Using neural networks to bridge scales in metamaterials

This notebook is a companion to “Mesh d-refinement: a data-based computational framework to account for complex material response”. The results presented in the appendix are derived herein.

J. Garcia-Suarez, 2023

Pre-processing

Meshing

Create mesh. Being by loading FEM tools:

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

Define region (exploiting symmetry), see figure 11(a).

```
In[2]:= ell = 0.75 / 1000;(*minimum mesh size, proportional to unit cell size*)
        a = 20 ell; (*crack half-length*)
        H = 200 ell; (*plate half-length*)
```

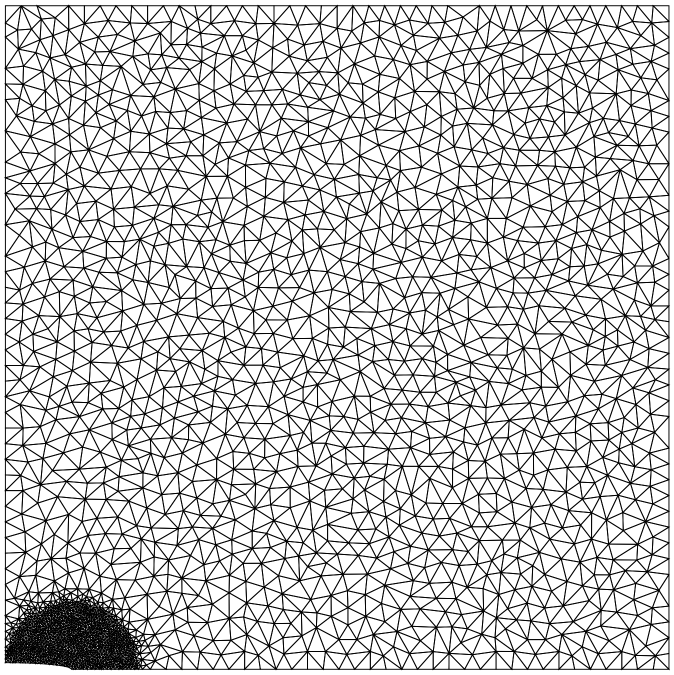
Refinement function (to create refinement around crack tip):

```
In[5]:= f2d = Function[{vertices, area}, Block[{x, y}, {x, y} = Mean[vertices];
            If[(x - a)^2 + y^2 < (20 ell)^2 && area > (ell)^2 / 2, True, False]]];
```

Create mesh:

In[6]:=
```
region = Polygon[{{0, 0}, {0., H}, {H, H}, {H, 0}}];
mesh = ToElementMesh[region, MaxCellMeasure → {"Length" → 10 ell}, "MeshOrder" → 1,
   "MeshElementType" → TriangleElement, MeshRefinementFunction → f2d];
region = RegionDifference[Polygon[{{0, 0}, {0., H}, {H, H}, {H, 0}}],
   Disk[-{0.0, 0.0}, {a, a / 10}]];
mesh = ToElementMesh[region, MaxCellMeasure → {"Length" → 10 ell}, "MeshOrder" → 1,
   "MeshElementType" → TriangleElement, MeshRefinementFunction → f2d];
Show[
 mesh["Wireframe"]
 , Background → White]
```

Out[10]=



Mesh properties

In[11]:=
```
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 boundary conditions)

```
In[17]:= lowerEdge = {};
         Do[
           If[nodes[[ii]][[2]] ≤ 0.1 ell / 2, 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[23]:= thickness = 1.;
```

Material properties

Critical tensile stress (this is used later as threshold to add data to the dataset):

```
In[24]:= limitStress = 0.28 * 10^6;(*Pa*)
```

Define intact (cubic) material in principal material directions: see eq.(14)

```
In[25]:= (*Intact plane-strain tensor*)
         E0 = 3 591 851.; (*this value is obtained from unit cell analysis using DDCM*)
         matCIntact = {{2 E0, E0, 0},
            {E0, 2 E0, 0},
            {0, 0, E0}};
```

Plane strain operator for Mathematica pre-processing

Loading: constant traction at the upper edge

```
In[27]:= p = 16. * 10^4;(*Pa (all units SI)*)
         tractions = NeumannValue[p, {y ≥ H && x ≥ 0}];
```

BCs: symmetry in four planes

Plain-strain operator for cubic material:

```
In[29]:= planeStrainOperator =
           {Inactive[Div][({{0, c12}, {c33, 0}}.Inactive[Grad][v[x, y], {x, y}]), {x, y}] +
              Inactive[Div][({{c11, 0}, {0, c33}}.Inactive[Grad][u[x, y], {x, y}]), {x, y}],
             Inactive[Div][({{0, c12}, {c33, 0}}.Inactive[Grad][u[x, y], {x, y}]), {x, y}] +
              Inactive[Div][({{c33, 0}, {0, c11}}.Inactive[Grad][v[x, y], {x, y}]), {x, y}]} /.
            {c12 → -matCIntact[[1, 2]], c33 → -matCIntact[[3, 3]], c11 → -matCIntact[[1, 1]]};
```

Define PDE:

```
In[30]:= pde2D = planeStrainOperator == {0, tractions};
```

BCs:

```
In[31]:= bcs = {
            DirichletCondition[u[x, y] == 0, x ≤ 0],
            DirichletCondition[v[x, y] == 0, y ≤ 0]};
```

Mathematica linear-elastic solution:

```
In[32]:= {usol, vsol} = NDSolveValue[{pde2D, bcs}, {u, v}, {x, y} ∈ mesh];
```

Stresses and straun

```
In[33]:= epsxx[x_, y_] = D[usol[x, y], x];
         epsyy[x_, y_] = D[vsol[x, y], y];
         epsxy[x_, y_] = 1 / 2 (D[usol[x, y], y] + D[vsol[x, y], x]);
         sigmax[x_, y_] = (λ + 2 μ) D[usol[x, y], x] + λ * D[vsol[x, y], y];
         sigmay[x_, y_] = (λ + 2 μ) D[vsol[x, y], y] + λ * D[usol[x, y], x];
         tau[x_, y_] = μ * (D[usol[x, y], y] + D[vsol[x, y], x]);
```

Pre-processing: solve linear-elastic problem

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

In[39]:=
```
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[53]:=
```
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[57]:= elMatK = Table[
    Transpose[elMatB[[ii]]].matCIntact.elMatB[[ii]] * elAreas[[ii]], {ii, 1, numElements}];
```

Assemble

```
In[58]:= 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[63]:= restrainedDOFsY = Sort[Flatten[{# + numNodes} & /@ lowerEdge]];
activeDOFs = DeleteCases[Range[numDOFs * numNodes], Alternatives @@ restrainedDOFsY];
```

Next the vertical symmetry plane

```
In[65]:= restrainedDOFsX = Sort[Flatten[{#} & /@ leftEdge]];
activeDOFs = DeleteCases[activeDOFs, Alternatives @@ restrainedDOFsX];
```

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

```
In[67]:= forceVecExt = load;
intactU = SparseArray[{}, {numNodes * numDOFs, 1}];
intactU[[activeDOFs]] =
   LinearSolve[globalK[[activeDOFs, activeDOFs]], forceVecExt[[activeDOFs]]];
```

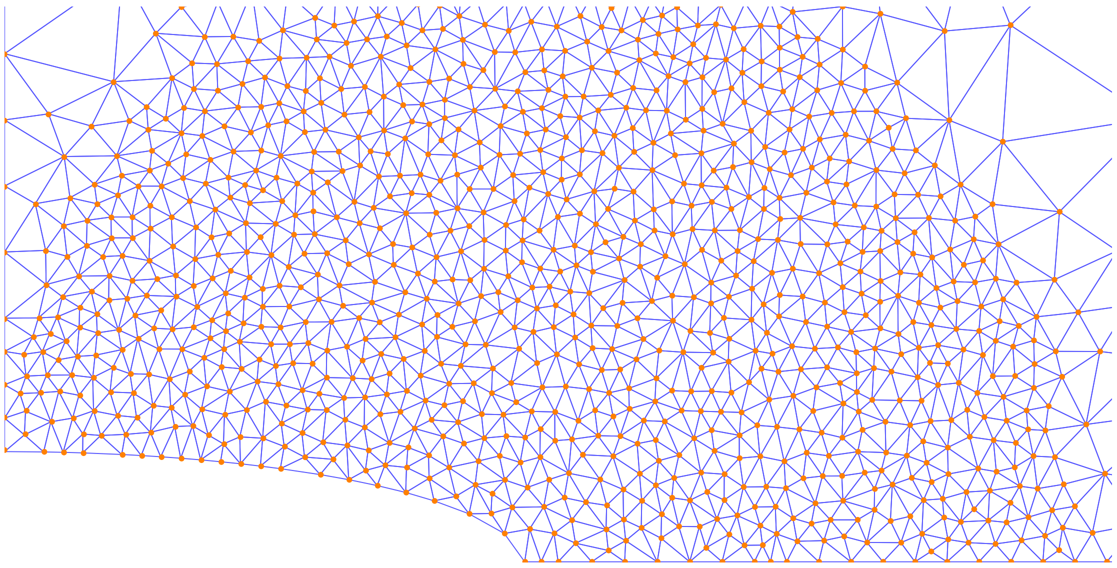
Compare to the Mathematica solution:

```
In[70]:= solU = Table[Flatten@{nodes[[ii]], intactU[[ii]]}, {ii, 1, numNodes}];
sf = 2;
deformedShapeLE =
   Table[nodes[[ii]] + sf * First /@ {intactU[[ii]], intactU[[ii + numNodes]]}, {ii, 1, numNodes}];
```

Compare: mesh represent deformation according to Mathematica, orange dots correspond to deformation computed with **LinearSolve**

In[73]:=
```
Show[
 ElementMeshDeformation [mesh, {usol, vsol}, "ScalingFactor " → sf]["Wireframe "[
   "ElementMeshDirective " -> Directive[EdgeForm[Lighter @Blue], FaceForm[]]]],
 Graphics[{{Orange, Point[deformedShapeLE ]}}]
 , PlotRange → {{0, 2 a}, {0, a}},
 ImageSize → Large,
 AspectRatio → 1 / 2]
```

Out[73]=



Generate values for dataset

As explained in the text, this linear-elastic solution is used to inform the creation of the dataset necessary for d-refinement.

The following lines are used to create “**states**”, phase space points corresponding to each and every element.

This information is processed in a Jupyter notebook to create a dataset.

In[74]:=
```
aboveThresholdQ = SparseArray[{}, {numDOFs * numNodes, 1}];
strainIntact = 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;
  strainIntact[[ii]] = strain;
  aboveThresholdQ[[ii]] = (Abs@stress[[2, 1]] > limitStress);
  , {ii, 1, numElements}];
```

In[78]:=
```
states = Table[Flatten@Join[strainIntact[[ii]], stressIntact[[ii]]], {ii, 1, numElements}];
```

D-refinement

DDCM method’s constant matrix

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

In[79]:=
```
matCinv = Inverse[matCIntact];
methodC = matCIntact;
methodCinv = Inverse[methodC];
methodK = globalK;
elMatC = elMatK;
```

Dataset

Load the dataset of material (unit-cell) non-linear response

In[84]:=
```
numLoadSteps = 29;
```

In[85]:=
```
SetDirectory[NotebookDirectory[]];
dataNL = Import["dataset.csv"];(*make sure that the file
 "dataset.csv" is in the same directory as the notebook, or add path*)
```

Save per-element stress-strain evolution:

In[87]:=
```
elementTrajectories =
  Table[dataNL[[1 + (ii - 1) * numLoadSteps ;; numLoadSteps * ii, All]], {ii, 1, numElements}];
```

Create set D for searches:

```
In[88]:= setD = {};
         Do[(*for each element*)
           Do[(*for each load increment*)
             If[elementTrajectories [[ii, jj, 5]] > 0.9 limitStress ,
               AppendTo[setD, elementTrajectories [[ii, jj, All]]]
             ]
             , {jj, 1, numLoadSteps}];
           , {ii, 1, numElements}];
```

Add the origin:

```
In[90]:= PrependTo[setD, {0., 0., 0, 0., 0., 0.}];
         setD = DeleteDuplicates @setD;
```

Distance function

Pre-compute the distance function we are going to use

```
In[92]:= 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[93]:= 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[100]:= numKernels = 6;
          CloseKernels[];
          LaunchKernels[numKernels];
```

Looping time

Prepare loop variables

```
In[103]:= 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 solve 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[113]:= 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(7)' 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[stress [[2 , 1]] > 0.90 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) = 4940 = 4778 FE elements + 162 DD elements

*--------------------------------------------------------*

# element (total) = 4940 = 4724 FE elements + 216 DD elements

Step: 2, # of changes: 88, Log10 data Energy: -3.26163

*--------------------------------------------------------*

# element (total) = 4940 = 4707 FE elements + 233 DD elements

Step: 3, # of changes: 64, Log10 data Energy: -3.27104
```

```
*-------------------------------------------------------------*
```

```
Out[115]= {8.02651, Null}
```

```
In[116]:= CloseKernels[];
```

Post-processing:

Visualize refinement

Where are the elements above threshold according to LE?

```
In[117]:= aboveThresholdElements = Flatten@Position[aboveThresholdQ, _?(# == True &)];
showAboveThresholdElements = Table[If[
      aboveThresholdQ[[ii]] == True,
      Polygon[nodes[[connectivity[[ii]]]]]
    ], {ii, 1, numElements}];
```

Where are the DD elements?

```
In[119]:= edgeListDD = {};(*create list to highlight refined elements*)
Do[
  list = DeleteDuplicates @
     Flatten[nodes[[#]] & /@ (Sort /@ Permutations[connectivity[[indexDDs[[ii]]]], {2}]), 1];
  AppendTo[edgeListDD, Line@AppendTo[list, First@list]];
  , {ii, 1, Length@indexDDs}]
```

```
In[121]:= showDDelements = Polygon[nodes[[connectivity[[#]]]]] & /@ indexDDs;
```

Compute stresses according to d-ref:

```
In[122]:= strainDref = SparseArray[{}, {numDOFs * numNodes, 1}];
          stressDref = SparseArray[{}, {numDOFs * numNodes, 1}];
          Do[
            If[MemberQ[indexDDs, ii], (*is DD?*)
             ll = First@Flatten@Position[indexDDs, ii];
             strainDref[[ii]] = zetaStar[[ll, 1 ;; 3]];
             stressDref[[ii]] = zetaStar[[ll, 4 ;; 6]];
             , (*no, it is FE*)
             (*displacements in the relevant nodes*)
             Ue = methodU[[locDOFs[[ii]]]];
             matB = elMatB[[ii]];
             (*element strains*)
             strain = matB.Ue;
             (*element stress*)
             stress = matCIntact.strain;
             strainDref[[ii]] = Flatten@strain;
             stressDref[[ii]] = Flatten@stress;
            ]
            , {ii, 1, numElements}];
```

Plot stress along the horizontal symmetry plane (fig. 12)

Find elements in the lower edge

```
In[125]:= tableAux = Table[Min[Last /@ nodes[[connectivity[[ii]]]]] < 0.1 ell / 2, {ii, 1, numElements}];
```

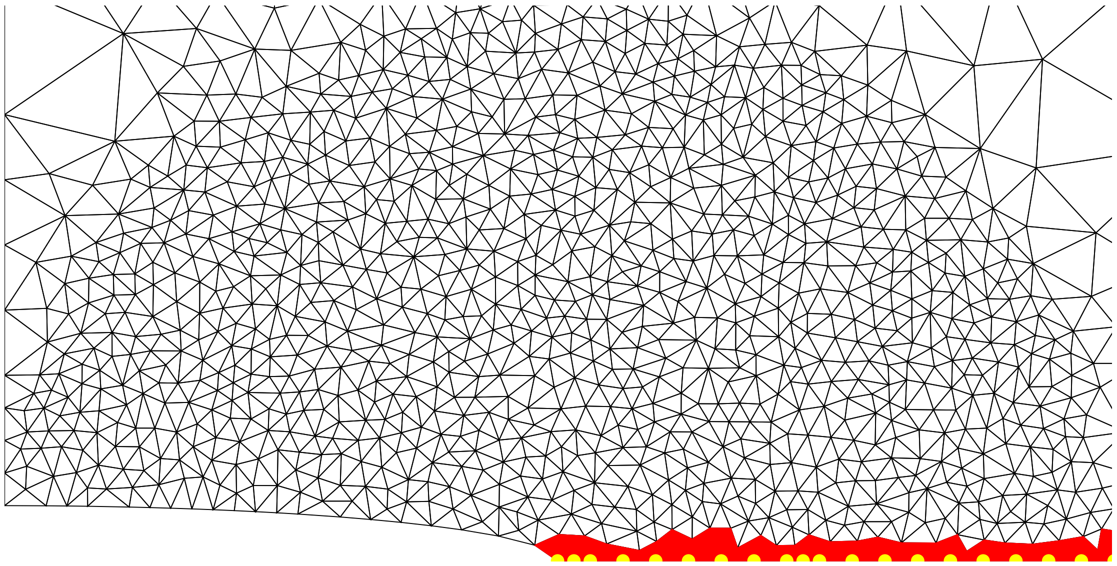
```
In[126]:= elementsLowerEdge = {};
          Do[
           If[tableAux[[ii]] == True, AppendTo[elementsLowerEdge, ii]];
           , {ii, 1, numElements}]
```

```
In[128]:= showElementsLowerEdge = Table[If[MemberQ[elementsLowerEdge, ii],
              Polygon[nodes[[connectivity[[ii]]]]]], {ii, 1, numElements}];
```

Visualize elements (see that we are taking one element over the crack):

In[129]:=
```
Show[
 mesh["Wireframe"],
 Graphics[{
   {Red, showElementsLowerEdge},
   {Yellow, PointSize[Large], Point[nodes[[lowerEdge]]]}
  }],
 PlotRange → {{0, 2 a}, {0, a}},
 Background → White,
 ImageSize → Large]
```

Out[129]=



Find elements that share each node, so that we can compute the stress at the node position by averag-ing the elements that arrive to it:

In[130]:=
```
nodeELemens = Table[{}, numNodes];(*for each node (all the nodes),
what lower-edge elements arrive*)
Do[
 auxList = connectivity[[elementsLowerEdge[[ii]]]];
 (*nodes in the ii-th lower-edge element*)
 Do[
  theNode = auxList[[kk]]; (*each node that appears*)
  AppendTo[nodeELemens[[theNode]], elementsLowerEdge[[ii]]];
  , {kk, 1, 3}]
 , {ii, 1, Length@elementsLowerEdge}]
```

Find max stress to normalize the stress at the edge:

In[132]:=
```
sigmaMax = Max@Table[
    If[Length@stressDref[[ii]] > 1, stressDref[[ii]][[2]], 0], {ii, 1, Length@stressDref}];
```

Compute stresses as the mean of the value associated to each element that arrives to that node

```
In[133]:= stressLowerEdgeV2DIntact = Table[{First@nodes[[lowerEdge[[ii]]]],
              First@Mean[(#[[2]]) & /@ stressIntact[[nodeELemens[[lowerEdge[[ii]]]]]]]},
            {ii, 1, Length@lowerEdge}];

In[134]:= stressLowerEdgeV2Dref = Table[{First@nodes[[lowerEdge[[ii]]]], Mean[
               (#[[2]]) & /@ stressDref[[nodeELemens[[lowerEdge[[ii]]]]]]]}, {ii, 1, Length@lowerEdge}];
```

In this case, we will show it later

NN d-refinement

Load neural-network

Characteristic values defined during training (see notebook “Garcia-Suarez_d-refinement_NNs_2023”)

```
In[135]:= Ec = 3 591 851. ;
          σc = 16. * 10^4;
          ϵc = σc / Ec;
```

Pre-trained w/ the same data as DDCM (again, see notebook “Garcia-Suarez_d-refinement_NNs_2023” for details as to its generation).
Make sure that it is in the same directory or change the path.

```
In[138]:= dNN = Import["trained.mx"];
```

Initialize

```
In[139]:= numTotalDOFs = numNodes * numDOFs;(*the number of dofs*)
          dofs2Solve4 = Join[activeDOFs, activeDOFs + numTotalDOFs];
          (*We still need to solve the double system*)
          numDDelements = 0; (*How many DD elements? Always zero at firtst 0*)
          indexDDs = RandomChoice[Range[numElements], numDDelements];
          indexFEs = Delete[Range[numElements], ArrayReshape[indexDDs, {numDDelements, 1}]];
          numFEelements = numElements - numDDelements; (*how many FE elements*)
          listFEelements = {indexFEs};
```

Iterate

Initialize kernels for parallel searches

Looping time

Initialize some stuff

```
In[143]:= zetaStar = {};(*because in this case there are no DD elements at first*)
          zetaStarIndices = {};
          indexList = {zetaStarIndices };(*these lists are unnecessary in the NN case,
          left here to stress the difference between NN d-ref and DDCM d-ref*)
          (*Tolerance : minimum ratio of the norm of the nodal
            Lagrange multipliers and the one of nodal displacements *)
          tol = 0.01;
          (*to store energy evolution over iterations *)
          energyList = {10^50};
          methodU = SparseArray [{}, {numTotalDOFs , 1}];
          methodEtas = SparseArray [{}, {numTotalDOFs , 1}];
          methodSol = SparseArray [{}, {2 numTotalDOFs , 1}];
          auxIndexFEs = indexFEs ;
          listChanges = {};
```

Initialize kernels for NN projections

```
In[153]:= numKernels = 6;
          CloseKernels [];
          LaunchKernels [numKernels ];

          ••• SubKernels`SubKernels : Timeout for subkernels . Received only 0 of 6 connections .
```

```
In[156]:= cont = True ;
          q = 0;
          (*loop*)
          AbsoluteTiming [While[cont == True ,
            q = q + 1;
            If[q > 10 000 , Break[]];
            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]];
tableFunc = If[numDDelements > 100, ParallelTable, Table];
zeta = tableFunc[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 ---------------------------------------------------------------------
            -----------------*)
newStrain = Flatten[#[[1 ;; 3]]] & /@ zeta;
newStress = σc * (dNN /@ (newStrain / ϵc));
(*unpack search
   results ----------------------------------------------------------------------
```

```
            ------------------------------*)
 (*new selected points in D*)
 newState =
  tableFunc[Flatten @ Join[newStrain[[ii]], newStress[[ii]]], {ii, 1, numDDelements }];
 (*---*)
 zetaStarIndices = newIndexInD ;
 (*compute norm multipliers and compare to displacements '*)
 etaNorm = Norm[methodEtas ];
 uNorm = Norm[methodU];
 AppendTo[energyList , etaNorm];
 If[tol * uNorm < etaNorm , (*no, keep going: store values and update*)
  cont = True ;
  AppendTo[indexList , zetaStarIndices ];
  zetaStar = newState ,
  (*yes, get outta here *)
  cont = False ;
  Print["Finished!"];
  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[stress[[2, 1]] > 0.90 limitStress ,
  (*indeed, delete this FE element from the list and it to the DD bin*)
  indexFEs = DeleteCases [indexFEs , ll];
  AppendTo[indexDDs , ll];
  AppendTo[zetaStar ,
   Flatten @ Join[elMatB[[ii]].methodU[[locDOFs[[ii]]]],
     σc * dNN @(Flatten @(elMatB[[ii]].methodU[[locDOFs[[ii]]]]) / ϵc)]];
 ]
```

```
   , {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["*----------------------------------------------------------*"]
 ]]
```

```
No DD elements , solved  w/ FEM

# element (total) = 4940  = 4778  FE elements  + 162  DD elements

*-------------------------------------------------------------*

Finished !
```

```
Out[158]= {2.37836 , Null}
```

```
In[159]:= CloseKernels [];
```

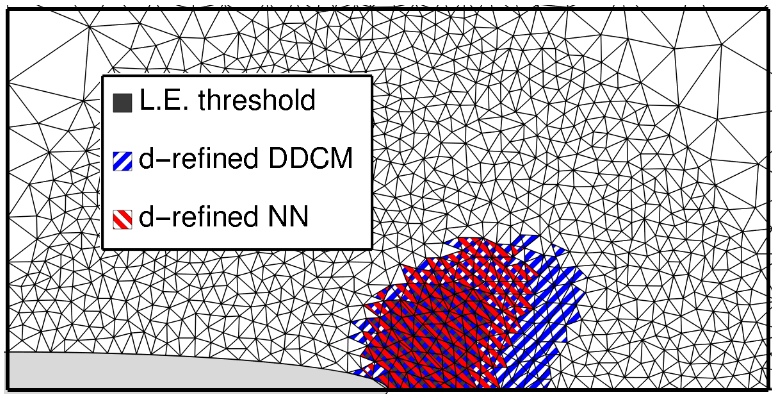
Compare extent of d-refinement in both cases

```
In[160]:= showNNElements  = Table[If[
              MemberQ[indexDDs , ii] == True ,
              Polygon[nodes[[connectivity [[ii]]]]]
            ], {ii, 1, numElements }];
```

```
In[161]:= lg1 = SwatchLegend [{Darker @ Darker @ Gray},
            {Style["L.E.  threshold ", 15, Black]}, LegendMarkers  → {Graphics [{Rectangle []}]}];
          lg2 = SwatchLegend [{Blue}, {Style["d-refined  DDCM", 15, Black]},
            LegendMarkers  → {Graphics [{HatchFilling [π / 4, 2], Rectangle []}]}];
          lg3 = SwatchLegend [{Darker @ Red}, {Style["d-refined  NN", 15, Black]},
            LegendMarkers  → {Graphics [{HatchFilling [-π / 4, 2], Rectangle []}]}];
```

In[164]:=
```
refinement = Legended[
  Show[
   Graphics[{
     {Darker @ Darker @ Gray , showAboveThresholdElements },
     {Blue , HatchFilling [Automatic , 2.25], showDDelements },
     {Darker @ Red , HatchFilling [- π / 4, 2.25], showNNElements },
     {LightGray , EdgeForm [Black], Disk[-{0.0 , 0.0}, {a, a / 10}]}

    }],
   mesh["Wireframe "["MeshElementStyle " → EdgeForm [Black]]],
   PlotRange → {{0, 2 a}, {0, a}},
   AspectRatio → 1 / 2,
   Background → White ,
   Frame → True ,
   FrameStyle → Directive [Black , Thick],
   PlotRangeClipping → True ,
   FrameTicks → None ,
   ImageSize → {400},
   ImagePadding → 2
  ], Placed [Framed [Column @{lg1 , lg2 , lg3},
    FrameMargins → None , Background → White], Scaled [{0.3 , 0.6}]]]
```

Out[164]=


Compute relative distance in phase space between solutions

```
In[165]:= (*DD phase space location: join FEM to DD*)
zetaNN = Table[
    If[MemberQ[indexDDs, ii], (*is DD?*)
     Flatten@Join[elMatB[[ii]].methodU[[locDOFs[[ii]]]],
       σc * dNN@(Flatten@(elMatB[[ii]].methodU[[locDOFs[[ii]]]]) / ϵc)], (*if DD*)
     Flatten@Join[elMatB[[ii]].methodU[[locDOFs[[ii]]]],
       matCIntact.elMatB[[ii]].methodU[[locDOFs[[ii]]]]](*if FEM*)
    ]
    , {ii, 1, numElements}];
```

```
In[166]:= 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[167]:= (*DD phase space location: join FEM to DD*)
zetaDDCM = 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}];
```

```
In[168]:= distanceNNtoDDCM =
   (Total[Table[thickness * elAreas[[ii]] * distanceSquare[zetaNN[[ii]], zetaDDCM[[ii]]],
       {ii, 1, numElements}]])^(1/2);
```

```
In[169]:= 100
  (distanceNNtoDDCM / (Total[Table[thickness * elAreas[[ii]] * distanceSquare[ConstantArray[
            0., {6}], zetaDDCM[[ii]]], {ii, 1, numElements}]])^(1/2))

Out[169]= 0.678095
```

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

Display comparison of hoop stresses along the horizontal symmetry edge, from the crack tip to the vertical edge

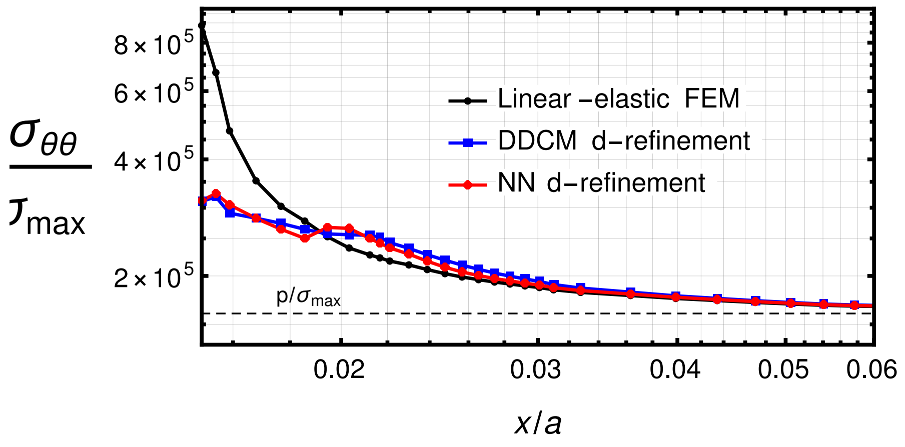
```
In[170]:= stressNN = zetaNN[[1 ;; -1, 5]];
```

```
In[171]:= sigmaYYdrefNN =
   Table[Flatten@{elCentroids[[ii]], 1/sigmaMax stressNN[[ii]]}, {ii, 1, numElements}];
```

```
In[172]:= stressLowerEdgeV2DrefNN  = Table[{First @ nodes[[lowerEdge [[ii]]]],
              Mean[(#) & /@ stressNN [[nodeELemens [[lowerEdge [[ii]]]]]]}, {ii , 1, Length @ lowerEdge }];

In[173]:= stressCrackTip = ListLogLogPlot [{SortBy[stressLowerEdgeV2DIntact , First],
            SortBy[stressLowerEdgeV2Dref , First], SortBy[stressLowerEdgeV2DrefNN , First]},
          Background → White,
          Joined → True,
          GridLines → All,
          PlotStyle →
           {{Black , Thickness [0.005]}, {Blue , Thickness [0.005]}, {Red , Thickness [0.005]}},
          PlotMarkers → {Automatic , Tiny},
          PlotRange → {{a, 4 a}, Automatic },
          Axes → Off,
          Frame → True,
          FrameStyle → Directive [Black , Thick],
          FrameTicksStyle → Directive [Black , 15],
          FrameLabel →
           {Style["x/a", FontSize → 18], Style[Rotate["σθθ/σmax", -π / 2], FontSize → 25]},
          AspectRatio → 1 / 2,
          ImageSize → 550,
          ImagePadding → {{100, 100}, {50, 10}},
          PlotLegends → Placed[{Style["Linear-elastic  FEM", 15, Background → White],
              Style["DDCM  d-refinement ", 15, Background → White],
              Style["NN  d-refinement ", 15, Background → White]}, Scaled[{0.6 , 0.6}]],
          Epilog → {
             {Dashed , Thickness [0.0025], Black , Line[{{Log[1 a], Log[1 p]}, {Log[5 a], Log[1 p]}}]},
             {Text[Style["p/σmax", 12], Scaled[{0.16 , 0.15}]]}
            },
          ImageSize → Medium]
```

Out[173]=


$\frac{\sigma_{\theta\theta}}{\sigma_{max}}$
8×10^5
6×10^5
4×10^5
2×10^5
Linear-elastic FEM
DDCM d-refinement
NN d-refinement
p/σ_{max}
x/a