D-refinement plane-stress example

This notebook is a companion to “Mesh d-refinement: a data-based computational framework to account for complex material response”.
The results presented in Section 3.1. (plate with circular hole) are derived herein.

J. Garcia-Suarez, 2022

Pre-processing

Create mesh. Being by loading FEM tools:

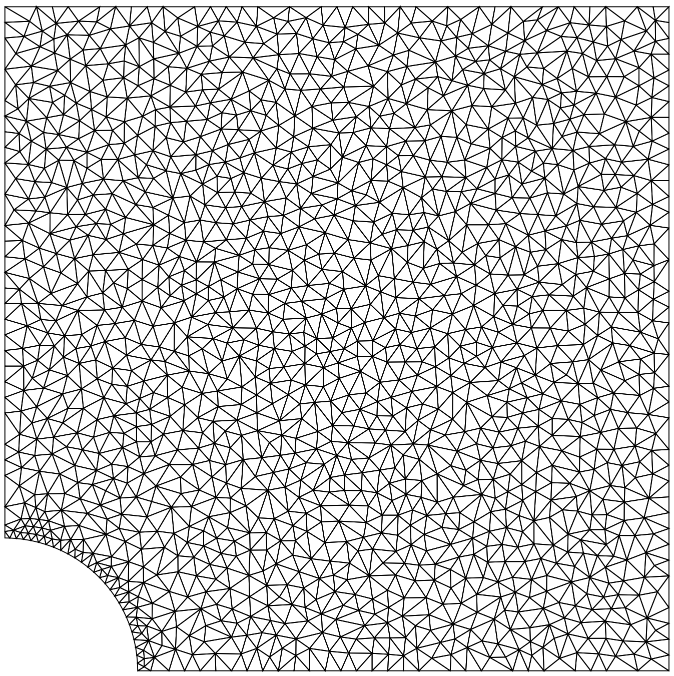
In[1]:=
```
Needs["NDSolve`FEM`"];
```

Define region (exploiting symmetry), see Fig.3a

In[2]:=
```
r = 0.02; (*hole radius*)
```

In[3]:=
```
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[5]=



Mesh properties

In[6]:=
```
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[12]:=
```
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[18]:=
```
thickness = 1.;
```

Plane stress operator for Mathematica pre-processing

Loading: constant traction at the upper edge

In[19]:=
```
p = 10^8;(*Pa (all units SI)*)
tractions = NeumannValue[p, y == 0.1];
```

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

In[21]:=
```
planeStress =
  {Inactive[Div][{{0, -((Y * ν) / (1 - ν ^ 2))}, {-(Y * (1 - ν)) / (2 * (1 - ν ^ 2)), 0}}.Inactive[Grad][v[x, y],
         {x, y}], {x, y}] + Inactive[Div][{{-(Y / (1 - ν ^ 2)), 0}, {0, -(Y * (1 - ν)) / (2 * (1 - ν ^ 2))}}.
        Inactive[Grad][u[x, y], {x, y}], {x, y}], Inactive[Div][
      {{0, -(Y * (1 - ν)) / (2 * (1 - ν ^ 2))}, {-((Y * ν) / (1 - ν ^ 2)), 0}}.Inactive[Grad][u[x, y], {x, y}],
      {x, y}] + Inactive[Div][{{-(Y * (1 - ν)) / (2 * (1 - ν ^ 2)), 0}, {0, -(Y / (1 - ν ^ 2))}}.
        Inactive[Grad][v[x, y], {x, y}], {x, y}]} /. {Y → 200. * 10^9, ν → 33. / 100};
```

BCs:

In[22]:=
```
(* held fixed at left *)
bcs = {
   DirichletCondition [u[x, y] == 0, x ≤ 0],
   DirichletCondition [v[x, y] == 0, y ≤ 0]};
```

Define PDE:

In[23]:=
```
pde2D = planeStress == {0, 1. * tractions};
```

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

In[24]:=
```
{usol, vsol} = NDSolveValue[{pde2D, bcs}, {u, v}, {x, y} ∈ mesh];
```

Pre-processing: solve linear-elastic problem

Define intact material

Intact Material: $E = 200^9$Pa and v=0.33

In[25]:=
```
youngModIntact = 200. * 10^9; nu = 0.33;
```

In[26]:=
```
(*Intact plane-stress tensor*)
matCIntact = ({{1, nu, 0},
     {nu, 1, 0},
     {0, 0, (1 - nu)/2}} * youngModIntact/(1 - nu^2));
```

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

In[27]:=
```
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

In[41]:=
```
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 FEM K matrix

Element-wise contributions:

In[45]:=
```
elMatK = Table[
   Transpose[elMatB[[ii]]].matCIntact.elMatB[[ii]] * elAreas[[ii]], {ii, 1, numElements}];
```

Assemble:

In[46]:=
```
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[51]:=
```
restrainedDOFsY = Sort[Flatten[{# + numNodes} & /@ lowerEdge]];
activeDOFs = DeleteCases[Range[numDOFs * numNodes], Alternatives @@ restrainedDOFsY];
```

Next the vertical symmetry plane:

In[53]:=
```
restrainedDOFsX = Sort[Flatten[{#} & /@ leftEdge]];
activeDOFs = DeleteCases[activeDOFs, Alternatives @@ restrainedDOFsX];
```

Solve linear-elastic solution using matrices that will later be used in d-refinement

In[55]:=
```
forceVecExt = load;
intactU = SparseArray[{}, {numNodes * numDOFs, 1}];
intactU[[activeDOFs]] =
  LinearSolve[globalK[[activeDOFs, activeDOFs]], forceVecExt[[activeDOFs]]];
```

Post-processing: identify elements over threshold

Mean stress function (necessary to define the threshold):

In[58]:=
```
sigmaMean[σ_] := (σ[[1, 1]] + σ[[2, 1]]) / 2;
```

Elements over σ_{lim}

In[59]:=
```
limitStress = 0.75 * 10^8;
```

Compute stresses and damage of every material

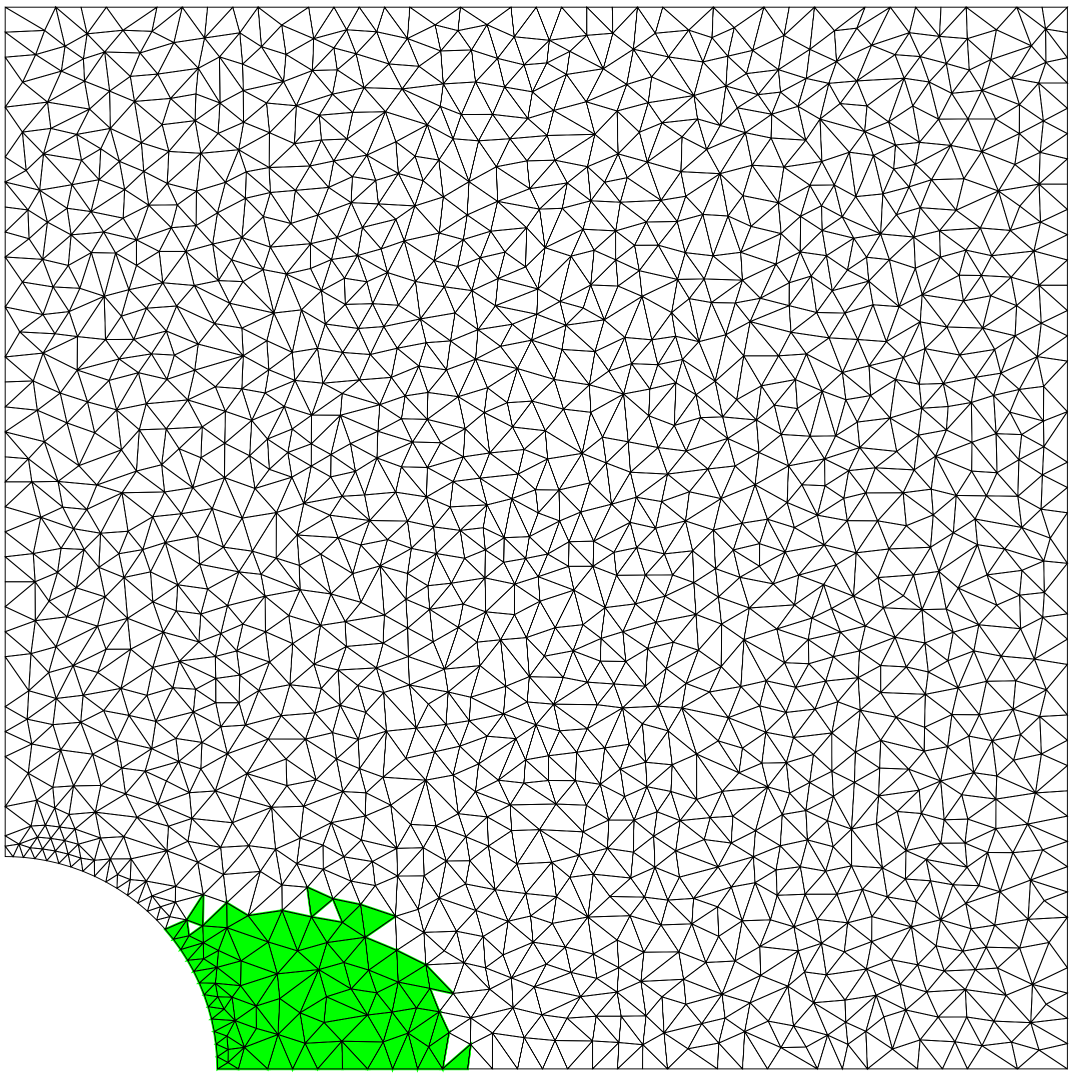
In[60]:=
```
aboveThresholdQ = SparseArray[{}, {numDOFs * numNodes, 1}];
stressIntact = SparseArray[{}, {numDOFs * numNodes, 1}];
Do[
  (*displacements in the relevant nodes*)
  Ue = intactU[[locDOFs[[ii]]]];
  matB = elMatB[[ii]];
  (*element strains*)
  strain = matB.Ue;
  (*element stress*)
  stress = matCIntact.strain;
  stressIntact[[ii]] = stress;
  aboveThresholdQ[[ii]] = (sigmaMean[stress] > limitStress);
  (*material stiffnesses*)
  , {ii, 1, numElements}];
```

How many above threshold? Important to compare later to NR and to d-refinement

In[63]:=
```
aboveThresholdElements = Flatten@Position[aboveThresholdQ, _?(# == True &)];
showAboveThresholdElements = Table[If[
     aboveThresholdQ[[ii]] == True,
     Polygon[nodes[[connectivity[[ii]]]]]
    ], {ii, 1, numElements}];
```

In[65]:=
```
Show[
 Graphics[{
   {Green, showAboveThresholdElements },
  }],
 mesh["Wireframe"["MeshElementStyle" → EdgeForm[Black]]]
 , ImageSize → Large]
```

Out[65]=



Solve using Newton-Raphson method

The reason to use NR is two-fold:

it is the main reference to compare d-refinement with

and it us used to generate the dataset that is later used by d-refinement framework

Define softening behavior

Use softening behavior for to the Young based on mean stress (inspired by crack-shielding behavior)

In[66]:=
```
youngMod[σ_] := If[σ < limitStress ,
  (*no damage*)
  youngModIntact ,
  (*damaged*)
  Max[(limitStress/σ)^1 youngModIntact , 0.5 * youngModIntact ]
 ]
```

The loop

Convergence tolerance: the step has converged once $|f_{ext} - f_{int}|/|f_{ext}|$<tol

In[67]:=
```
tol = 1. * 10^-3;
```

Initialize the rest

The following arrays store the states “visited” by each element during the NR simulation (they will later be used as dataset for d-refinement)

In[68]:=
```
stressHistories = {};(*SparseArray [{},{numLoadSteps }];
for each loading step, for each element*)
strainHistories = {};(*SparseArray [{},{numLoadSteps }];
for each loading step, for each element*)
```

In[70]:=
```
numLoadSteps = 10; (*number of steps we use to derive the load*)
(*save material response history*)
(*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 ----------------------------------------------------------------------------
     -------*)
AbsoluteTiming [
 Do[
   forceVecExt = ff/numLoadSteps load;
   normForceExt = Norm[forceVecExt ];
   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*)
  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!*)
    (*-------------------------------------------------------------------------*)
   ];
   AppendTo[stressHistories, stressList];
   AppendTo[strainHistories, strainList],
   (*save histories*)
   {ff, 1, numLoadSteps}];]
```

```
*---------------------------*
Load step # 1 out of 10
*---------------------------*
Converged in 2 steps
*---------------------------*
Load step # 2 out of 10
*---------------------------*
Converged in 2 steps
*---------------------------*
Load step # 3 out of 10
*---------------------------*
Converged in 2 steps
*---------------------------*
Load step # 4 out of 10
*---------------------------*
Converged in 2 steps
```

```
*----------------------------*
Load step # 5 out of 10
*----------------------------*
Converged in 4 steps
*----------------------------*
Load step # 6 out of 10
*----------------------------*
Converged in 6 steps
*----------------------------*
Load step # 7 out of 10
*----------------------------*
Converged in 6 steps
*----------------------------*
Load step # 8 out of 10
*----------------------------*
Converged in 7 steps
*----------------------------*
Load step # 9 out of 10
*----------------------------*
Converged in 7 steps
*----------------------------*
Load step # 10 out of 10
*----------------------------*
Converged in 8 steps
```

Out[73]= {38.4604, Null}

Post-process

Deformation:

In[74]:=
```
sf = 200;
deformedShape =
  Table[nodes[[ii]] + sf * First /@ {Unr[[ii]], Unr[[ii + numNodes]]}, {ii, 1, numNodes}];
```

Damaged elements:

In[76]:=
```
strains = ConstantArray [0., numElements ];
stresses = ConstantArray [0., numElements ];
Do[
  (*displacements  in  the  relevant  nodes*)
  Ue = Unr[[locDOFs[[ii]]]];
  matB = elMatB[[ii]];
  (*element  strains*)
  strain = matB.Ue;
  (*element  stresses*)
  stress = matCList[[ii]].strain ;
  (*save*)
  strains[[ii]] = strain ;
  stresses[[ii]] = stress ;
  , {ii, 1, numElements }];
```

The following array contains the position of the damaged elements:

In[79]:=
```
damagedQ = (sigmaMean[#] > limitStress ) & /@ stressList ;
```

Take a quick look at the number of elements that are above the threshold according the linear-elastic simualtion...

In[80]:=
```
Total @ aboveThresholdQ
```

Out[80]=
```
{2801 False + 123 True}
```

... and according to the non-linear solver...

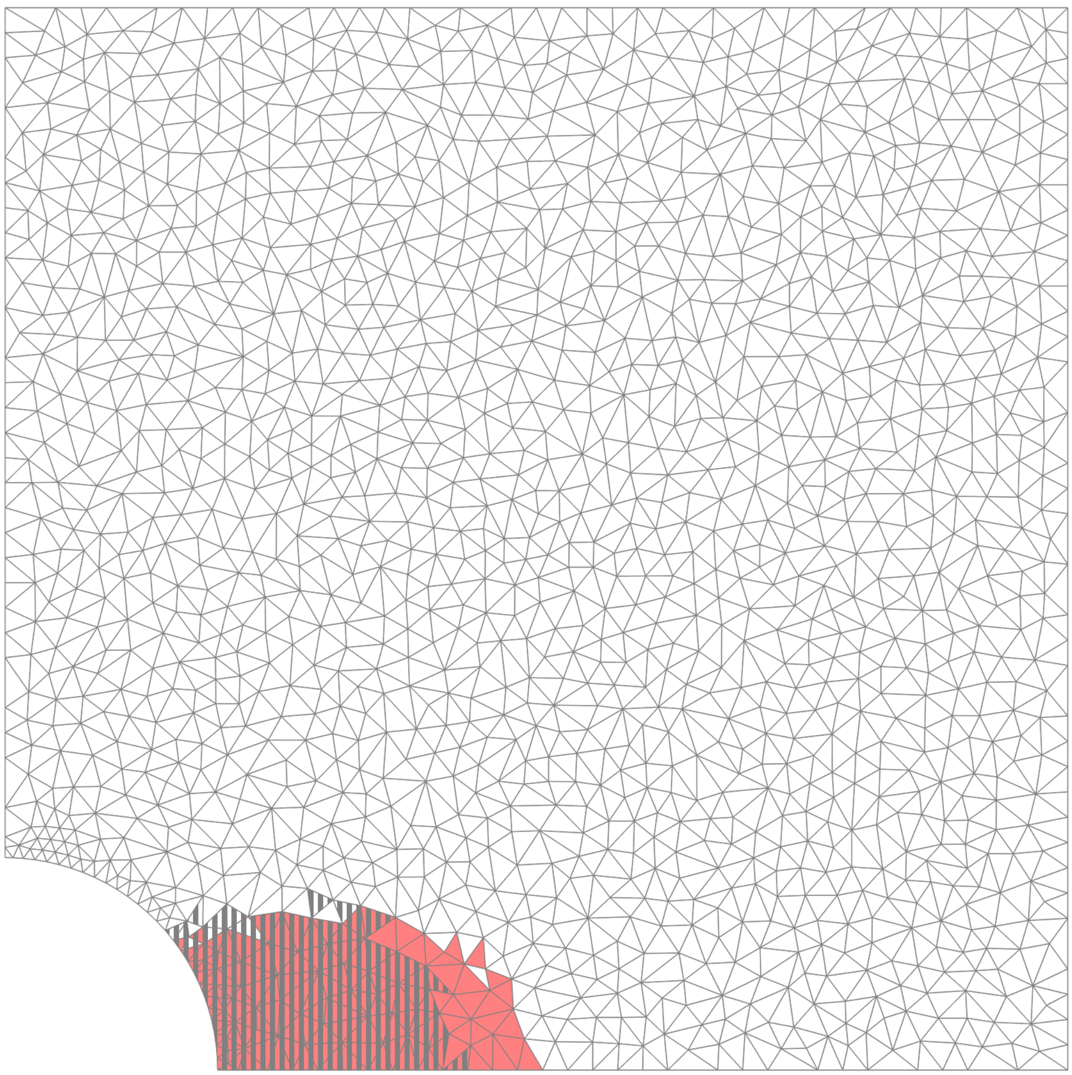
In[81]:=
```
Total @ damagedQ
```

Out[81]=
```
2785 False + 139 True
```

Visualize (zooming out Fig.3b upper panel)

In[82]:=
```
damagedElements = Flatten@Position[damagedQ, _?(# == True &)];
showDamagedElements = Table[If[
     damagedQ[[ii]] == True,
     Polygon[nodes[[connectivity[[ii]]]]]
   ], {ii, 1, numElements}];
Show[
 Graphics[{
   {Pink, showDamagedElements},
   {Gray, HatchFilling[π/2, 2], showAboveThresholdElements}
  }],
 mesh["Wireframe"["MeshElementStyle" → EdgeForm[Gray]]]
 , ImageSize → Large]
```

Out[84]=



D-refinement

DDCM method's constant matrix

Peculiarities of d-refinement: we can use the "intact material" matrix for the method (**methodC**)

In[85]:=
```
matCinv = Inverse[matCIntact];
methodC = matCIntact;
methodCinv = Inverse[methodC];
methodK = globalK;
elMatC = elMatK;
```

Dataset

In[90]:=
```
(* (*Do this instead in case that you want
 to include also the elastic part in the dataset*)
setD=Table[
  Flatten@Join[strainHistories[[jj,ii]],stressHistories[[jj,ii]]],
  {ii,1,numElements},{jj,1,numLoadSteps}];
setD=Flatten[setD,1];
*)
```

In[91]:=
```
setD = {};
Do[(*for each element*)
  Do[(*for each load increment*)
    If[sigmaMean[stressHistories[[jj, ii]]] > 0.8 limitStress,
      AppendTo[setD,
       Flatten@Join[strainHistories[[jj, ii]], stressHistories[[jj, ii]]]]];
    , {jj, 1, numLoadSteps}];
  , {ii, 1, numElements}];
```

In[93]:=
```
PrependTo[setD, {0., 0., 0, 0., 0., 0.}];
setD = DeleteDuplicates@setD;
```

In[95]:=
```
Length@setD
```

Out[95]= 886

Distance function

Pre-compute the distance function we are going to use

In[96]:=
```
distFunc = Nearest[setD → {"Element", "Index", "Distance"},
    Method → "Scan",
    DistanceFunction → (methodC.(#1[[1 ;; 3]] - #2[[1 ;; 3]]).(#1[[1 ;; 3]] - #2[[1 ;; 3]]) +
        methodCinv.(#1[[4 ;; 6]] - #2[[4 ;; 6]]).(#1[[4 ;; 6]] - #2[[4 ;; 6]]) &)];
```

Initialize Parameters

In[97]:=
```
numTotalDOFs = numNodes * numDOFs;(*the number of dofs*)
dofs2Solve4 = Join[activeDOFs, activeDOFs + numTotalDOFs];
(*matrix to solve the coupled system, eq.(7)*)
(*Which are the DD elements?*)
numDDelements = 0;(*How many DD elements? Always zero at firtst 0*)
indexDDs = RandomChoice[Range[numElements], numDDelements];
(*position of DD elements, this is created as a random choice as per tradition,
but in this case is an empty set*)
indexFEs = Delete[Range[numElements], ArrayReshape[indexDDs, {numDDelements, 1}]];
(*will return all positions in this case*)
numFEelements = numElements - numDDelements;
(*how many FE elements? In this case, all*)
listFEelements = {indexFEs};(*auxiliary list*)
```

Iterate

Initialize kernels for parallel searches

In[104]:=
```
numKernels = 6;
CloseKernels[];
LaunchKernels[numKernels];
```

Looping time

Prepare loop variables

In[107]:=
```
zetaStar = {};(*because in this case there are no DD elements at first*)
zetaStarIndices = {};(*this list points to the label of the datum ∈
 D assigned to the corresponding DD element*)
(*to store index changes over the simulation*)
indexList = {zetaStarIndices};
(*Initial energy (basically inf)*)
dataEnergy = 10^50;
(*to store energy evolution over iterations*)
energyList = {dataEnergy};
(*auxiliary arrays to store information*)
methodU = SparseArray[{}, {numTotalDOFs, 1}];
methodEtas = SparseArray[{}, {numTotalDOFs, 1}];
methodSol = SparseArray[{}, {2 numTotalDOFs, 1}];
(*auxiliary array to compare FE elements between iterations*)
auxIndexFEs = indexFEs;
(*auxiliary array to save number of elements that change datum*)
listChanges = {};
```

In[117]:=
```
cont = True;
q = 0;
(*loop*)
AbsoluteTiming[While[cont == True,
  q = q + 1;
  If[q > 10 000, Break[]]; (*maximum number of iterations*)
  rhsU = SparseArray[{}, {numDOFs * numNodes, 1}]; (*initialize*)
  rhsF = SparseArray[{}, {numDOFs * numNodes, 1}]; (*initialize*)
  (*build eqns' rhs for this
   step ----------------------------------------------------------------------------
    --*)
  (*K_method u - K_mat η = Sum[w B^T C ϵ*]*)
  (*K_mat u + K_method η = f - Sum[w B^T σ*]*)
  If[numElements == numFEelements,
   (*just FE*)
   (*K_mat u = f*)
   methodU[[activeDOFs]] =
    LinearSolve[globalK[[activeDOFs, activeDOFs]], forceVecExt[[activeDOFs]]];
   Print["No DD elements, solved w/ FEM"],
   (*there are DD elements*)
   Do[
    ll = indexDDs[[ii]];
    (*strain contribution*)
    rhsU[[locDOFs[[ll]]]] = elAreas[[ll]] * thickness *
       Transpose[elMatB[[ll]]].methodC.zetaStar[[ii]][[1 ;; 3]] + rhsU[[locDOFs[[ll]]]];
```

```
 (*stress contribution*)
 rhsF[[locDOFs[[ll]]]] = elAreas[[ll]] * thickness *
    Transpose[elMatB[[ll]]].zetaStar[[ii]][[4 ;; 6]] + rhsF[[locDOFs[[ll]]]];
 , {ii, 1, numDDelements}];
(*assemble rhs into single vector*)
rhs = Join[rhsU, forceVecExt - rhsF];
(*-----------------------------------------------------------------------------
                                    ----------------------------------------
                                    --------*)
(*Construct coupling matrices*)
(*----------------------------------------------------------------------------*)
(*Material entries, FEM entries (antidiagonal block)*)
totalListFE = ConstantArray[0., {numFEelements, 1}];
Do[
 ll = indexFEs[[kk]];
 subList1 = Table[
   {If[ii ≤ 3, connectivity[[ll]][[ii]],
       connectivity[[ll]][[ii - 3]] + numNodes], (*row: 1st position*)
     If[jj ≤ 3, connectivity[[ll]][[jj]], connectivity[[ll]][[jj - 3]] + numNodes] +
      numTotalDOFs (*column: 2nd position*)
    } → -1.0 elMatK[[ll]][[ii, jj]], (*minus material values*)
   {ii, 1, 3 * numDOFs}, {jj, 1, 3 * numDOFs}];
 subList2 = Table[
   {If[ii ≤ 3, connectivity[[ll]][[ii]],
       connectivity[[ll]][[ii - 3]] + numNodes] + numTotalDOFs, (*row*)
     If[jj ≤ 3, connectivity[[ll]][[jj]], connectivity[[ll]][[jj - 3]] + numNodes]
     (*column*)
    } → elMatK[[ll]][[ii, jj]], (*material values*)
   {ii, 1, 3 * numDOFs}, {jj, 1, 3 * numDOFs}];
 totalListFE[[kk]] = Join[Flatten[subList1, 1], Flatten[subList2, 1]];
 , {kk, 1, numFEelements}];
totalListDD = ConstantArray[0., {numDDelements, 1}];
(*Method entries, DD entries (diagonal block)*)
Do[
 ll = indexDDs[[kk]];
 subList1 = Table[
   {If[ii ≤ 3, connectivity[[ll]][[ii]], connectivity[[ll]][[ii - 3]] + numNodes], (*row*)
     If[jj ≤ 3, connectivity[[ll]][[jj]],
      connectivity[[ll]][[jj - 3]] + numNodes](*column*)
    } → elMatC[[ll]][[ii, jj]], (*method values*)
   {ii, 1, 3 * numDOFs}, {jj, 1, 3 * numDOFs}];
 subList2 = Table[
   {If[ii ≤ 3, connectivity[[ll]][[ii]],
```

```
          connectivity [[ll]][[ii - 3]] + numNodes] + numTotalDOFs , (*row*)
        If[jj ≤ 3, connectivity [[ll]][[jj]], connectivity [[ll]][[jj - 3]] + numNodes] +
         numTotalDOFs (*column*)
      } → elMatC[[ll]][[ii, jj]], (*method values*)
     {ii, 1, 3 * numDOFs}, {jj, 1, 3 * numDOFs}];
  totalListDD [[kk]] = Join[Flatten[subList1 , 1], Flatten[subList2 , 1]];
  , {kk, 1, numDDelements }];
(*Assemble*)
totalListFE = Flatten @ totalListFE ;
totalListDD = Flatten @ totalListDD ;
totalList = Flatten @ Join[totalListFE , totalListDD ];
SetSystemOptions ["SparseArrayOptions " → {"TreatRepeatedEntries " → 1}];
couplingMatrix = SparseArray [totalList , {2 numTotalDOFs , 2 numTotalDOFs }];
SetSystemOptions ["SparseArrayOptions " → {"TreatRepeatedEntries " → 0}];
(*-----------------------------------------------------------------------------
                                             -------------------------------------------
                                             --------*)
(*---------------------------------------------------------------------------------*)
(*Solve linear system*)
methodSol = SparseArray [{}, {2 numTotalDOFs , 1}];
methodSol [[dofs2Solve4 ]] =
  LinearSolve [couplingMatrix [[dofs2Solve4 , dofs2Solve4 ]], rhs[[dofs2Solve4 ]]];
(*Compute new state z_k in E (projection onto E)*)
methodU = methodSol [[1 ;; numTotalDOFs ]];
methodEtas = methodSol [[1 + numTotalDOFs ;; -1]];
zeta = Table[Flatten @ Join[
      elMatB [[indexDDs [[ii]]]].methodU [[locDOFs [[indexDDs [[ii]]]]]], (*ϵ_e = B_e u_e*)
      zetaStar [[ii]][[4 ;; 6]] + Flatten [methodC .elMatB [[indexDDs [[ii]]]].
          methodEtas [[locDOFs [[indexDDs [[ii]]]]]]](*σ_e = σ_e^* + C B_e η_e*)
    ], {ii, 1, numDDelements }];
(*Project z_k onto D to find
  (z^*)_k+1 ------------------------------------------------------------------------
           -----------------*)
If[numDDelements > 12,
  searchResults = ParallelMap [distFunc , zeta[[1 ;; -1]]],
  searchResults = Map[distFunc , zeta[[1 ;; -1]]]
];
(*unpack search
   results ---------------------------------------------------------------------
           ------------------------------*)
(*new selected points in D*)
newState = Table[searchResults [[ii]][[1]][[1]], {ii, 1, numDDelements }];
(*index of the new selected points in D*)
```

```
  newIndexInD = Table[searchResults [[ii]][[1]][[2]], {ii , 1, numDDelements }];
  (*distance between the new selected points in D and the points in E*)
  distances = Table[searchResults [[ii]][[1]][[3]], {ii , 1, numDDelements }];
  (*compute the number of elements that have changed*)
  numChanges = Total[
    Boole[newIndexInD [[#]] ≠ Flatten[zetaStarIndices ][[#]]] & /@ Range[numDDelements ]];
  AppendTo[listChanges , numChanges ];
  zetaStarIndices = newIndexInD ;
  (*compute new energy difference*)
  newDataEnergy = Total[
    Table[thickness * elAreas[[indexDDs [[ii]]]] * distances [[ii]], {ii , 1, numDDelements }]];
  If[newDataEnergy < dataEnergy , (*no, keep going: store values and update*)
   cont = True;
   dataEnergy = newDataEnergy ;
   AppendTo[energyList , dataEnergy ];
   AppendTo[indexList , zetaStarIndices ];
   zetaStar = newState ,
   (*yes, get outta here *)
   cont = False;
   Break[]];
 ];
 (*Check elements over the
   threshold --------------------------------------------------------------------------
              -------------------------*)
 (*compute strains and check*)
 auxIndexFEs = indexFEs ;
 (*because indexFEs is gonna change in the loop*)
 Do[
  ll = auxIndexFEs [[ii]];
  (*displacements in the relevant nodes*)
  Ue = methodU [[locDOFs [[ll]]]];
  matB = elMatB [[ll]];
  (*element strains*)
  strain = matB .Ue;
  (*too much stress?*)
  stress = matCIntact .strain ;
  If[sigmaMean [stress] > 0.9 limitStress ,
   (*indeed, delete this FE element from the list and it to the DD bin*)
   indexFEs = DeleteCases [indexFEs , ll];
   AppendTo[indexDDs , ll];
   (*assign a datum to the new DD element*)
   (*newIndexMaterialPoint =RandomChoice [Range[Length @setD],1];
   AppendTo[zetaStar ,Flatten @setD[[newIndexMaterialPoint ]]];
```

```
    AppendTo[zetaStar ,{0.,0.,0.,0.,0.,0.}];
    AppendTo[zetaStarIndices ,1];*)
    searchOutcome = Flatten[distFunc @ Flatten[Join[strain , stress]], 1];
    AppendTo[zetaStar , searchOutcome [[1]]];
    AppendTo[zetaStarIndices , searchOutcome [[2]]];
   ]
   , {ii , 1, numFEelements }];
  numFEelements = Length @ indexFEs ;
  AppendTo[listFEelements , indexFEs];
  numDDelements = Length @ indexDDs ;
  Print[
   "# element (total) = " <> ToString[numElements ] <> " = " <> ToString[numFEelements ] <>
    " FE elements + " <> ToString[numDDelements ] <> " DD elements "
  ];
  (*If after the 1st check we have no refined any element ,
  no need of refinement *)
  If[numDDelements == 0, Print["No need of further refinement "];
    cont = False]
   (*Print
     progress ----------------------------------------------------------------------------
                                  ---------------------------------------------*)
   If[q > 1,
    Print["Step: " <> ToString[q] <> ", # of changes: " <>
      ToString[numChanges ] <> ", Log10 data Energy: " <> ToString[Log @ dataEnergy ]]
   ];
  Print["*--------------------------------------------------------*"]
 ]]
```

```
No DD elements , solved w/ FEM
# element (total) = 2924 = 2745 FE elements + 179 DD elements
*--------------------------------------------------------*
# element (total) = 2924 = 2735 FE elements + 189 DD elements
Step: 2, # of changes: 115, Log10 data Energy: -0.72199
*--------------------------------------------------------*
# element (total) = 2924 = 2729 FE elements + 195 DD elements
Step: 3, # of changes: 51, Log10 data Energy: -1.04368
*--------------------------------------------------------*
# element (total) = 2924 = 2726 FE elements + 198 DD elements
Step: 4, # of changes: 26, Log10 data Energy: -1.2351
*--------------------------------------------------------*
# element (total) = 2924 = 2725 FE elements + 199 DD elements
```

```
Step : 5, # of changes : 19, Log10 data Energy : -1.30114
*--------------------------------------------------------*
# element (total) = 2924 = 2724 FE elements + 200 DD elements
Step : 6, # of changes : 9, Log10 data Energy : -1.33944
*--------------------------------------------------------*
# element (total) = 2924 = 2722 FE elements + 202 DD elements
Step : 7, # of changes : 4, Log10 data Energy : -1.3631
*--------------------------------------------------------*
# element (total) = 2924 = 2722 FE elements + 202 DD elements
Step : 8, # of changes : 1, Log10 data Energy : -1.37101
*--------------------------------------------------------*
# element (total) = 2924 = 2722 FE elements + 202 DD elements
Step : 9, # of changes : 0, Log10 data Energy : -1.37978
*--------------------------------------------------------*
```

Out[119]=
```
{20.3036, Null}
```

In[120]:=
```
CloseKernels[];
```

Post-processing

Compare deformed shapes

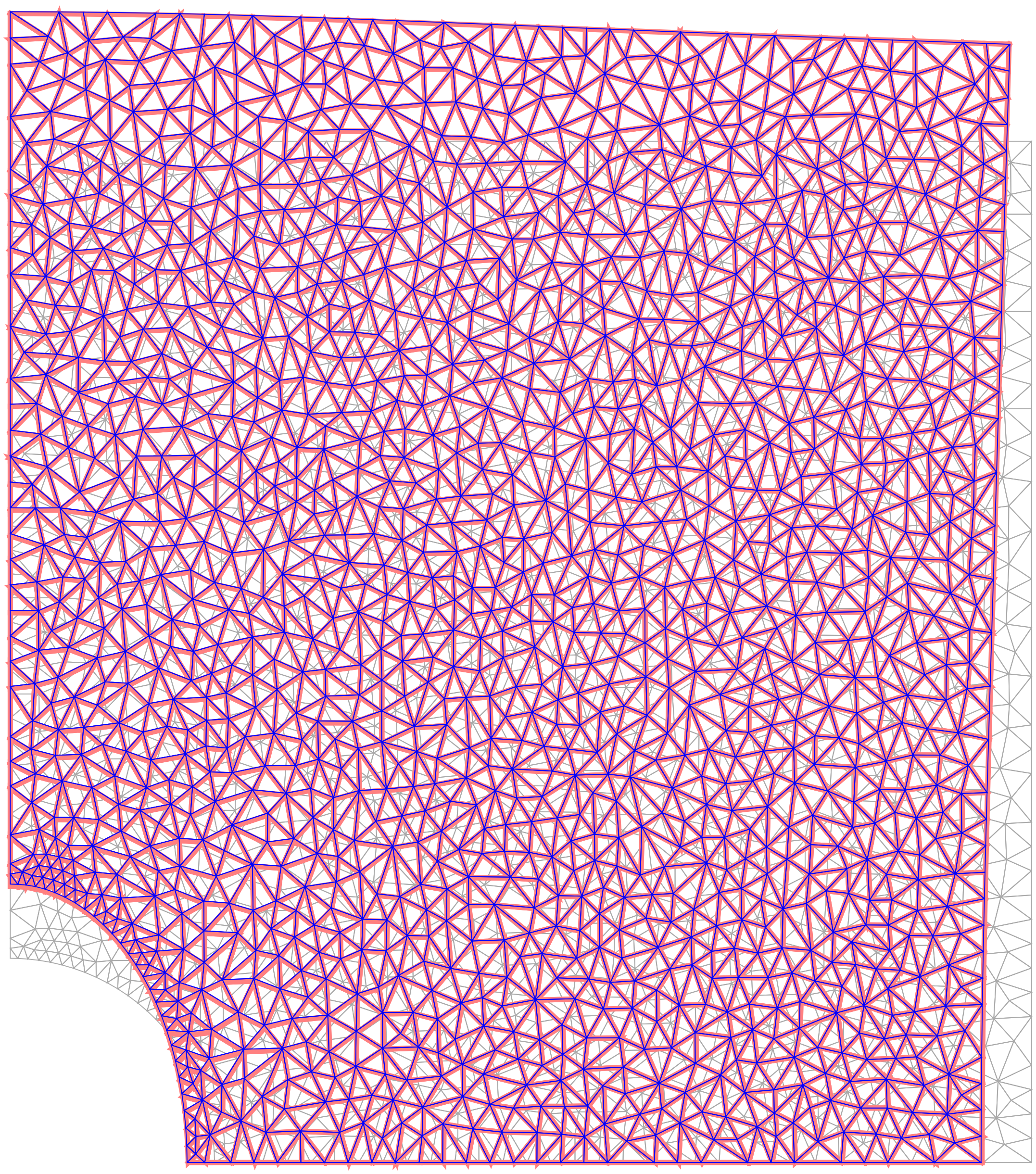
In[121]:=
```
sf = 200; (*deformation much exagerated to better notice differences*)
deformedShapeC =
 Table[nodes[[ii]] + sf * First /@ {methodU[[ii]], methodU[[ii + numNodes]]}, {ii, 1, numNodes}];
```

In[122]:=
```
edgeList = {};
Do[
 list = DeleteDuplicates @
   Flatten[deformedShape[[#]] & /@ (Sort /@ Permutations[connectivity[[ii]], {2}]), 1];
 AppendTo[edgeList, Line@AppendTo[list, First@list]];
 , {ii, 1, numElements}]
```

In[124]:=
```
edgeListC = {};
Do[
 list = DeleteDuplicates @
   Flatten[deformedShapeC[[#]] & /@ (Sort /@ Permutations[connectivity[[ii]], {2}]), 1];
 AppendTo[edgeListC, Line@AppendTo[list, First@list]];
 , {ii, 1, numElements}]
```

In[126]:=
```
Show[
 mesh["Wireframe"["MeshElementStyle" → EdgeForm[Lighter@Gray]]],
 Graphics[{
   {Pink, Thickness[0.005], edgeListC},
   {Blue, edgeList}
  }], ImageSize → Large]
```

Out[126]=

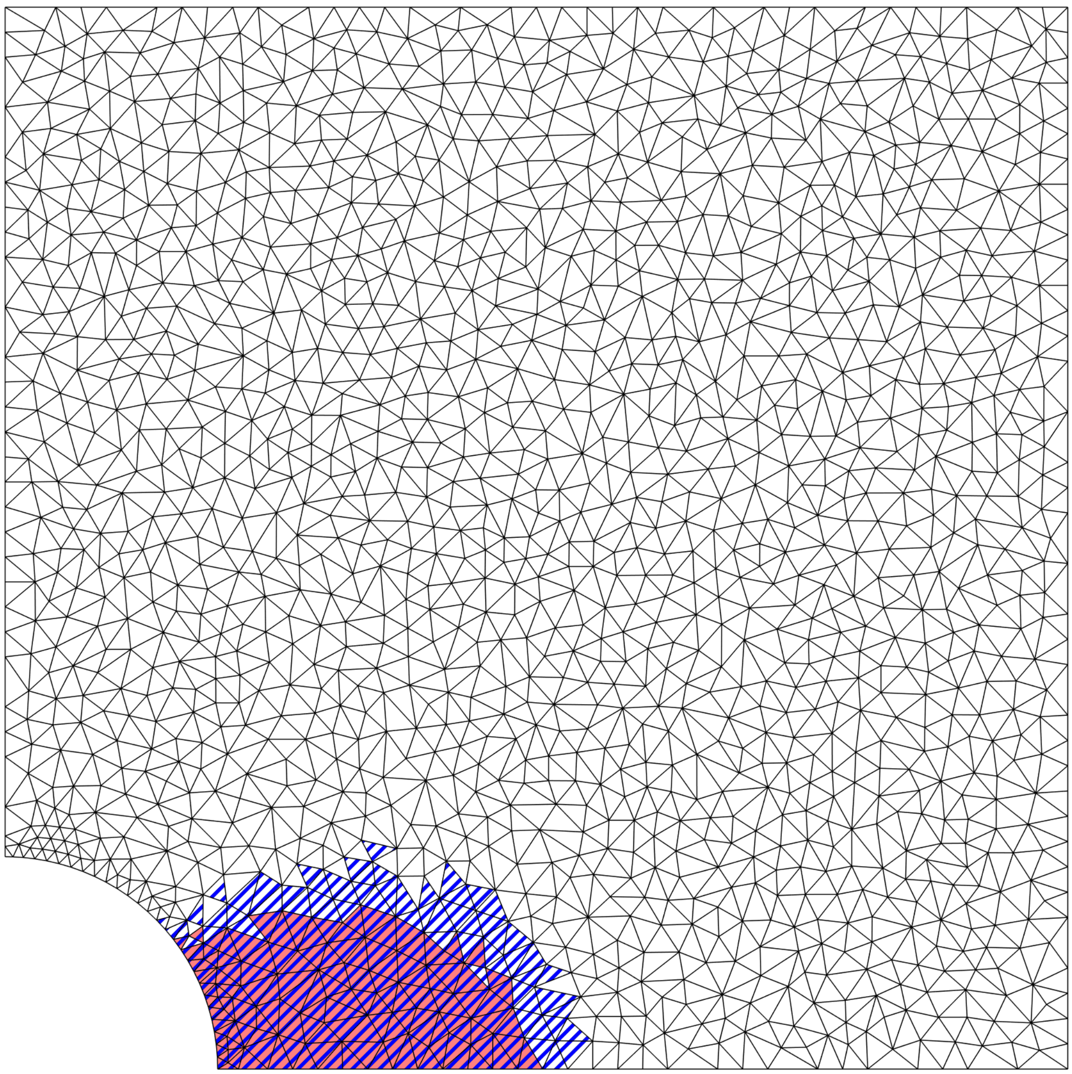


Visualize refinement

(this corresponds to zoom out of lower panel in fig.11b)

In[127]:=
```
showDDelements = Polygon[nodes[[connectivity[[#]]]]] & /@ indexDDs;
Show[
 Graphics[{Pink, showDamagedElements}],
 Graphics[{Blue, HatchFilling[Automatic, 1.25], showDDelements}],
 mesh["Wireframe"["MeshElementStyle" → EdgeForm[Black]]],
 Background → White,
 ImageSize → Large
]
```

Out[128]=



Phase-space distance between d-ref and NR solutions

In[129]:=
```
(*NR phase space location*)
zetaNR = Table[Flatten@Join[strainList[[ii]], stressList[[ii]]], {ii, 1, numElements}];
```

In[130]:=
```
(*DD phase space location: join FEM to DD*)
zetaDref = Table[
    If[MemberQ[indexDDs, ii], (*is DD?*)
     Flatten@zetaStar[[First@First@Position[indexDDs, ii]]], (*if DD*)
     Flatten@Join[elMatB[[ii]].methodU[[locDOFs[[ii]]]],
       matCIntact.elMatB[[ii]].methodU[[locDOFs[[ii]]]]](*if FEM*)
    ]
    , {ii, 1, numElements}];
```

Auxiliary phase-space distance squared function:

In[131]:=
```
distanceSquare[a_, b_] := methodC.(a[[1 ;; 3]] - b[[1 ;; 3]]).(a[[1 ;; 3]] - b[[1 ;; 3]]) +
   methodCinv.(a[[4 ;; 6]] - b[[4 ;; 6]]).(a[[4 ;; 6]] - b[[4 ;; 6]])
```

Distance between NR solution and d-refinement solution:

In[132]:=
```
distanceNTtoDref =
   (Total[Table[thickness * elAreas[[ii]] * distanceSquare[zetaNR[[ii]], zetaDref[[ii]]],
       {ii, 1, numElements}]])^(1/2);
```

as a percentage of distance from DD solution to the origin:

In[133]:=
```
100
 (distanceNTtoDref / (Total[Table[thickness * elAreas[[ii]] * distanceSquare[ConstantArray[
           0., {6}], zetaDref[[ii]]], {ii, 1, numElements}]])^(1/2))
```

Out[133]= 3.40614

Solve also w/ pure DD

Dataset

Make sure that it is properly created given the material

In[134]:=
```
(*This in case that we want to include also the elastic part in the dataset*)
setD = Table[
    Flatten@Join[strainHistories[[jj, ii]], stressHistories[[jj, ii]]],
    {ii, 1, numElements}, {jj, 1, numLoadSteps}];
setD = Flatten[setD, 1];
```

The regular DDCM is slower than either NR or d-ref. No need to use the whole dataset to prove that, so we use a smaller set of about 5000 points chosen at random for the DDCM simulations

In[136]:=
```
setD = RandomChoice[setD, 5000];
```

In[137]:=
```
PrependTo[setD, {0., 0., 0, 0., 0., 0.}];
setD = DeleteDuplicates@setD;
```

In[139]:=
```
Length @ setD
```
Out[139]=
```
4598
```

Distance function

Pre-compute the distance function we are going to use

In[140]:=
```
distFunc = Nearest[setD → {"Element", "Index", "Distance"},
    Method → "Scan",
    DistanceFunction → (methodC.(#1[[1 ;; 3]] - #2[[1 ;; 3]]).(#1[[1 ;; 3]] - #2[[1 ;; 3]]) +
        methodCinv.(#1[[4 ;; 6]] - #2[[4 ;; 6]]).(#1[[4 ;; 6]] - #2[[4 ;; 6]]) &)];
```

Iterate

Initialize kernels for parallel searches

In[141]:=
```
numKernels = 6;
CloseKernels[];
LaunchKernels[numKernels];
```

Looping time

Prepare loop variables (just like in d-refinement, but this time there are no FE elements)

In[144]:=
```
(*first material point assignation at random*)
zetaStarIndices = RandomChoice[Range[Length @ setD], numElements];
zetaStar = setD[[zetaStarIndices]];
(*to store index changes over the simulation*)
indexList = {zetaStarIndices};
(*Initial energy (basically inf)*)
dataEnergy = 10^10;
(*to store energy evolution over iterations*)
energyList = {dataEnergy};
methodU = SparseArray[{}, {numNodes * numDOFs, 1}];
methodEtas = SparseArray[{}, {numNodes * numDOFs, 1}];
```

In[151]:=
```
cont = True;
q = 0;
(*loop*)
AbsoluteTiming[
 While[cont,
  q = q + 1;
  If[q > 100, Break[]];
  rhsU = SparseArray[{}, {numDOFs * numNodes, 1}]; (*initialize*)
  rhsF = SparseArray[{}, {numDOFs * numNodes, 1}]; (*initialize*)
  (*build eqns' rhs for this
```

```
 step --------------------------------------------------------------------------
  --*)
(*K u = Sum[w B^T C ϵ*]*)
(*K η = f - Sum[w B^T σ*]*)
Do[
 (*strain contribution*)
 rhsU[[locDOFs[[ii]]]] = elAreas[[ii]] * thickness *
    Transpose[elMatB[[ii]]].methodC.zetaStar[[ii]][[1 ;; 3]] + rhsU[[locDOFs[[ii]]]];
 (*stress contribution*)
 rhsF[[locDOFs[[ii]]]] = elAreas[[ii]] * thickness *
    Transpose[elMatB[[ii]]].zetaStar[[ii]][[4 ;; 6]] + rhsF[[locDOFs[[ii]]]];
 , {ii, 1, numElements}];
(*----------------------------------------------------------------------------*)
(*Solve linear system*)
methodU[[activeDOFs]] =
 LinearSolve[methodK[[activeDOFs, activeDOFs]], rhsU[[activeDOFs]]];
methodEtas[[activeDOFs]] = LinearSolve[methodK[[activeDOFs, activeDOFs]],
  forceVecExt[[activeDOFs]] - rhsF[[activeDOFs]]];
(*Compute new state z_k in E (projection onto E)*)
zeta = Table[Flatten@Join[
    elMatB[[ii]].methodU[[locDOFs[[ii]]]], (*ϵ_e = B_e u_e*)
    zetaStar[[ii]][[4 ;; 6]] +
     Flatten[methodC.elMatB[[ii]].methodEtas[[locDOFs[[ii]]]]](*σ_e = σ_e* + C B_e η_e*)
   ], {ii, 1, numElements}];
(*Project z_k onto D to find
 (z*)_k+1 -----------------------------------------------------------------------
          -----------------*)
Print["Searching..."];
If[numElements > 50,
 searchResults = ParallelMap[distFunc, zeta[[1 ;; -1]]];,
 searchResults = Map[distFunc, zeta[[1 ;; -1]]];
];
(*unpack search
  results -------------------------------------------------------------------
          ------------------------------*)
(*new selected points in D*)
newState = Table[searchResults[[ii]][[1]][[1]], {ii, 1, numElements}];
(*index of the new selected points in D*)
newIndexInD = Table[searchResults[[ii]][[1]][[2]], {ii, 1, numElements}];
(*distance between the new selected points in D and the points in E*)
distances = Table[searchResults[[ii]][[1]][[3]], {ii, 1, numElements}];
(*compute the number of elements that have changed*)
numChanges =
```

```
    Total[Boole[newIndexInD[[#]] ≠ zetaStarIndices[[#]]] & /@ Range[numElements]];
   zetaStarIndices = newIndexInD;
   (*compute new energy difference*)
   newDataEnergy =
    Total[Table[thickness * elAreas[[ii]] * distances[[ii]], {ii, 1, numElements}]];
   If[newDataEnergy < dataEnergy, (*no, keep going: store values and update*)
    cont = True;
    dataEnergy = newDataEnergy;
    AppendTo[energyList, dataEnergy];
    AppendTo[indexList, zetaStarIndices];
    zetaStar = newState,
    (*yes, get outta here *)
    cont = False;
    Break[]];
   (*Print progress*)
   If[True(*Mod[q,50]==1*),
    Print["Step: " <> ToString[q] <> ", # of changes: " <>
      ToString[numChanges] <> ", Data Energy: " <> ToString[Log@dataEnergy]]
   ];
  ](*end while*)
 ]
```

```
Searching ...
Step: 1, # of changes: 2918, Data Energy: 4.65535
Searching ...
Step: 2, # of changes: 2898, Data Energy: 3.33029
Searching ...
Step: 3, # of changes: 2734, Data Energy: 2.18646
Searching ...
Step: 4, # of changes: 2288, Data Energy: 1.35189
Searching ...
Step: 5, # of changes: 1609, Data Energy: 0.865359
Searching ...
Step: 6, # of changes: 907, Data Energy: 0.615899
Searching ...
Step: 7, # of changes: 389, Data Energy: 0.506415
Searching ...
Step: 8, # of changes: 183, Data Energy: 0.471511
Searching ...
Step: 9, # of changes: 60, Data Energy: 0.456364
```

```
Searching ...
Step : 10 , # of changes : 13 , Data Energy : 0.452321
Searching ...
Step : 11 , # of changes : 3 , Data Energy : 0.450108
Searching ...
Step : 12 , # of changes : 6 , Data Energy : 0.446926
Searching ...
Step : 13 , # of changes : 1 , Data Energy : 0.444934
Searching ...
Step : 14 , # of changes : 2 , Data Energy : 0.444139
Searching ...
Step : 15 , # of changes : 7 , Data Energy : 0.442507
Searching ...
Step : 16 , # of changes : 2 , Data Energy : 0.44196
Searching ...
Step : 17 , # of changes : 0 , Data Energy : 0.441896
Searching ...
```

Out[153]= {928.684 , Null}

In[154]:=
```
CloseKernels [];
```

Post-processing

Distance between NR solution and DDCM solution:

In[155]:=
```
distanceNRtoDDCM =
  (Total[Table[thickness * elAreas[[ii]] * distanceSquare [zetaNR[[ii]], zetaStar[[ii]]],
      {ii, 1, numElements }]])^(1/2);
```

as a percentage of distance from NR solution to the origin:

In[156]:=
```
100
 (distanceNRtoDDCM / (Total[Table[thickness * elAreas[[ii]] * distanceSquare [ConstantArray [
          0., {6}], zetaNR[[ii]]], {ii, 1, numElements }]])^(1/2))
```

Out[156]= 6.94308