PSI plane-strain example

This notebook is a companion to “Phase-space iterative solvers”, it reproduces the results in section “Plate with hole”.

J. Garcia-Suarez, 2025

Parameters of this study

Material

Intact Material: $E = 200^{9}$ Pa and ν=0.33

In[1]:=
```
youngModIntact = 200.*10^9;
nu = 0.33;
(*function*)
p = 2.0*10^-4;
c = (1/p)^(1/(p-1));
youngMod[x_] = p*youngModIntact*((Abs[x]+c)^(p-1));
youngModTan[x_] = p*(p-1)*youngModIntact*((Abs[x]+c)^(p-2));
```

Convergence

Convergence tolerance: the step has converged once

$|f_{ext} - f_{int}| / |f_{ext}| < tol$

In[7]:=
```
tol = 1 * 10^-2; (*in terms of eq.*)
```

Phase-space convergence tolerance: the step has converged once $\left| z^{(n+1)} - z^{(n)} \right| / \left| z^{(n)} \right| <$ tolPS

In[8]:=
```
tolPS = 1 * 10^-3; (*in terms of phase-space distance*)
```

PSI

At this point I define the parameters of the phase-space solver.

In[9]:=
```
(*method*)
approach = "minimization";(* "E-L" or "minimization"*)
optimalC = "False";
(*method in the case of minimization*)
methodMin = "ConjugateGradient";
(*"ConjugateGradient", "Newton", "QuasiNewton",
"PrincipalAxis"*)
(*number of tests PSI*)
Ntests = 1;
(*Constant for the distance function*)
Cd = 0.1 * youngModIntact;
(*Number of CPUs*)
numKernels = 4;
```

NR

In[15]:=
```
(*number of tests NR*)
NtestsNR = 1;
```

In[16]:=
```
numLoadSteps = 1;
(*number of steps we use to derive the load*)
```

Pre-processing

Create mesh. Begin by loading FEM tools:

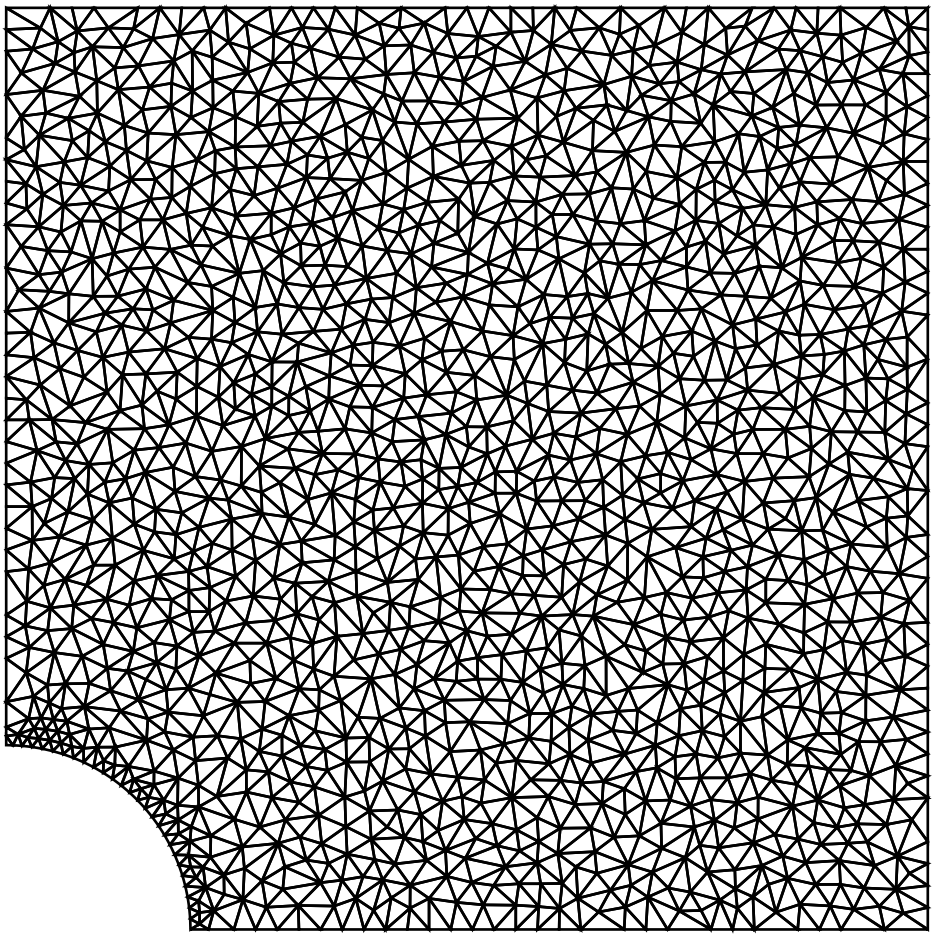
In[17]:=
```
Needs["NDSolve`FEM`"];
```

Define region (exploiting symmetry), see Fig.1a in the paper.

In[18]:=
```
r = 0.02; (*hole radius*)
```

In[19]:=
```
region = RegionDifference[
    Polygon[{{0, 0}, {0., 0.1}, {0.1, 0.1}, {0.1, 0}}],
    Disk[-{0.0, 0.0}, r]];
mesh = ToElementMesh[region,
    MaxCellMeasure → {"Length" → 0.005}, "MeshOrder" → 1,
    AccuracyGoal → 5];
Show[
 mesh["Wireframe"]
 , Background → White]
```

Out[21]=



Mesh properties

Use Mathematica's built-in tools to extract information

In[22]:=
```
nodes = mesh["Coordinates"];(*nodes in the mesh*)
numNodes = Length@nodes;(*number of nodes*)
connectivity = mesh["MeshElements"][[1]][[1]];
(*element connectivity*)
numElements = Length@connectivity;
(*number of elements*)
numDOFs = 2;(*dofs per node: u and v*)
(*for each element,
to what dofs contributes to w/ forces
 this is consistent w/ the way in which the B
 matrives are constructed*)
locDOFs =
  Table[
   Flatten[{connectivity[[ii]],
     numNodes + connectivity[[ii]]}], {ii, 1, numElements}];
```

Identify fixed nodes and loaded (to apply BCs)

In[28]:=
```
lowerEdge = {};
Do[
  If[nodes[[ii]][[2]] ≤ 0.0005, AppendTo[lowerEdge, ii]]
  , {ii, 1, numNodes}];
upperEdge = {};
Do[
  If[nodes[[ii]][[2]] == 0.1, AppendTo[upperEdge, ii]]
  , {ii, 1, numNodes}];
leftEdge = {};
Do[
  If[nodes[[ii]][[1]] ≤ 0.001, AppendTo[leftEdge, ii]]
  , {ii, 1, numNodes}];
```

2D problem: define a thickness

In[34]:=
```
thickness = 1.;
```

Plane strain operator for Mathematica pre-processing

Loading: constant traction at the upper edge

In[35]:=
```
ps = 10^8;(*Pa/m (all units SI)*)
tractions = NeumannValue[ps, y == 0.1];
```

Plane-strain operator for linear-elastic homogeneous isotropic material:

In[37]:=
```
planeStrainOperator =
  {Inactive[Div][
      ({{0, -((Y ν)/((1 - 2 ν) (1 + ν)))}, {-(Y/(2 (1 + ν))), 0}}.
        Inactive[Grad][v[x, y], {x, y}]), {x, y}] +
    Inactive[Div][
     ({{-((Y (1 - ν))/((1 - 2 ν) (1 + ν))), 0}, {0, -(Y/(2 (1 + ν)))}}.
        Inactive[Grad][u[x, y], {x, y}]), {x, y}],
   Inactive[Div][
     ({{0, -(Y/(2 (1 + ν)))}, {-((Y ν)/((1 - 2 ν) (1 + ν))), 0}}.
        Inactive[Grad][u[x, y], {x, y}]), {x, y}] +
    Inactive[Div][
     ({{-(Y/(2 (1 + ν))), 0}, {0, -((Y (1 - ν))/((1 - 2 ν) (1 + ν)))}}.
        Inactive[Grad][v[x, y], {x, y}]), {x, y}]} /.
  {Y → 200. * 10^9, ν → 33./100};
```

BCs:

In[38]:=
```
(* held fixed at left *)
bcs = {
   DirichletCondition[u[x, y] == 0, x ≤ 0],
   DirichletCondition[v[x, y] == 0, y ≤ 0]};
```

Define PDE:

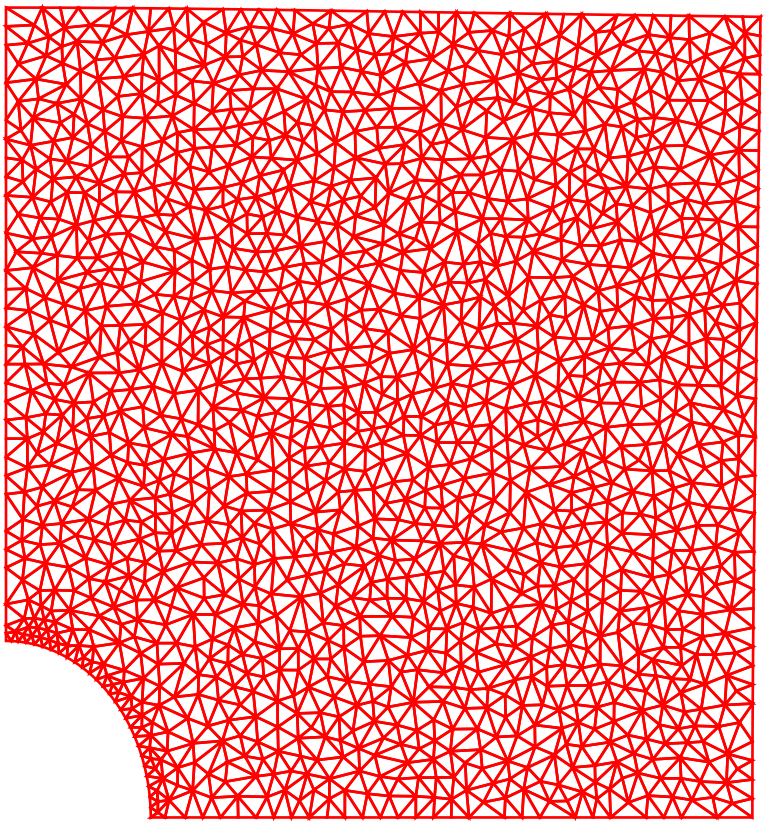
In[39]:=
```
pde2D = planeStrainOperator == {0, 1. * tractions};
```

Mathematica linear-elastic solution (shown for completeness for other users to play with if interested, not necessary):

In[40]:=
```
{usol, vsol} = NDSolveValue[{pde2D, bcs}, {u, v},
   {x, y} ∈ mesh];
```

In[41]:=
```
mesh = usol["ElementMesh"];
Show[
 {mesh["Wireframe"["MeshElementStyle" → EdgeForm[White]]],
  ElementMeshDeformation[mesh, {usol, vsol},
    "ScalingFactor" → 100][
   "Wireframe"["ElementMeshDirective" →
     Directive[EdgeForm[Red], FaceForm[]]]]},
 ImageSize → 300]
```

Out[42]=



Pre-processing: solve linear-elastic problem

Define intact material

In[43]:=
```
(*Intact plane-strain tensor*)
```

$$(*\begin{pmatrix}\sigma_{11}\\ \sigma_{22}\\ \tau_{12}\end{pmatrix}=\begin{pmatrix}\lambda+2\mu & \lambda & 0\\ \lambda & \lambda+2\mu & 0\\ 0 & 0 & \mu\end{pmatrix}\begin{pmatrix}\epsilon_{11}\\ \epsilon_{22}\\ \gamma_{12}\end{pmatrix}*)$$

```
matCIntact = (({{2 + 2 nu/(1 - 2 nu), 2 nu/(1 - 2 nu), 0},
      {2 nu/(1 - 2 nu), 2 + 2 nu/(1 - 2 nu), 0},
      {0, 0, 1}} * youngModIntact/(2 (1 + nu)));
```

Preprocessing: take advantage of Mathematica to distribute the load to the nodes

In[44]:=
```
nr = ToNumericalRegion[mesh];
vd = NDSolve`VariableData[
    {"DependentVariables", "Space"} → {{u, v}, {x, y}}];
sd = NDSolve`SolutionData[{"Space"} → {nr}];
(*We use NDSolve as a pre-processor:*)
{state} =
 NDSolve`ProcessEquations[{pde2D, bcs}, {u, v},
    {x, y} ∈ mesh];
(*Extract the finite element data:*)
femdata = state["FiniteElementData"];
initBCs = femdata["BoundaryConditionData"];
methodData = femdata["FEMMethodData"];
initCoeffs = femdata["PDECoefficientData"];
(*discretize*)
discretePDE = DiscretizePDE[initCoeffs, methodData,
    sd, "SaveFiniteElements" → True,
    "AssembleSystemMatrices" → True];
discreteBCs = DiscretizeBoundaryConditions[initBCs,
    methodData, sd];
(*Extract the system matrices:*)
load = discretePDE["LoadVector"];
stiffness = discretePDE["StiffnessMatrix"];
stiffnessBeforeBCs = stiffness;
DeployBoundaryConditions[{load, stiffness},
  discreteBCs];
```

Construct matrices B

I need to do this manually although most probably there is a way to get the same using **NDSolve**

In[58]:=
```
elCentroids = RegionCentroid[Polygon[nodes[[#]]]] & /@
   connectivity;
elAreas = ConstantArray[0., {numElements, 1}];
elMatB = ConstantArray[0., {numElements, 1}];
Do[
 (*relevant nodal coordinates
  (to compute the coefficients of B)*)
 node1 = nodes[[connectivity[[ii]][[1]]]];
 node2 = nodes[[connectivity[[ii]][[2]]]];
 node3 = nodes[[connectivity[[ii]][[3]]]];
 (*Compute area*)
 elArea = Area@Polygon[nodes[[connectivity[[ii]]]]];
 elAreas[[ii]] = elArea;
 (*compute the B matrix of the
   element--------------------*)
 matB = ConstantArray[0., {3, 6}];
 matB[[1, 1]] = 1/(2*elArea) (Last@node2 - Last@node3);
 matB[[1, 2]] = 1/(2*elArea) (Last@node3 - Last@node1);
 matB[[1, 3]] = 1/(2*elArea) (Last@node1 - Last@node2);
 matB[[2, 4]] = 1/(2*elArea) (First@node3 - First@node2);
 matB[[2, 5]] = 1/(2*elArea) (First@node1 - First@node3);
```

```
matB[[2, 6]] = 1/(2*elArea) (First@node2 - First@node1);
matB[[3, 4]] = 1/(2*elArea) (Last@node2 - Last@node3);
matB[[3, 5]] = 1/(2*elArea) (Last@node3 - Last@node1);
matB[[3, 6]] = 1/(2*elArea) (Last@node1 - Last@node2);
matB[[3, 1]] = 1/(2*elArea) (First@node3 - First@node2);
matB[[3, 2]] = 1/(2*elArea) (First@node1 - First@node3);
matB[[3, 3]] = 1/(2*elArea) (First@node2 - First@node1);
elMatB[[ii]] = matB;
, {ii, 1, numElements}]
```

Construct the K matrix

Element-wise contributions:

In[62]:=
```
elMatK =
  Table[Transpose[elMatB[[ii]]].matCIntact.elMatB[[ii]]*
    elAreas[[ii]], {ii, 1, numElements}];
```

Assemble:

In[63]:=
```
totalList = ConstantArray[0., {numElements, 1}];
Do[
  (*If[Mod[kk,100]==0,Print[kk]];*)
  subList = Table[
    {If[ii ≤ 3, connectivity[[kk]][[ii]],
        connectivity[[kk]][[ii - 3]] + numNodes],
      If[jj ≤ 3, connectivity[[kk]][[jj]],
       connectivity[[kk]][[jj - 3]] + numNodes]}
     → elMatK[[kk]][[ii, jj]],
    {ii, 1, 3*numDOFs}, {jj, 1, 3*numDOFs}];
  totalList[[kk]] = Flatten[subList, 1];
  , {kk, 1, numElements}];
SetSystemOptions[
  "SparseArrayOptions" →
   {"TreatRepeatedEntries" → Total}];
globalK = SparseArray[Flatten[totalList, 1],
   {numDOFs*numNodes, numDOFs*numNodes}];
SetSystemOptions["SparseArrayOptions" →
   {"TreatRepeatedEntries" → 0}];
```

Apply BCs

First, the horizontal symmetry plane:

In[68]:=
```
restrainedDOFsY =
  Sort[Flatten[{# + numNodes} & /@ lowerEdge]];
activeDOFs = DeleteCases[Range[numDOFs*numNodes],
   Alternatives @@ restrainedDOFsY];
```

Next the vertical symmetry plane:

In[70]:=
```
restrainedDOFsX = Sort[Flatten[{#} & /@ leftEdge]];
activeDOFs = DeleteCases[activeDOFs,
    Alternatives @@ restrainedDOFsX];
```

Solve linear-elastic solution using matrices that will later be used PSI

Just for sanity check purposes: this yields the same than using **NDSolve** above

In[72]:=
```
forceVecExt = load;
intactU = SparseArray[{}, {numNodes*numDOFs, 1}];
intactU[[activeDOFs]] =
  LinearSolve[globalK[[activeDOFs, activeDOFs]],
   forceVecExt[[activeDOFs]]];
```

Solve using Newton-Raphson method

The reason to use NR:
it is the main reference to compare PSI against

Mean stress function (necessary to define the threshold):

Introducing the mean strain

I use the mean strain to gauge how much the stiffness changes as strain accumulates (recall, plane strain -> ϵ_{33}=0)

In[75]:=
```
epsilonMean[ϵ_] := (ϵ[[1, 1]] + ϵ[[2, 1]])/3;
```

The loop

Initialize the rest

In[76]:=
```
timeListNR = Table[, NtestsNR];
Do[
 (*initialize material stiffness to the intact
  stiffness*)
 matCList = Table[matCIntact, {ii, 1, numElements}];
 matT = SparseArray[{},
   {numNodes*numDOFs, numNodes*numDOFs}];
 Unr = SparseArray[{}, {numNodes*numDOFs, 1}];
 (*loop ------------------------------------------------
                        ---------------------------------
     -----*)
 timeNR = AbsoluteTiming[
   Do[
     forceVecExt = ff/numLoadSteps load;
     normForceExt = Norm[forceVecExt[[activeDOFs]]];
     Print["*--------------------------*"];
     Print["Load step # " <> ToString[ff] <>
       " out of " <> ToString[numLoadSteps]];
     Print["*--------------------------*"];
     (*prepare for the iterations at that load level*)
     steps = 1;
     resError = 1.;
     done = False;
     While[done == False,
```

```
forceVecInt = SparseArray[{},
  {numNodes * numDOFs, 1}]; (*initialize*)
matT = 0. * matT;
(*build stiffness matric fot this
 step ----------------------------------------
                          ----------------------------
      --------*)
stressList = ConstantArray[0., {numElements, 1}];
strainList = ConstantArray[0., {numElements, 1}];
Do[
 (*displacements in the relevant nodes*)
 Ue = Unr[[locDOFs[[ii]]]];
 matB = elMatB[[ii]];
 (*element strains*)
 strain = matB.Ue;
 strainList[[ii]] = strain;
 (*material stiffnesses*)
 matC = matCList[[ii]];
 (*element stresses*)
 stress = matC.strain;
 (*update element tangent matrix for next
  iteration*)
```

$$\text{matCList}[[ii]] = \frac{\text{youngMod}\left[\frac{\text{epsilonMean@strain}}{1.00}\right]}{2\,(1+\text{nu})}$$

$$\left\{\left\{2+\frac{2\,\text{nu}}{1-2\,\text{nu}},\ \frac{2\,\text{nu}}{1-2\,\text{nu}},\ 0\right\},\ \left\{\frac{2\,\text{nu}}{1-2\,\text{nu}},\ 2+\frac{2\,\text{nu}}{1-2\,\text{nu}},\ 0\right\},\ \{0,\ 0,\ 1\}\right\};$$

```
 (*matCList[[ii]]=youngMod[sigmaMean@stress]
                  1-nu^2
```

```
    ({{1,nu,0},{nu,1,0},{0,0,(1-nu)/2}});*)

 (*take the average betwen two steps,
 this boosts convergence*)
 matC = 0.5 (matC + matCList[[ii]]);
 stress = matC.strain;
 stressList[[ii]] = stress;
 (*save history*)
 (*internal force contribution*)
 forceVecInt[[locDOFs[[ii]]]] =
  Transpose[elMatB[[ii]]].stress * elAreas[[ii]] *
    thickness + forceVecInt[[locDOFs[[ii]]]];
 (*contribution to the stiffness matrix*)

 matT[[locDOFs[[ii]], locDOFs[[ii]]]] =
  matT[[locDOFs[[ii]], locDOFs[[ii]]]] +
   Transpose[elMatB[[ii]]].matC.elMatB[[ii]] *
    elAreas[[ii]];
 , {ii, 1, numElements}];
deltaF = SparseArray[forceVecExt - forceVecInt];
(*-----------------------------------------------
                        ------------------------
                        ----*)
resError = Norm[deltaF[[activeDOFs]]] /
  normForceExt;
deltaU = SparseArray[{}, Dimensions@Unr];
deltaU[[activeDOFs]] =
 LinearSolve[matT[[activeDOFs, activeDOFs]],
  deltaF[[activeDOFs]]];
Unr[[activeDOFs]] = Unr[[activeDOFs]] +
```

```
    deltaU[[activeDOFs]];
   (*-------------------------------------------------
                              ------------------------------
                              ----*)
   If[Mod[steps, 25] == 0,
    Print["Step #" <> ToString[steps] <>
       "  Log Res. Error (Force):" <>
       ToString[Log10@resError]];]
    (*-------------------------------------------------
                              ------------------------------
                              ----*)×
    If[steps > 500,
     Print["Failed to converge in " <>
       ToString[steps] <> " steps"]; Break[]];
   If[resError < tol, (*done yet?*)
    Print["Converged in " <> ToString[steps] <>
      " steps"];
    done = True, (*yes*)
    steps = steps + 1;(*next step*)
   ](*done!*)
   (*-------------------------------------------------
                              ------------------------------
                              ----*)
  ];,
  {ff, 1, numLoadSteps}];];
 timeListNR[[kk]] = First@timeNR;
 , {kk, 1, NtestsNR}]
```

```
*-------------------------*
Load step # 1 out of 1
```

```
*--------------------------*
Converged in 12 steps
```

PSI

Solve the same project using phase-space iterations

Preliminaries

In[78]:=
```
methodC = matCIntact;
methodCinv = Inverse[methodC];
methodK = globalK;
elMatC = elMatK;
(*elementary K matrix -- before assembling*)
```

Define material

A bit redundant, but I define the constitutive law again here.
This is because, due to my lack of consistency, the strain is taken as a matrix in NR and as vector in PSI.

σ = material(ϵ)

In[82]:=
```
epsilonMean[ϵ_] := (ϵ[[1]] + ϵ[[2]])/3;
```

This seems clogged but it is just Eq.(18) in the paper

In[83]:=
$$\text{material}[\text{eps}\_] := \frac{\text{youngMod}\left[\frac{\text{epsilonMean@eps}}{1.}\right]}{2\,(1+\text{nu})}\left\{\left\{2+\frac{2\,\text{nu}}{1-2\,\text{nu}},\ \frac{2\,\text{nu}}{1-2\,\text{nu}},\ 0\right\},\ \left\{\frac{2\,\text{nu}}{1-2\,\text{nu}},\ 2+\frac{2\,\text{nu}}{1-2\,\text{nu}},\ 0\right\},\ \{0,\ 0,\ 1\}\right\}.\text{eps};$$

Iterate

Initialize kernels for parallel projections into the material law

In[84]:=
```
CloseKernels[];
LaunchKernels[numKernels];
```

Looping time

The commented lines below are to change some parameters of the method
w/o having to go all the way to the top of the notebook.

In[86]:=
```
2
```

Out[86]= 2

In[87]:=
```
(*(*method in the case of minimization*)
methodMin="ConjugateGradient";(*"ConjugateGradient",
"Newton", "QuasiNewton",
"PrincipalAxis"*)*)
```

In[88]:=
```
methodMin = "QuasiNewton";
```

Iterate

In[89]:=
```
timeListPSI = Table[, {Ntests}];
```

```
Do[
 (*initialize displacements and Lagrange multipliers*)
 methodU = ConstantArray[0., {numNodes*numDOFs, 1}];
 methodEtas = ConstantArray[0., {numNodes*numDOFs, 1}];
 (*auxiliary srrays to save information about
  the iterations*)
 internalForcesListIPS = {};
 (*force resdiual across iterations*)
 psDistList = {};
 (*distance after iterations*)
 elemN = 10;
 (*particular element number to monitor phase
  space points and visualize convergence -- see
  figure 3 in the paper*)
 psListE = {};
 (*to save phase-space points in E visited by
   that element*)
 psListD = {};
 (*to save phase-space points in D visited by
   that element*)
 timeProjsE = {}; (*to save time invested in
  projecting onto E*)
 timeProjsD = {}; (*to save time invested in
  projecting onto D*)
 (*iterations*)
 (*just initialization*)
 (*initial point (origin),
 this is actually a material-admissible state*)
 sigmaPrime = ConstantArray[{0., 0., 0.}, numElements];
 epsilonPrime = ConstantArray[{0., 0., 0.}, numElements];
 sigma = ConstantArray[{0, 0, 0}, numElements];
```

```
epsilon = ConstantArray[{0, 0, 0}, numElements];
zetaPrev = Table[Join[epsilon[[ii]], sigma[[ii]]],
  {ii, 1, numElements}];
cont = True;
q = 0;
Print["Solving with approach " <> approach <>
  ", with method " <> methodMin];
(*loop*)
timePSI = AbsoluteTiming[
  While[cont,
   q = q + 1;
   (*------------------------------------------------------
                              ----------------------------
                             *)
   (*------------------------------------ PROJECTION
     ONTO E ------------------------------*)
   tictoc = AbsoluteTiming[
     rhsU = ConstantArray[0., {numDOFs*numNodes, 1}];
     (*initialize*)
     forceVecInt = ConstantArray[0.,
       {numDOFs*numNodes, 1}]; (*initialize*)
     (*build eqns' rhs for this
      step ----------------------------------------
                              ----------------------------
           --------*)
     (*K u = Sum[w B^T C ϵ*]=rhsU i.e.,
     right-hand side of the equation to compute U*)
     (*K η = F_ext - Sum[w B^T σ*]*)
     Do[
      (*strain contribution*)
```

```
 rhsU[[locDOFs[[ii]]]] =
  elAreas[[ii]]*thickness*
    Transpose[elMatB[[ii]]].methodC.
     epsilonPrime[[ii]] + rhsU[[locDOFs[[ii]]]];
 (*stress contribution*)
 forceVecInt[[locDOFs[[ii]]]] =
  elAreas[[ii]]*thickness*
    Transpose[elMatB[[ii]]].sigmaPrime[[ii]] +
   forceVecInt[[locDOFs[[ii]]]];
 , {ii, 1, numElements}];
(*-----------------------------------------------
                          ------------------------
                          ----*)
(*Solve linear system*)
methodU[[activeDOFs]] =
 LinearSolve[methodK[[activeDOFs, activeDOFs]],
  rhsU[[activeDOFs]]];
(*no need to condense forces,
no forced displacement*)
methodEtas[[activeDOFs]] =
 LinearSolve[methodK[[activeDOFs, activeDOFs]],
  forceVecExt[[activeDOFs]] -
   forceVecInt[[activeDOFs]]];
(*update stress and strain*)
Do[
 (*σe = σe' + C Be η*)
 sigma[[ii]] = sigmaPrime[[ii]] +
   Flatten[methodC.elMatB[[ii]].
     methodEtas[[locDOFs[[ii]]]]];
 (*σe = σe* + C Be ηe*)
 (*εe = Be u*)
```

```
    epsilon[[ii]] =
     Flatten[elMatB[[ii]].methodU[[locDOFs[[ii]]]]];
    (*ϵe = Be ue*)
    , {ii, 1, numElements}];
 ];
(*-------------------------------------------------------
                              ----------------------------
                              -------------*)
AppendTo[timeProjsE, tictoc];
AppendTo[psListE, {epsilon[[elemN]], sigma[[elemN]]}];
(*-------------------------------------------------------
                              ----------------------------
                              *)
AppendTo[internalForcesListIPS, forceVecInt];
resError =
 Norm[methodK[[activeDOFs, activeDOFs]].
    methodEtas[[activeDOFs]]] /
  Norm[forceVecExt[[activeDOFs]]];
zeta = Table[Join[epsilon[[ii]], sigma[[ii]]],
  {ii, 1, numElements}];
psDist = Norm[Flatten[zeta] - Flatten[zetaPrev]] / Norm[Flatten[zeta]];
AppendTo[psDistList, psDist];
(*-------------------------------------------------------
                              ----------------------------
                              *)
Which[
 q > 2000, Print["Solver failed to converge"];
 Break[], (*surpassed the limit of iterations*)
 (*-----------------------------------------------*)
 resError < tol,
```

```
  Print["Solver converged in " <> ToString[q] <>
    " iterations (equilibrium tolerance
      satisfied)."]; Break[],
  (*-----------------------------------------------*)
  psDist < tolPS,
  Print["Solver converged in " <> ToString[q] <>
    " iterations; phase-space distance
      converged (eq.res=" <>
    ToString[DecimalForm[100*resError, {3, 2}]] <>
    "%)"]; Break[],
  (*-----------------------------------------------*)
  True,
  Print["Residual: " <>
    ToString[DecimalForm@resError] <>
    ", phase-space percentual increment: " <>
    ToString[DecimalForm@psDist]]
];
zetaPrev = zeta;
(*-------------------------------------------------
                          ----------------------------
                          *)
(*------------------------------------ PROJECTION
  ONTO D ------------------------------*)
If[approach == "minimization",
 (*YES, go use minimization*)
 If[optimalC == "False",
   (*---------- Using distance
    minimization -------------------------*)
   tictoc = First@AbsoluteTiming[
      epsilonPrime = ParallelTable[
```

```
        {a, b, y} /. Last@Quiet@
            FindMinimum[
             Cd * Norm[{a, b, y} - epsilon[[ii]]]^2 +
              Norm[material[{a, b, y}] - sigma[[ii]]]^2 / Cd,
             {{a, epsilon[[ii]][[1]]},
              {b, epsilon[[ii]][[2]]},
              {y, epsilon[[ii]][[3]]}},
             (*initial guess*)
             Method -> methodMin](*method*)
        , {ii, numElements}];];
    (*Project stresses*)
    sigmaPrime = material /@ epsilonPrime;,
    (*---------- Using distance minimization
     w/ optimal C --------------------------*)
    tictoc = First@AbsoluteTiming[
        epsilonPrime = ParallelTable[
            {a, b, y} /. Last@Quiet@FindMinimum[
             (material[{a, b, c}] - sigma[[ii]]).
              (material[{a, b, y}] - sigma[[ii]]),
             {{a, epsilon[[ii]][[1]]},
              {b, epsilon[[ii]][[2]]},
              {y, epsilon[[ii]][[3]]}},
             (*initial guess*)
             Method -> methodMin](*method*)
        , {ii, numElements}];];
    (*Project stresses*)
    sigmaPrime = sigma;
];, (*NO, use Euler-Lagrange equations*)
```

```
    (*Break[];*)
    (*---------- Using distance E-
     L w/ optimal C -------------------------*)
    tictoc = First@AbsoluteTiming[
       epsilonPrime = ParallelTable[
          {a, b, y} /. Last@Quiet@FindMinimum[
               (material[{a, b, y}] - sigma[[ii]]).
                (material[{a, b, y}] - sigma[[ii]]),
               {{a, epsilon[[ii]][[1]]},
                {b, epsilon[[ii]][[2]]},
                {y, epsilon[[ii]][[3]]}}, (*initial
                guess*)
               Method -> methodMin](*method*)
            , {ii, numElements}];];
     (*Project stresses*)
     sigmaPrime = sigma;
    ];
    (*-----------------------------------------------------
                          ------------------- save
     time*)
    AppendTo[psListD, {epsilonPrime[[elemN]],
      sigmaPrime[[elemN]]}];
    AppendTo[timeProjsD, tictoc];
   ](*end while*)
  ];
 timeListPSI[[tt]] = timePSI;
 , {tt, 1, Ntests}]
```

```
Solving with approach minimization, with method QuasiNewton
Residual: 1., phase-space percentual increment: 1.
```

```
Residual: 0.0993757,
  phase-space percentual increment: 0.0303667
Residual: 0.0731393,
  phase-space percentual increment: 0.0196742
Residual: 0.0563818,
  phase-space percentual increment: 0.015241
Residual: 0.0432433,
  phase-space percentual increment: 0.0120636
Residual: 0.0333261, phase-space
  percentual increment: 0.00949006
Residual: 0.0259936, phase-space
  percentual increment: 0.00748172
Residual: 0.0206514, phase-space
  percentual increment: 0.00596663
Residual: 0.0168229, phase-space
  percentual increment: 0.00484044
Residual: 0.0141308, phase-space
  percentual increment: 0.00400283
Residual: 0.0122708, phase-space
  percentual increment: 0.00337154
Residual: 0.0110004, phase-space
  percentual increment: 0.00288712
Residual: 0.0101307, phase-space
  percentual increment: 0.00250725
Solver converged in 14
  iterations (equilibrium tolerance satisfied).
```

In[91]:=
```
sf = 100;
(*--------------*)
deformedShape =
```

```
  Table[
   nodes[[ii]] + sf * First /@ {Unr[[ii]], Unr[[ii + numNodes]]},
   {ii, 1, numNodes}];
edgeList = {};
Do[
 list = DeleteDuplicates@
   Flatten[deformedShape[[#]] & /@
     (Sort /@ Permutations[connectivity[[ii]], {2}]), 1];
 AppendTo[edgeList, Line@AppendTo[list, First@list]];
 , {ii, 1, numElements}]
(*--------------*)
deformedShapeM =
  Table[
   nodes[[ii]] +
    sf * First /@ {methodU[[ii]], methodU[[ii + numNodes]]},
   {ii, 1, numNodes}];
edgeListM = {};
Do[
 list = DeleteDuplicates@
   Flatten[deformedShapeM[[#]] & /@
     (Sort /@ Permutations[connectivity[[ii]], {2}]), 1];
 AppendTo[edgeListM, Line@AppendTo[list, First@list]];
 , {ii, 1, numElements}]
(*--------------*)
Show[
 ElementMeshDeformation[mesh, {usol, vsol},
   "ScalingFactor" → 0][
  "Wireframe"["ElementMeshDirective" →
    Directive[EdgeForm[Red], FaceForm[]]]],
 Graphics[{
   {Lighter@Gray, Thickness[0.01], edgeList},
```

```
  {Dashed, Black, Thickness[0.0015], edgeListM}
 }],
Axes → True,
PlotRange → All,
PlotRangeClipping -> True,
ImageSize → Large,
Background -> White]
```

Out[98]=