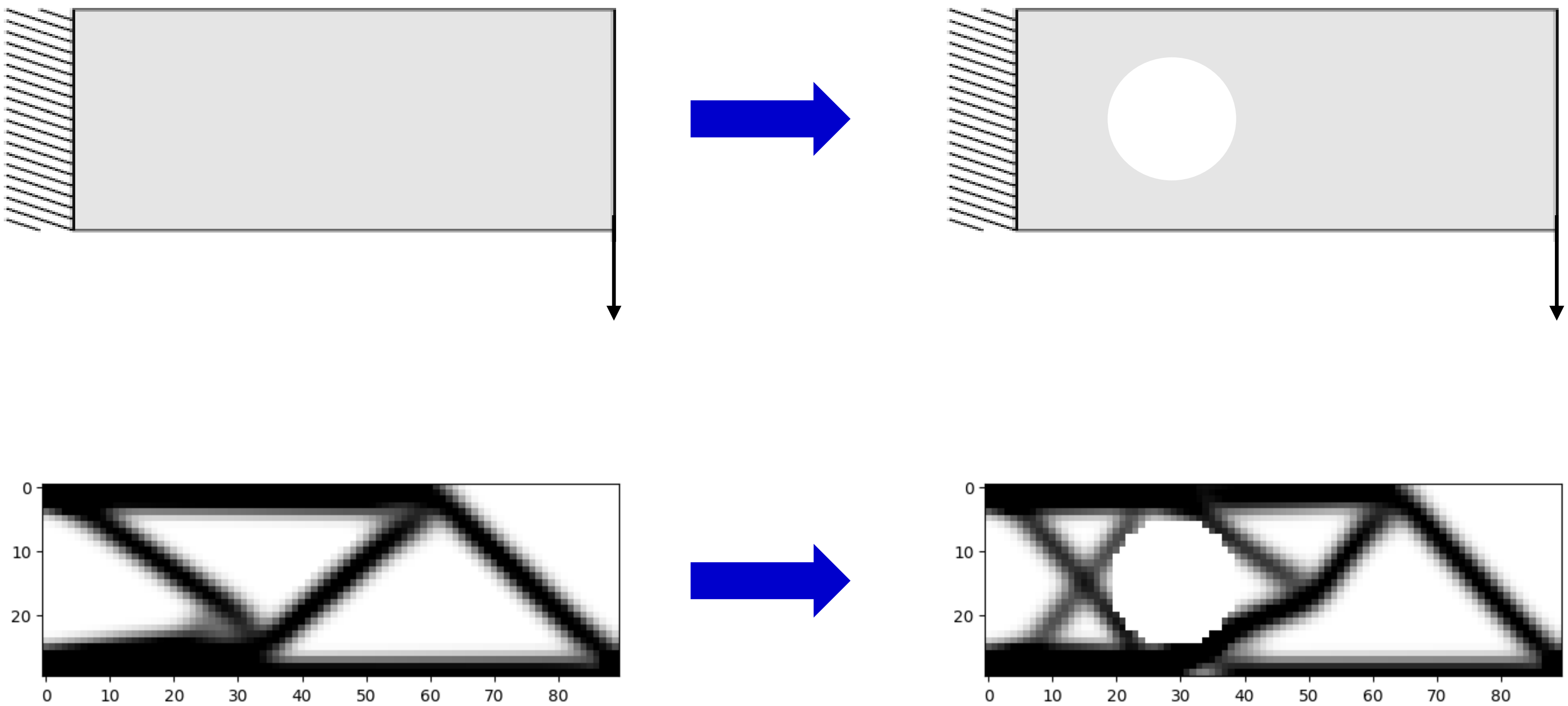
```
# Define Passive elements
passive = np.zeros((nely,nelx), dtype=int)
for i in range(nelx):
  for j in range(nely):
    if np.sqrt((j-nely/2+1)**2+(i-nelx/3+1)**2) < nely/3:
      passive[j,i] = 1
```

These lines must be inserted before the start of the optimization loop.

Then we can simply add two line in optimality criteria subroutines.

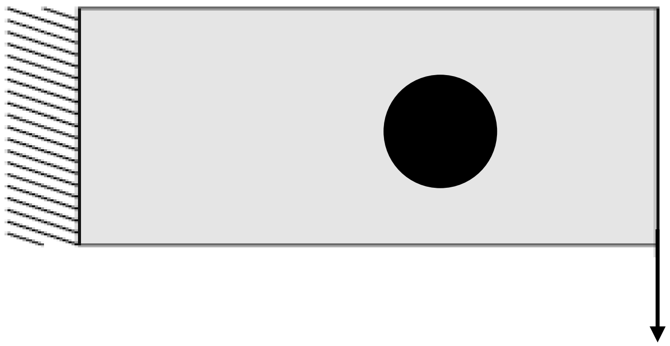
```
    xnew[nely*np.where(passive==1)[1]+np.where(passive==1)[0]+1] = 0
    xnew[nely*np.where(passive==2)[1]+np.where(passive==2)[0]+1] = 1
```

- **Extensions: Passive element**



Exercise

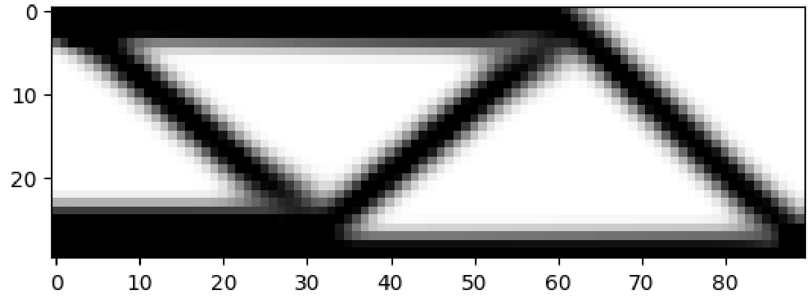
- Center of solid is `(nely/2, nely×2)`and radius with `nelx/10`



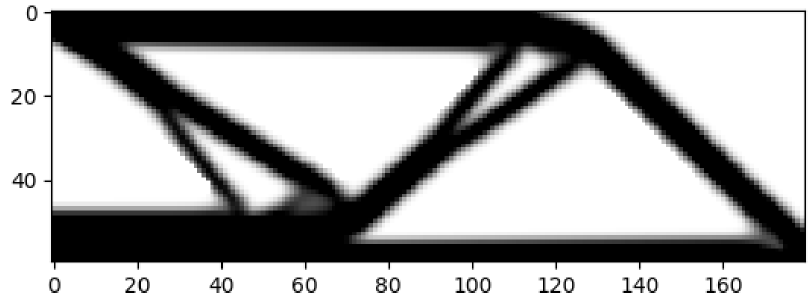
▪ Parameter test

- `main(90,30,0.4,3.0,3.0,1)`
- `main(180,60,0.4,3.0,3.0,1)` ← To check mesh-independency
- `main(90,30,0.4,3.0,0.1,1)` ← To check checkerboard pattern
- `main(90,30,0.7,3.0,3.0,1)` ← To check volume fraction ratio
- `main(90,30,0.5,1.0,3.0,1)` ← To check penalty term

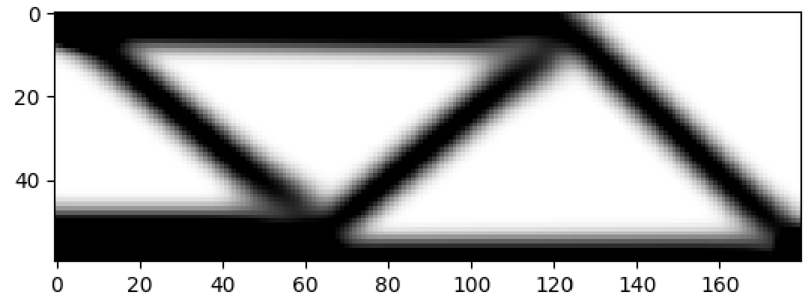
- **Parameter test: Mesh dependency**



➢ `main(90,30,0.4,3.0,1.5,1)`

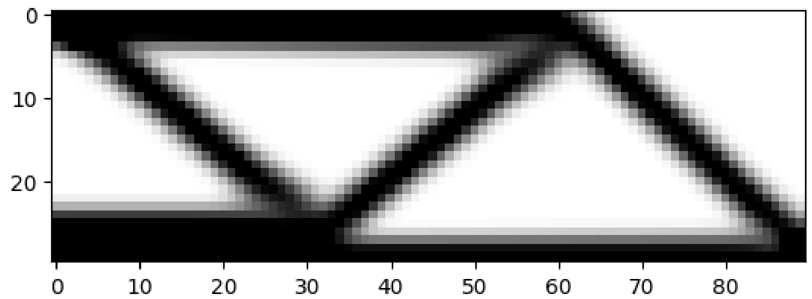


➢ `main(180,60,0.4,3.0,1.5,1)`

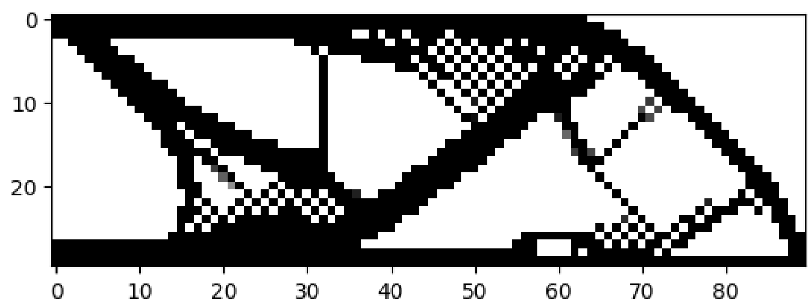


➢ `main(180,60,0.4,6.0,1.5,1)`

▪ Parameter test: Checkerboard pattern

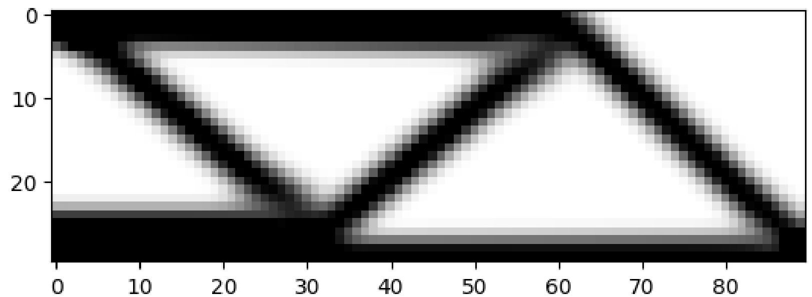


➢ `main(90,30,0.4,3.0,1.5,1)`

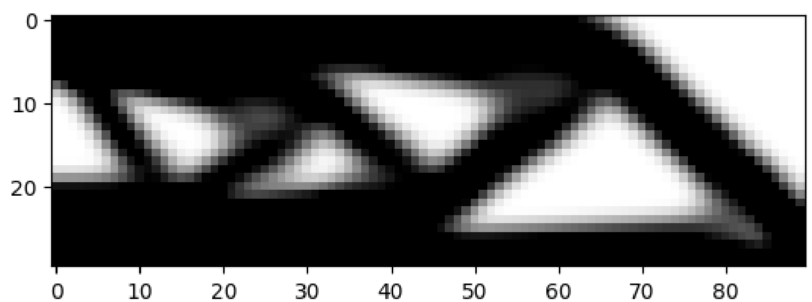


➢ `main(90,30,0.4,3.0,0.1,1)`

- **Parameter test: Volume fraction ratio**

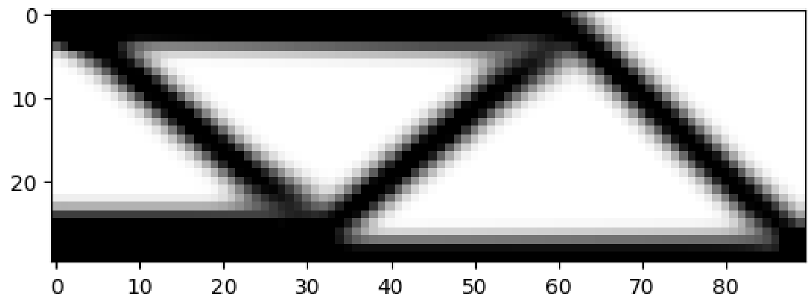


➢ main(90,30,0.4,3.0,1.5,1)

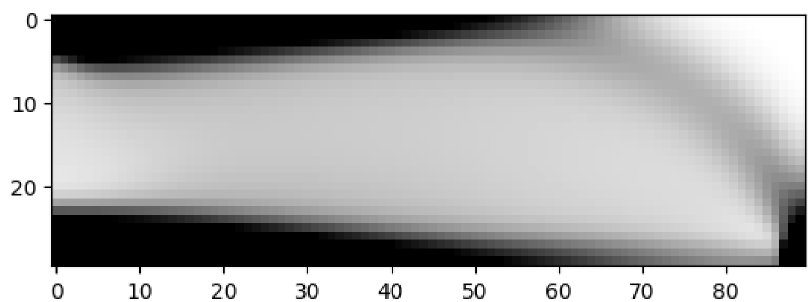


➢ main(90,30,0.7,3.0,0.1,1)

- **Parameter test: Penalty term**



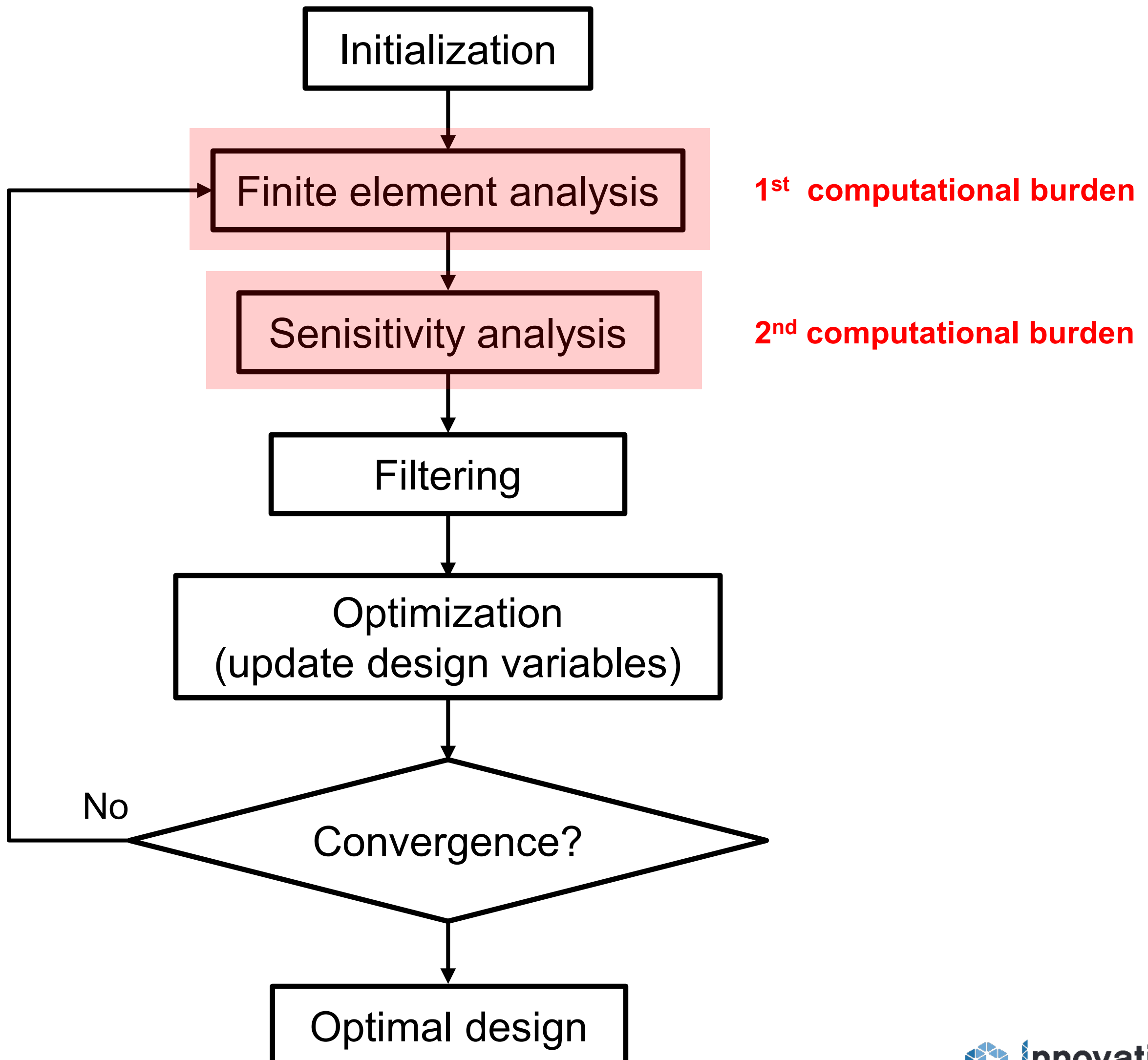
➢ `main(90,30,0.4,3.0,1.5,1)`



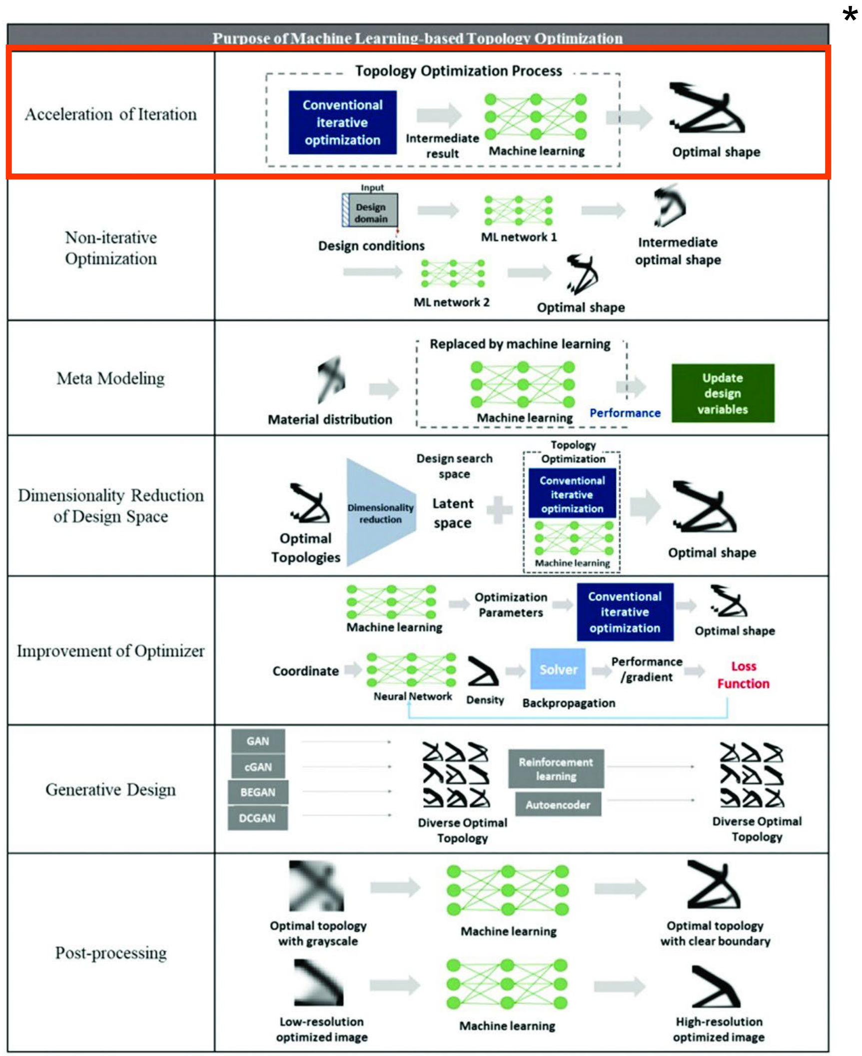
➢ `main(90,30,0.4,1.0,3.0,1)`

Acceleration of topology optimization using artificial intelligence

- **Acceleration of topology optimization using artificial intelligence**



▪ Acceleration of topology optimization using artificial intelligence



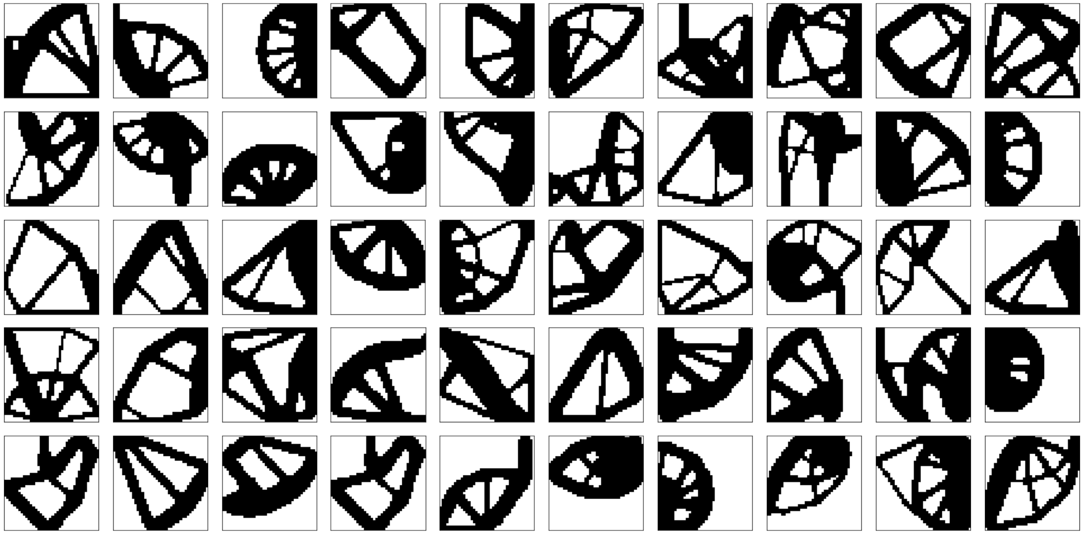
*S.Shin, D. Shin, N. Kang, "Topology optimization via machine learning and deep learning: a review", *Journal of Computational Design and Engineering*, vol. 10, no. 4, pp. 1736-1766, 2023.

▪ Dataset

- The number of nodes with fixed x and y translations and the number of loads are sampled from the Poisson distribution:

$$N_x \sim P(\lambda = 2)$$
$$N_y, N_{\mathrm{L}} \sim P(\lambda = 1).$$

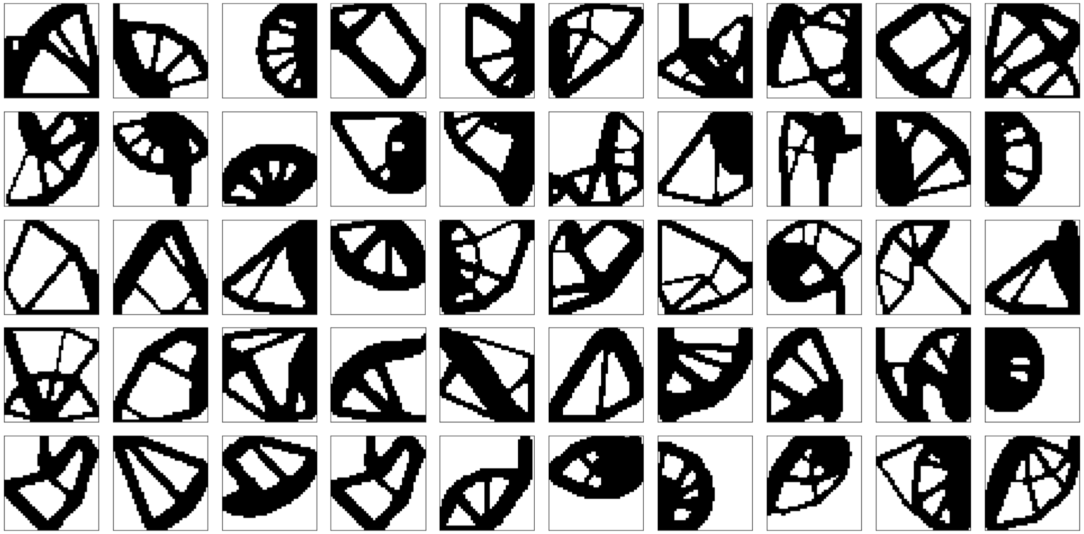
- The nodes for each of the above-described constraints are sampled from the distribution defined on the grid. The probability of choosing the boundary node is 100 times higher than that for an inner node.


*

*I. Sosnovik, I. Oseledets, "Neural networks for topology optimization", *Russian Journal of Numerical Analysis and Mathematical Modelling*, vol.34, no. 4, pp. 215-223, 2019.

▪ Dataset

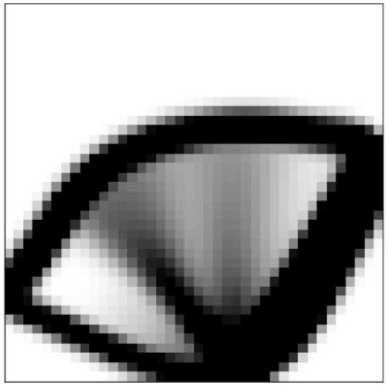
- The load values are chosen as –1.
- The volume fraction is sampled from the Gaussian distribution with mean of 0.5 and std of 0.1
- The obtained dataset has 10,000 objects. 7,000 objects are allocated for training and 3,000 objects are allocated for testing.
- Each object is a tensor of shape 2×40×40: 7, 100 iterations of the optimization process for the problem defined on a regular 40×40 grid.


*

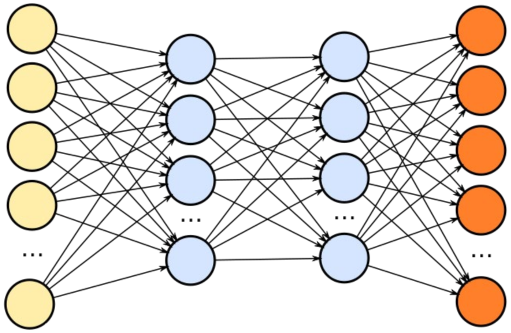
*I. Sosnovik, I. Oseledets, “Neural networks for topology optimization”, *Russian Journal of Numerical Analysis and Mathematical Modelling*, vol.34, no. 4, pp. 215-223, 2019.

▪ Deep learning architecture

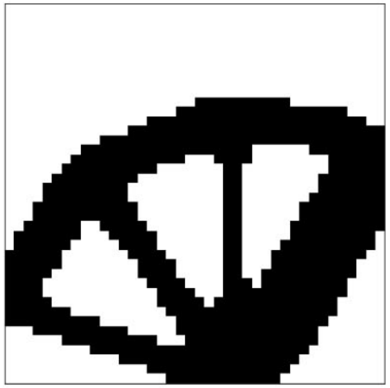
- PyTorch is used as a deep learning framework.
- Training is conducted with 20 fixed epochs, 16 batch size, MAE loss, Adam optimizer.



Intermediate result
(iter 7)

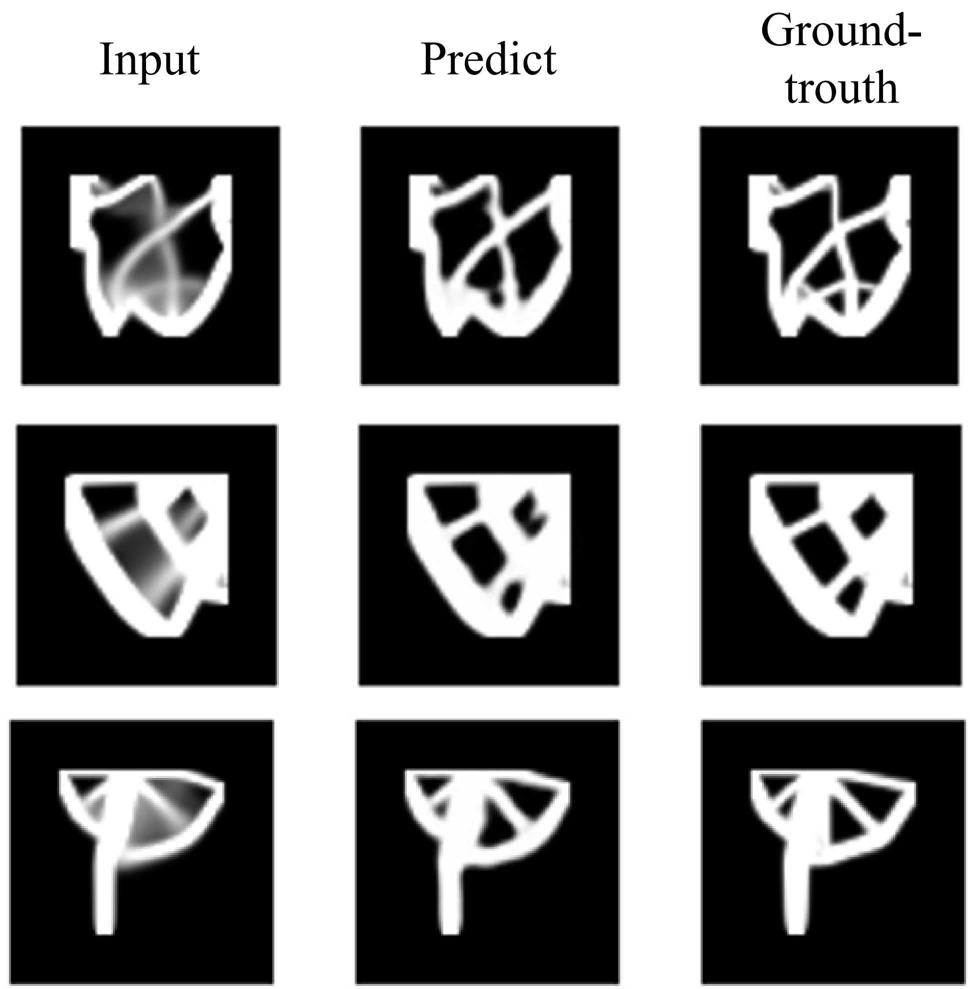


Deep neural network
(Unet)



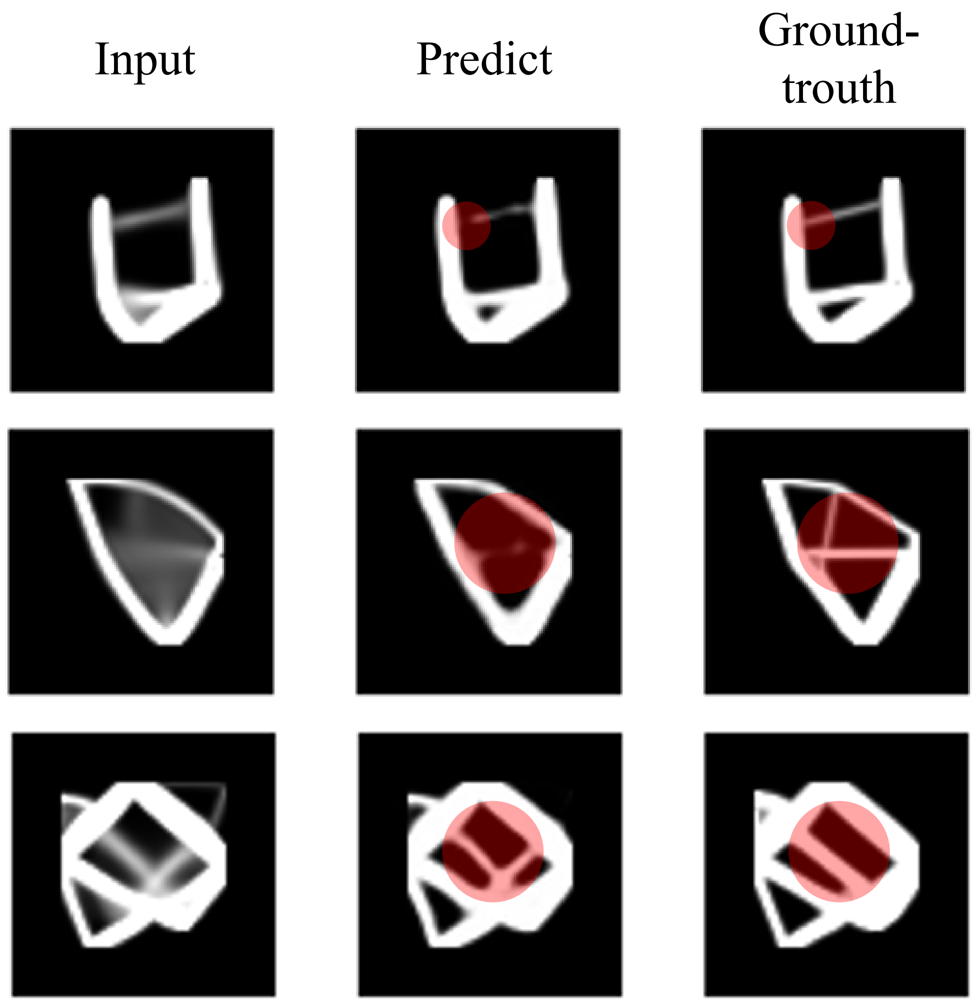
Ground trouth
(iter 100)

▪ Result



*S.Shin, D. Shin, N. Kang, "Topology optimization via machine learning and deep learning: a review", *Journal of Computational Design and Engineering*, vol. 10, no. 4, pp. 1736-1766, 2023.

▪ Result



*S.Shin, D. Shin, N. Kang, "Topology optimization via machine learning and deep learning: a review", *Journal of Computational Design and Engineering*, vol. 10, no. 4, pp. 1736-1766, 2023.

Thank you!

- Contact: hyukjin.koh@kaist.ac.kr

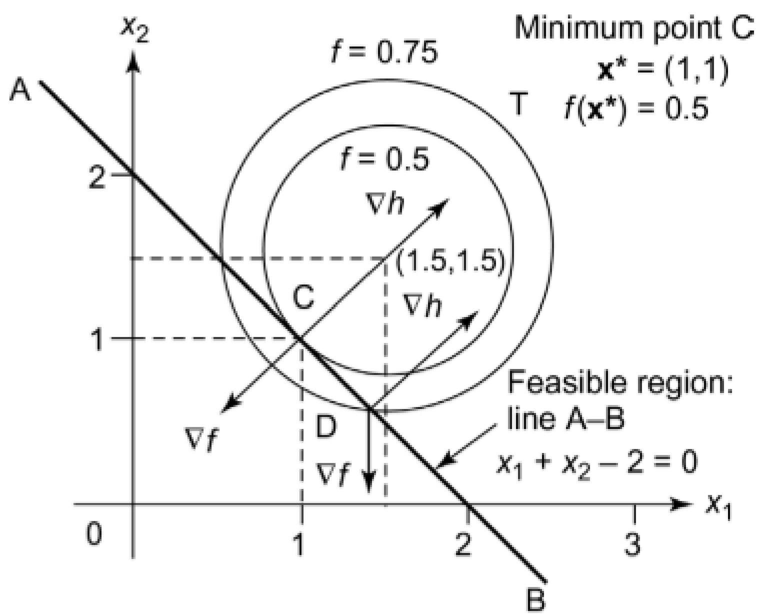
▪ Lagrange multiplier

Minimize

$$f(x_1, x_2) = (x_1 - 1.5)^2 + (x_2 - 1.5)^2$$

subject to an equality constraint:

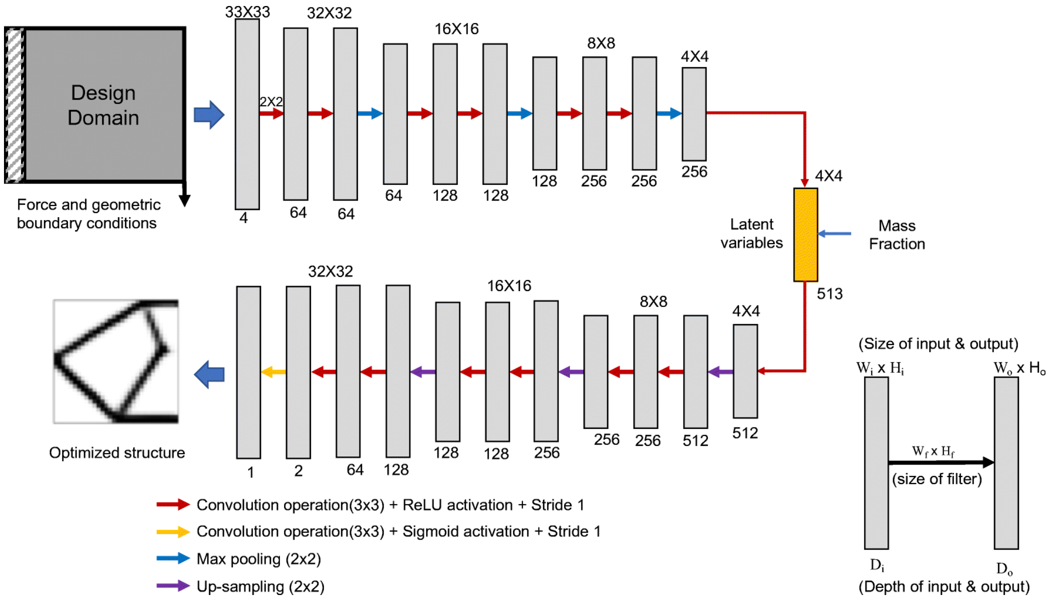
$$h(x_1, x_2) = x_1 + x_2 - 2 = 0$$



*Yu Y, Hur T, Jung J, Jang IG (2019) Deep learning for determin-ing a near-optimal topological design without any iteration. Struct Multidisc Optim 59(3):787–799. https:// doi. org/ 10. 1007/s00158- 018- 2101-5

▪ Direct design

- The direct design model approach is currently one of the most popular applications of AI in TO, and the aim is to directly achieve an optimised structure for a given problem definition, **completely removing the need for expensive iterative procedures**.
- Commonly this is achieved by implementing neural network architectures popular in image segmentation, like CNN or GAN.

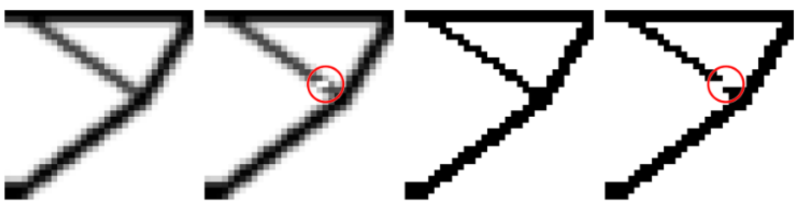


*Yu Y, Hur T, Jung J, Jang IG (2019) Deep learning for determin-ing a near-optimal topological design without any iteration. Struct Multidisc Optim 59(3):787–799. https:// doi. org/ 10. 1007/s00158- 018- 2101-5

▪ Direct design: limitations

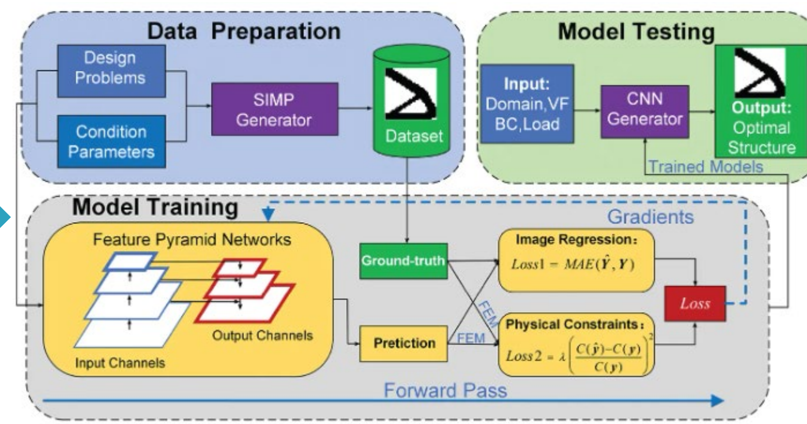
1. Mesh dependency
   - The presented direct design models are typically **restricted to a small fixed mesh**.
   - It is unclear whether the model is readily translated to problems with different mesh dimensions or resolutions, even given the inherent flexibility of the CNN.
   - The network size increases with the number of elements in the mesh, resulting in more parameters to be determined during training and a larger memory consumption for storing the model. In turn, this also increases the cost of obtaining training samples, as finer meshes imply **more time needed to optimize a structure using conventional TO**.

2. Structural disconnection
   - Image-based errors (e. g. MAE) not reflect the quality of a structure and thus the network learns based on an incorrect measure.



(a) Grey scale connected structure (compliance 90.04). **(b)** Grey scale disconnected structure (compliance 101.63). **(c)** Black-white connected structure (compliance 66.16). **(d)** Black-white disconnected structure (compliance 199.46).

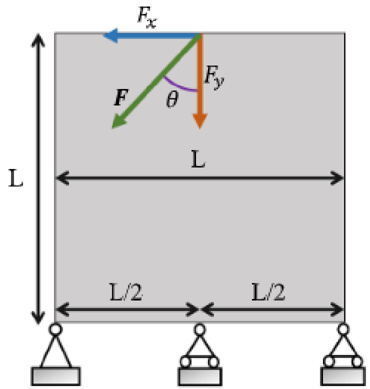
	Grey scale	Black-and-white
MAE	0.0014	0.0039
Gap	0.1288	2.0148



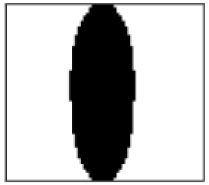
*Luo J, Li Y, Zhou W, Gong Z, Zhang Z, Yao W (2021) An Improved Data-Driven Topology Optimization Method Using Feature Pyramid Networks with Physical Constraints. Comput Model Eng Sci 128(3):823–848. https:// doi. org/ 10. 32604/ cmes. 2021.016737, https:// www. techs cience. com/ CMES/ v128n3/ 44011

▪ Direct design: limitations

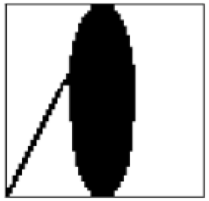
3. Vulnerability to perturbation (major)
    - Relatively small changes to the boundary conditions can lead to a very different solution being optimal.
    - Any learning-based approach would face the challenge that a small perturbation of the boundary conditions could lead to a big change in the optimal structure.
    - Unless the types of problems are strictly limited, it is clear that an **unbounded number of examples could be necessary** to learn all the discontinuities in the mapping from latent space to mechanical structure.



(a) Load magnitude $|\mathbf{F}|=1$, volume fraction of 0.3, 64x64 elements, filter radius 2.0



(b) $\theta=0$



(c) $\theta=0.01$

Design	(b)	(c)
$\theta=0$	7.255	7.372
$\theta=0.01$	1,649,351,760	7.5591

Compliance