

3D Vertex Model :: infinite sheet of cells

Initialization

```
In[ ]:= NotebookDirectory[]
```

```
Out[ ]:= D:\LocalData\hashmial\Research\Vertex Model 3D\vertex model sim\
```

importing mesh

mesh parameters and specifying precision

```
In[ ]:= (*DumpGet["C:\\Users\\aliha\\Desktop\\optimize.mx"];*)
DumpGet["D:\\LocalData\\hashmial\\Research\\Vertex
        Model 3D\\vertex model sim\\infinitesheet-noise.mx"];
ptsToIndAssoc = KeyMap[SetPrecision[#, 8] &, ptsToIndAssoc];
indToPtsAssoc = SetPrecision[#, 8] & /@ indToPtsAssoc;
wrappedMat = SetPrecision[#, 8] & /@ wrappedMat;
faceListCoords = SetPrecision[#, 8] & /@ faceListCoords;
```

```
In[ ]:= Names["Global`*"]
```

```
Out[ ]:= {a, cellVertexGrouping, edges, faceListCoords,
        indToPtsAssoc, ptsToIndAssoc, vertexToCell, wrappedMat, xLim, yLim}
```

call dependencies/misc







Launch IGraphM

```
In[ ]:= Needs["IGraphM`"]
```

```
IGraph/M 0.5.1 (October 12, 2020)
Out[ ]:= Evaluate IGDokumentation[] to get started.
```

Launch Subkernels for parallel processing

```
In[ ]:= LaunchKernels[]
ParallelTable[$KernelID, {i, $KernelCount}]
```

```
Out[ ]:= {KernelObject[ Name: local KernelID: 1], KernelObject[ Name: local KernelID: 2],
KernelObject[ Name: local KernelID: 3], KernelObject[ Name: local KernelID: 4],
KernelObject[ Name: local KernelID: 5], KernelObject[ Name: local KernelID: 6]}
```

```
Out[ ]:= {6, 5, 4, 3, 2, 1}
```

simulation variables and parameters

```
In[ ]:= ClearAll[paramFinder];
Options[paramFinder] = {"OrientedFaces" → False};
paramFinder::OrientedFaces =
  "the option should be set to True if the faces of the
   cell are arranged in c.c.w direction.
Default
  →
  False";
```

```
In[ ]:= SetOptions[paramFinder, "OrientedFaces" → True]
```

```
Out[ ]:= {OrientedFaces → True}
```

simulation parameters

```
In[ ]:= time =  $\delta t$  = 0.021; (*time params*)
 $\epsilon_{cc}$  = 1.;  $\epsilon_{co}$  = 0.7; (*params for surface tension*)
 $k_{cv}$  = 14.0;  $V_o$  = 1.0; (*volume elasticity and equilibrium volume*)
 $V_{growth}$  =  $1.0 \times 10^{-4}$ ; (*volume growth-rate*)
 $\delta$  =  $1.0 \times 10^{-3}$ ; (*length threshold for topological transitions*)
growingcellIndices = {90, 52, 266, 286, 96, 289, 233, 360, 91, 127, 163, 39,
  300, 249, 28, 272, 42, 7, 386, 115, 204, 251, 21, 12, 18, 125, 282, 298};
```

```
In[ ]:= ClearAll[orderlessHead];
SetAttributes[orderlessHead, {Orderless}];
```

simulation domain

```
In[ ]:=  $\mathcal{D}$  = Rectangle[{First@xLim, First@yLim}, {Last@xLim, Last@yLim}];
```

local topological information

In[]:=

```

periodicRules::Information =
  "shift the points outside the simulation domain to inside the domain";
transformRules::Information =
  "vector that shifts the point outside the simulation domain back inside";
Clear@periodicRules;
With[{xlim1 = xLim[[1]], xlim2 = xLim[[2]], ylim1 = yLim[[1]], ylim2 = yLim[[2]]},
  periodicRules = Dispatch[{
    {x_ /; x ≥ xlim2, y_ /; y ≤ ylim1, z_} => SetPrecision[{x - xlim2, y + ylim2, z}, 8],
    {x_ /; x ≥ xlim2, y_ /; ylim1 < y < ylim2, z_} => SetPrecision[{x - xlim2, y, z}, 8],
    {x_ /; xlim1 < x < xlim2, y_ /; y ≤ ylim1, z_} => SetPrecision[{x, y + ylim2, z}, 8],
    {x_ /; x ≤ xlim1, y_ /; y ≤ ylim1, z_} => SetPrecision[{x + xlim2, y + ylim2, z}, 8],
    {x_ /; x ≤ xlim1, y_ /; ylim1 < y < ylim2, z_} => SetPrecision[{x + xlim2, y, z}, 8],
    {x_ /; x ≤ xlim1, y_ /; y ≥ ylim2, z_} => SetPrecision[{x + xlim2, y - ylim2, z}, 8],
    {x_ /; xlim1 < x < xlim2, y_ /; y ≥ ylim2, z_} => SetPrecision[{x, y - ylim2, z}, 8],
    {x_ /; x ≥ xlim2, y_ /; y ≥ ylim2, z_} => SetPrecision[{x - xlim2, y - ylim2, z}, 8]
  }];

transformRules = Dispatch[{
  {x_ /; x ≥ xlim2, y_ /; y ≤ ylim1, _} => SetPrecision[{-xlim2, ylim2, 0}, 8],
  {x_ /; x ≥ xlim2, y_ /; ylim1 < y < ylim2, _} => SetPrecision[{-xlim2, 0, 0}, 8],
  {x_ /; xlim1 < x < xlim2, y_ /; y ≤ ylim1, _} => SetPrecision[{0, ylim2, 0}, 8],
  {x_ /; x ≤ xlim1, y_ /; y ≤ ylim1, _} => SetPrecision[{xlim2, ylim2, 0}, 8],
  {x_ /; x ≤ xlim1, y_ /; ylim1 < y < ylim2, _} => SetPrecision[{xlim2, 0, 0}, 8],
  {x_ /; x ≤ xlim1, y_ /; y ≥ ylim2, _} => SetPrecision[{xlim2, -ylim2, 0}, 8],
  {x_ /; xlim1 < x < xlim2, y_ /; y ≥ ylim2, _} => SetPrecision[{0, -ylim2, 0}, 8],
  {x_ /; x ≥ xlim2, y_ /; y ≥ ylim2, _} => SetPrecision[{-xlim2, -ylim2, 0}, 8],
  {___Real} => SetPrecision[{0, 0, 0}, 8]
}];
];

```

In[]:=

```

(*
origcellOrient=<|MapIndexed[First[#2]→#1&, faceListCoords]|>;
boundaryCells=
  With[{ylim1=yLim[[1]],ylim2=yLim[[2]],xlim1=xLim[[1]],xlim2=xLim[[2]]},
    Union[First/@Position[origcellOrient,
      {x_ /; x ≥ xlim2, __} | {x_ /; x ≤ xlim1, __} | {_, y_ /; y ≥ ylim2, __} | {_, y_ /; y ≤ ylim1, __}]/.
      Key[x_] => x]
  ];
wrappedMat=AssociationThread[
  Keys[cellVertexGrouping]→ Map[Lookup[indToPtsAssoc, #]/.periodicRules&,
    Lookup[cellVertexGrouping, Keys[cellVertexGrouping]], {2}]];
*)

```

In[]:=

```

getLocalTopology[ptsToIndAssoc_, indToPtsAssoc_, vertexToCell_,
  cellVertexGrouping_, wrappedMat_, faceListCoords_] [vertices_] :=
  Block[{localTopology = <|>, wrappedcellList = {}, vertcellconns,
    localcellunion, v, wrappedcellpos, vertcs = vertices, r11, r12,
    transVector, wrappedcellCoords, wrappedcells, vertOutofBounds,
    shiftedPt, transvecList = {}, $faceListCoords = Values@faceListCoords,
    vertexQ, boundsCheck, rules, extractcellkeys, vertind,
    cellsconnected, wrappedcellsrem},

```

```

vertexQ = MatchQ[vertices, {__?NumberQ}];
If[vertexQ,
  (vertcellconns =
    AssociationThread[{#}, {vertexToCell[ptsToIndAssoc[#]]}] &@vertices;
    vertcs = {vertices};
    localcellunion = Flatten[Values@vertcellconns]),
  (vertcellconns = AssociationThread[#,
    Lookup[vertexToCell, Lookup[ptsToIndAssoc, #]]] &@vertices;
    localcellunion = Union@Flatten[Values@vertcellconns])
];

If[localcellunion ≠ {},
  AppendTo[localtopology,
    Thread[localcellunion →
      Map[Lookup[indToPtsAssoc, #] &, cellVertexGrouping /@ localcellunion, {2}]]
  ];

(* condition to be an internal edge: both vertices should have 3 neighbours *)
(* if a vertex has 3 cells in its local neighbourhood then the entire
  network topology about the vertex is known → no wrapping required *)
(* else we need to wrap around the vertex because other cells
  are connected to it → periodic boundary conditions *)
With[{vert = #},
  vertind = ptsToIndAssoc[vert];
  cellsconnected = vertexToCell[vertind];
  If[Length[cellsconnected] ≠ 3,
    If[(!RegionMember[Most[vert]]),
      (*Print["vertex inside bounds"];*)
      v = vert;
      With[{x = v[[1]], y = v[[2]]}, boundsCheck =
        (x == xLim[[1]] || x == xLim[[2]] || y == yLim[[1]] || y == yLim[[2]])];

    extractcellkeys = If[boundsCheck,
      {r11, r12} = {v, v /. periodicRules};
      rules = Block[{x$},
        With[{r = r11, s = r12},
          DeleteDuplicates[
            HoldPattern[SameQ[x$, r]] || HoldPattern[SameQ[x$, s]]]
        ]
      ];
      Position@@With[{rule = rules},
        Hold[wrappedMat, x_ /; ReleaseHold@rule, {3}]
      ],
      Position[wrappedMat, x_ /; SameQ[x, v], {3}]
    ];
    (* find cell indices that are attached to the vertex in wrappedMat *)
    wrappedcellpos = DeleteDuplicatesBy[
      Cases[extractcellkeys,
        {Key[p : Except[Alternatives@@Join[localcellunion,
          Flatten@wrappedcellList]]], y_} :> {p, y}],
      First];
    (*wrappedcellpos = wrappedcellpos /.
      {Alternatives@@Flatten[wrappedcellList], _} :> Sequence[];*)
    (* if a wrapped cell has not been considered earlier (i.e. is new)
      then we translate it to the position of the vertex *)

```

```

If[wrappedcellpos ≠ {},
  If[vertexQ,
    transVector = SetPrecision[(v - Extract[$faceListCoords, Replace[#,
      {p_, q_} => {Key[p], q}, {1}]] & /@wrappedcellpos, 8],
    (* call to function is enquiring an edge and not a vertex*)
    transVector =
      SetPrecision[(v - Extract[$faceListCoords, #]) & /@wrappedcellpos, 8]
  ];
  wrappedcellCoords = MapThread[#1 → Map[Function[x,
    SetPrecision[x + #2, 8]], $faceListCoords[[#1]], {2}] &,
    {First /@wrappedcellpos, transVector}];
  wrappedcells = Keys@wrappedcellCoords;
  AppendTo[wrappedcellList, Flatten@wrappedcells];
  AppendTo[transvecList, transVector];
  AppendTo[localtopology, wrappedcellCoords];
],
(* the else clause: vertex is out of bounds *)
(*Print["vertex out of bounds"];*)
vertOutOfBounds = vert;
(* translate the vertex back into mesh *)
transVector = vertOutOfBounds /. transformRules;
shiftedPt = SetPrecision[vertOutOfBounds + transVector, 8];
(* ----- CORE B ----- *)
(* find which cells the
  shifted vertex is a part of in the wrapped matrix *)
wrappedcells = Complement[
  Union@Cases[Position[wrappedMat, x_ /;
    SameQ[x, shiftedPt] || SameQ[x, vertOutOfBounds], {3}],
    x_Key => Sequence @@ x, {2}] /. Alternatives @@
    localcellunion → Sequence[],
  Flatten@wrappedcellList];

(*forming local topology now that we know the wrapped cells *)
If[wrappedcells ≠ {},
  AppendTo[wrappedcellList, Flatten@wrappedcells];
  wrappedcellCoords = AssociationThread[wrappedcells,
    Map[Lookup[indToPtsAssoc, #] &,
      cellVertexGrouping[#] & /@wrappedcells, {2}]];
  With[{opt = (vertOutOfBounds /. periodicRules) | vertOutOfBounds},
    Block[{pos, vertref, transvec},
      Do[
        With[{cellcoords = wrappedcellCoords[cell]},
          pos = FirstPosition[cellcoords /. periodicRules, opt];
          If[Head[pos] === Missing,
            pos = FirstPosition[
              Chop[cellcoords /. periodicRules, 10^-7], Chop[opt, 10^-7]];
          ];
          vertref = Extract[cellcoords, pos];
          transvec = SetPrecision[vertOutOfBounds - vertref, 8];
          AppendTo[transvecList, transvec];
          AppendTo[localtopology,
            cell → Map[SetPrecision[# + transvec, 8] &, cellcoords, {2}]];
        ], {cell, wrappedcells}
      ];
    ];
];

```

```

];
(* to detect wrapped cells not detected by CORE B*)
(* ----- CORE C ----- *)
Block[{pos, celllocs, ls, transvec, assoc, tvecLs = {}, ckey},
  ls = Union@Flatten@Join[cellsconnected, wrappedcells];
  If[Length[ls] ≠ 3,
    pos = Position[faceListCoords, x_ /; SameQ[x, shiftedPt], {3}];
    celllocs = DeleteDuplicatesBy[Cases[pos, Except[{Key[Alternatives@@ls],
      __}]], First] /. {Key[x_], z__} => {Key[x], {z}};
  If[celllocs ≠ {},
    celllocs = Transpose@celllocs;
    assoc = <|
      MapThread[
        (transvec = SetPrecision[vertOutOfBounds -
          Extract[faceListCoords[Sequence@@#1], #2], 8];
          ckey = Identity@@#1;
          AppendTo[tvecLs, transvec];
          ckey → Map[SetPrecision[Lookup[indToPtsAssoc, #] + transvec, 8] &,
            cellVertexGrouping[Sequence@@#1], {2}]
        ) &, celllocs]
      |>;
    AppendTo[localtopology, assoc];
    AppendTo[wrappedcellList, Keys@assoc];
    AppendTo[transvecList, tvecLs];
  ];
];
];
];
];
];
] & /@ vertcs;

transvecList = Which[
  MatchQ[transvecList, {{{__?NumberQ}}}], First[transvecList],
  MatchQ[transvecList, {{__?NumberQ} ..}], transvecList,
  True, transvecList /. {x___, {p : {__?NumberQ} ..}, y___} => {x, p, y}
];
{localtopology, Flatten@wrappedcellList, transvecList}
];

```

In[]:=

```

Clear@outerCellsFn;
outerCellsFn::Information = "the function finds the cells at the boundary";
outerCellsFn[faceListCoords_, vertexToCell_, ptsToIndAssoc_] :=
  With[{xlim1 = xLim[[1]], xlim2 = xLim[[2]], ylim1 = yLim[[1]], ylim2 = yLim[[2]]},
    Block[{boundaryCells, bcells, temp, res},
      temp = <|MapIndexed[First[#2] → #1 &, faceListCoords] |>;
      boundaryCells = Union[First /@ Position[temp,
        {x_ /; x ≥ xlim2, _} | {x_ /; x ≤ xlim1, _} |
        {_, y_ /; y ≥ ylim2, _} | {_, y_ /; y ≤ ylim1, _}] /. Key[x_] → x];
      bcells = KeyTake[faceListCoords, boundaryCells];
      res = Union@(Flatten@Lookup[vertexToCell,
        Lookup[ptsToIndAssoc,
          DeleteDuplicates@Cases[bcells,
            {x_ /; x ≥ xlim2, _} | {x_ /; x ≤ xlim1,
              _} | {_, y_ /; y ≥ ylim2, _} | {_, y_ /; y ≤ ylim1, _}, {3}]
          /. periodicRules
        ]
      ] ~Join~ boundaryCells);
      res
    ]
  ];

```

face triangulation and associated $f(x)$'s

In[]:=

```

Clear[triangulateFaces];
triangulateFaces::Information =
  "the function takes in cell faces and triangulates them";
triangulateFaces[faces_] := Block[{edgelen, ls, mean},
  (If[Length[#] ≠ 3,
    ls = Partition[#, 2, 1, 1];
    edgelen = Norm[SetPrecision[First[#] - Last[#], 8]] & /@ ls;
    mean = Total[edgelen * (Midpoint /@ ls)] / Total[edgelen];
    mean = mean ~SetPrecision~ 8;
    Map[Append[#, mean] &, ls],
    {#}
  ] & /@ faces
];

```

In[]:=

```

Clear[meanFaces];
meanFaces = Compile[{{faces, _Real, 2}},
  Block[{facepart, edgelen, mean},
    facepart = Partition[faces, 2, 1];
    AppendTo[facepart, {facepart[[-1, -1]], faces[[1]]}];
    edgelen = Table[Norm[SetPrecision[First@i - Last@i, 8]], {i, facepart}];
    mean = Total[edgelen * (Mean /@ facepart)] / Total[edgelen];
    mean],
  RuntimeAttributes → {Listable}, CompilationTarget → "C", RuntimeOptions → "Speed",
  CompilationOptions → {"InlineExternalDefinitions" → True}
]
(*Needs["CompiledFunctionTools`"]*)
(*CompilePrint[meanFaces];*)
Clear[triangulateToMesh];
triangulateToMesh::Information =
  "the function takes in cell faces and triangulates them";
triangulateToMesh[faces_] := Block[{mf, partfaces},
  mf = SetPrecision[meanFaces@faces, 8];
  partfaces = Partition[#, 2, 1, 1] & /@ faces;
  MapThread[
    If[Length[#] ≠ 3,
      Function[x, Join[x, {#2}]] /@ #1,
      {#[[All, 1]]}
    ] &, {partfaces, mf}
];

```

Out[]:= CompiledFunction[ Argument count: 1
Argument types: {{_Real, 2}}]

In[]:=

```

Clear[areaTriFn];
areaTriFn::Information = "areaTriFn finds the area of a triangle";
areaTriFn = Compile[{{face, _Real, 2}},
  Block[{v1, v2},
    v2 = face[[2]] - face[[1]];
    v1 = face[[3]] - face[[1]];
    0.5 Norm@Cross[v2, v1]
  ], CompilationTarget → "C", RuntimeOptions → "Speed"
]

```

Out[]:= CompiledFunction[ Argument count: 1
Argument types: {{_Real, 2}}]


```

In[ ]:= Clear[meanTri];
meanTri::Information = "meanTri returns the centroid of the triangle";
meanTri = Compile[{{faces, _Real, 2}},
  Mean@faces,
  CompilationTarget → "C", RuntimeAttributes → {Listable},
  Parallelization → True, RuntimeOptions → "Speed"
]

```

Out[]:= CompiledFunction[ Argument count: 1
Argument types: {{_Real, 2}}]

```

In[ ]:= Clear[triNormal];
triNormal::Information = "triNormal returns the normal of a triangle face";
triNormal = Compile[{{ls, _Real, 2}},
  Block[{res},
    res = Partition[ls, 2, 1];
    Cross[res[[1, 1]] - res[[1, 2]], res[[2, 1]] - res[[2, 2]]]
  ], CompilationTarget → "C", RuntimeAttributes → {Listable},
  RuntimeOptions → "Speed"
]

```

Out[]:= CompiledFunction[ Argument count: 1
Argument types: {{_Real, 2}}]

```

In[ ]:= Clear[lenEdge];
lenEdge::Information = "lenEdge returns the length of an edge";
lenEdge = Compile[{{edge1, _Real, 1}, {edge2, _Real, 1}},
  Norm[SetPrecision[edge1 - edge2, 8]],
  CompilationTarget → "C",
  CompilationOptions → {"InlineExternalDefinitions" → True},
  RuntimeOptions → "Speed"
]

```

Out[]:= CompiledFunction[ Argument count: 2
Argument types: {{_Real, 1}, {_Real, 1}}]

centroid/volume for polyhedral cells

```
In[ ]:= Clear@volumePolyhedHelper;
volumePolyhedHelper = Compile[{{triFaces, _Real, 2}},
  Block[{V1, V2, V3},
    {V1, V2, V3} = Transpose[triFaces];
    Cross[V1, V2].V3
  ], CompilationTarget -> "C", RuntimeAttributes -> {Listable},
  RuntimeOptions -> "Speed"
]
```

```
Clear@volumePolyhedra;
volumePolyhedra::Information =
  "returns the volume of the triangulated polyhedral cell";
volumePolyhedra[facecollec_] :=
  1. / 6 Total[Flatten@volumePolyhedHelper[facecollec]]];
```

Out[]:= CompiledFunction[ Argument count: 1
Argument types: {{_Real, 2}}]

```
In[ ]:= Clear@centroidPolyhedraHelper;
centroidPolyhedraHelper = Compile[{{triFaces, _Real, 2}},
  Block[{V1, V2, V3, normal},
    {V1, V2, V3} = triFaces;
    normal = triNormal@triFaces;
    normal ((V1 + V2) ^ 2 + (V2 + V3) ^ 2 + (V3 + V1) ^ 2)
  ], CompilationTarget -> "C", RuntimeAttributes -> {Listable},
  RuntimeOptions -> "Speed"
]
```

```
Clear@centroidPolyhedra;
centroidPolyhedra::Information =
  "returns the centroid of the triangulated polyhedral cell";
centroidPolyhedra[polyhed_, vol_] :=
  1 / (2. vol) × 1. / 24 Total[Flatten[centroidPolyhedraHelper@polyhed, 1]]];
```

Out[]:= CompiledFunction[ Argument count: 1
Argument types: {{_Real, 2}}]

```
In[ ]:= Clear@parPolyhedProp;
parPolyhedProp::Information =
  "returns the centroid and volume of the polyhedral cells
  in the mesh using built-in modules";
parPolyhedProp[polyhedra_] := Module[{vals = Values@polyhedra, ls = {}, rcent, rvol},
  SetSharedVariable[ls];
  ParallelDo[AppendTo[ls, {RegionCentroid[i], Volume[i]}], {i, vals}];
  Transpose@ls
];
```

polyhedron collision avoidance

centroids of the local polyhedral neighbourhood

the version below uses the shift vector and is more optimized in terms of speed

```
In[ ]:= Clear[cellCentroids];
cellCentroids::Information =
  "the function yields the centroids of the polyhedral cells
  present in the local cell topology";
cellCentroids[polyhedCentAssoc_, keystopo_, shiftvec_] :=
  Block[{assoc = <| |>, regcent, counter},
    AssociationThread[Keys@keystopo →
      KeyValueMap[
        Function[{key, cellassoc},
          If[KeyFreeQ[shiftvec, key],
            Lookup[polyhedCentAssoc, cellassoc],
            If[KeyFreeQ[shiftvec[key], #],
              regcent = polyhedCentAssoc[#],
              regcent = polyhedCentAssoc[#] + shiftvec[key][#];
            regcent
          ] & /@ cellassoc
        ]
      ], keystopo]
  ];
```



form local topology by shifting cells

```
In[ ]:= Clear[cellTranslator];
cellTranslator[facelsc_, keyslocaltopo_, shiftVecA_, vertkeys_] :=
  Block[{shiftvec, svkeys, cellids},
    <|Table[
      cellids = keyslocaltopo[i];
      i → If[KeyFreeQ[shiftVecA, i],
        {AssociationThread[cellids, Lookup[facelsc, cellids]], {}, {}},
        shiftvec = shiftVecA[i];
        svkeys = Keys@shiftvec;
        {AssociationThread[cellids,
          If[FreeQ[svkeys, #],
            facelsc[#],
            Map[Function[fc, SetPrecision[fc + shiftvec[#], 8]], facelsc[#], {2}]
          ] & /@ cellids], svkeys, Values@shiftvec}
      ], {i, vertkeys}] |>
  ];
```

surface ▽

```
In[ ]:= ClearAll[rotateSourceVertOrderFn];
rotateSourceVertOrderFn = Compile[{{sourcept, _Real, 1}, {ls, _Real, 3}},
  Block[{temp, vec, pos = 0},
    (vec = #;
      If[Chop[Compile`GetElement[vec, 1] - sourcept, 10^-8] == {0., 0., 0.},
        vec,
        Which[
          Chop[Compile`GetElement[vec, 2] - sourcept, 10^-8] == {0., 0., 0.},
            pos = 2,
          Chop[Compile`GetElement[vec, 3] - sourcept, 10^-8] == {0., 0., 0.},
            pos = 3
        ];
      RotateLeft[vec, pos - 1]
    ) & /@ ls
  ], CompilationTarget -> "C", RuntimeOptions -> "Speed", CompilationOptions ->
  {"ExpressionOptimization" -> True, "InlineExternalDefinitions" -> True}];
```

```
In[ ]:= ClearAll@surfaceGrad;
surfaceGrad::Information =
  "surfaceGrad takes in arguments and computes the surface gradient about a point";
With[{epcc =  $\epsilon_{cc}$ , epco =  $\epsilon_{co}$ },
  surfaceGrad = Compile[{{point, _Real, 1}, {opentr, _Real, 3},
    {normal0, _Real, 2}, {closedtr, _Real, 3}, {normalC, _Real, 2}},
    Block[{ptTri, source = point, normal, u1, u2, cross,
      openS = {0., 0., 0.}, closedS = {0., 0., 0.}},
      Do[
        ptTri = opentr[[i]];
        normal = normal0[[i]];
        {u1, u2} = {ptTri[[2]], ptTri[[-1]]};
        cross = Cross[normal, u2 - u1];
        openS += (0.5 * cross), {i, Length@normal0}
      ];
      Do[
        ptTri = closedtr[[j]];
        normal = normalC[[j]];
        {u1, u2} = {ptTri[[2]], ptTri[[-1]]};
        cross = Cross[normal, u2 - u1];
        closedS += (0.5 * cross), {j, Length@normalC}
      ];
      epcc * closedS + epco * openS
    ], CompilationTarget -> "C", RuntimeOptions -> "Speed",
    CompilationOptions ->
      {"ExpressionOptimization" -> True, "InlineExternalDefinitions" -> True}]
  ]
```

Out[]:= CompiledFunction[  Argument count: 5
Argument types: {{_Real, 1}, {_Real, 3}, {_Real, 2}, {_Real, 3}, {_Real, 2}}]

```

In[ ]:= (* (* Alternative implementation *)
ClearAll@surfaceGrad;
surfaceGrad::Information=
"surfaceGrad takes in arguments and computes the surface gradient about a point";
With[{epcc= $\epsilon_{cc}$ , epco= $\epsilon_{co}$ },
surfaceGrad=Compile[{{point,_Real,1},{opentri,_Real,3},
{norm0,_Real,2},{closedtri,_Real,3},{normC,_Real,2}},
Block[{openSCont,closedSCont,ptTri,source=point,normal,target,facept,cross},
openSCont=Total@MapThread[
(ptTri=#1;normal=#2;
cross=If[Chop[ptTri[[1]]-source,10^-8]=={0.,0.,0.},
{target,facept}={ptTri[[2]],ptTri[[-1]]};
Cross[normal,facept-target],
{target,facept}={ptTri[[1]],ptTri[[-1]]};
Cross[normal,target-facept]
];0.5cross)&,{opentri,norm0}];
closedSCont=Total@MapThread[
(ptTri=#1;normal=#2;
cross=If[Chop[ptTri[[1]]-source,10^-8]=={0.,0.,0.},
{target,facept}={ptTri[[2]],ptTri[[-1]]};
Cross[normal,facept-target],
{target,facept}={ptTri[[1]],ptTri[[-1]]};
Cross[normal,target-facept]
];0.5cross)&,{closedtri,normC}];
epcc closedSCont + epco openSCont
],CompilationOptions->{"ExpressionOptimization"->True},CompilationTarget->"C"]
]
*)

```

volumetric ∇

```

In[ ]:= Clear@volumeGrad;
volumeGrad::Information="volumeGrad computes the volume gradient about a point";
volumeGrad[normalsAssoc_, cellids_, assocTri_, polyhedVols_, growingCellIds_] :=
Block[{celltopology, gradV, vol, growindkeys},
gradV = With[{nA = normalsAssoc, aT = assocTri},
Table[
celltopology = aT[cell];
(1. / 3.) Total[(areaTriFn[#]  $\times$  nA[#]) & /@ celltopology ], {cell, cellids}
];
vol = AssociationThread[cellids -> ConstantArray[V0, Length@cellids]];
growindkeys =
Replace[Intersection[cellids, growingCellIds], k_Integer -> {Key[k]}, {1}];
vol = Values@If[growindkeys  $\neq$  {},
MapAt[(1 + Vgrowth time) # &, vol, growindkeys], vol];
kcv Total[(polyhedVols - vol) / (vol^2) gradV
];

```

vertex ∇

```
In[ ]:= Clear@gradientVertex;
gradientVertex::Information =
  "determines the sum of the gradients of all the potentials about a vertex";
gradientVertex[ind_, indToPtsAssoc_, triDistAssoc_, vertTriNormalpairings_,
  associatedtri2_, topoF_, polyhedVol_] := Block[{sgterm, vgterm,
  opentri, closetri, norm0, normC, pt, triAssoc,
  normalAssoc, keys, assocTri, polyvol},
  pt = indToPtsAssoc[ind];
  triAssoc = triDistAssoc[ind];
  normalAssoc = vertTriNormalpairings[ind];
  keys = Keys@First@topoF[ind];
  assocTri = associatedtri2[ind];
  polyvol = Lookup[polyhedVol, keys];
  {opentri, closetri} = {triAssoc[1], triAssoc[2]};
  {norm0, normC} = {Lookup[normalAssoc, opentri], Lookup[normalAssoc, closetri]};
  opentri = rotateSourceVertOrderFn[pt, opentri];
  closetri = rotateSourceVertOrderFn[pt, closetri];
  sgterm = surfaceGrad[pt, opentri, norm0, closetri, normC];
  vgterm = volumeGrad[normalAssoc, keys, assocTri, polyvol, growingcellIndices];
  - (sgterm + vgterm)
]
```

pair boundary points for computing ∇

```
In[ ]:= Clear@boundaryPtsPairing;
boundaryPtsPairing::Information =
  "the function pairs the points at the boundaries
  with corresponding mirror points";
boundaryPtsPairing[vertexToCell_, indToPtsAssoc_, ptsToIndAssoc_] :=
  Block[{outerpts, mirrorpairs, pt, mirror,
  temp = KeyMap[Chop[#, 10^-7] &, ptsToIndAssoc]},
  outerpts = Keys@Select[vertexToCell, Length[#] > 3 &];
  mirrorpairs = <|
    (pt = Lookup[indToPtsAssoc, #];
    If[
      pt[[1]] < xLim[[1]] ||
      pt[[1]] > xLim[[2]] || pt[[2]] < yLim[[1]] || pt[[2]] > yLim[[2]],
      (*mirror=ptsToIndAssoc[pt/.periodicRules];*)
      mirror = Lookup[ptsToIndAssoc,
        {pt /. periodicRules}, temp[Chop[(pt /. periodicRules), 10^-7]]];
      # -> mirror,
      Nothing]) & /@ outerpts
  |> // KeySort;
  Map[Sort@*Flatten] [List @@@ Normal@GroupBy[Normal@mirrorpairs, Last -> First]]
  ];
```

```

In[ ]:= (*pointPairingFn[mirrorpairAssoc_] := Block[{ls, partner, pos, pcheck},
  ls = {};
  Scan[
    (partner = mirrorpairAssoc[#];
     pcheck = FreeQ[ls, partner];
     If[pcheck,
       AppendTo[ls, {#, partner}]];
     If[!pcheck && FreeQ[ls, #],
       pos = First@Position[ls, partner, {2}];
       ls[[pos]] = ls[[pos]] ~ Append ~ #) &,
    Keys[mirrorpairAssoc]];
  ls
];*)

```

integrating mesh

adjust gradient

```

In[ ]:= Clear@adjustGrad;
adjustGrad::Information =
  "the function ensures that the boundary points and their
  mirror points have the same gradient";
adjustGrad[grad_, bptpairs_] := Block[{vals, g = grad},
  Scan[
    keys ↦ (
      vals = SetPrecision[Mean@Lookup[grad, keys], 8];
      Scan[(g[#] = vals) &, keys]
    ), bptpairs];
  g
];

```

paramFinder

```

In[ ]:= paramFinder[indToPtsAssoc_, $CVG_, keyslocaltopo_, shiftVecAssoc_,
  vertKeys_, OptionsPattern[]] := Block[{$faceListAssoc, $topo,
  localtrimesh, tempassoc = <||>, meshed, $assoctri, trimesh,
  polyhed, polyhedcent, $polyhedVol, cellcent, normals, normNormals,
  centTri, signednormals, $vertTriNormpair, opencloseTri,
  $triDistAssoc, trianglemembers, ckeys, signs, dim,
  faceorientedQ = OptionValue["OrientedFaces"]},
  (* adjust params for the new geometry M2 *)
  $faceListAssoc = Map[Lookup[indToPtsAssoc, #] &, $CVG, {2}];
  $topo = cellTranslator[$faceListAssoc, keyslocaltopo, shiftVecAssoc, vertKeys];
  (*Print[Union@
    Values[Length@DeleteDuplicates@Flatten[Values[First@#], 2] & /@ $topo]]];*)
  localtrimesh = Map[
    If[KeyFreeQ[tempassoc, #],
      meshed = triangulateToMesh[#];
      tempassoc[#] = meshed;
      meshed, tempassoc[#]
    ] &, $topo[[All, 1]], {2}];

```

```

$assoctri = AssociationThread[vertKeys,
  (vert ↦ With[{pt = Chop[indToPtsAssoc[vert], 10^-7]},
    <|GroupBy[Chop[Flatten[#, 1], 10^-7],
      MemberQ[#, x_ /; x === pt] &][True] & /@ localtrimesh[vert] |>
    ])/@vertKeys];
opencloseTri = Flatten[Values@#, 1] & /@ $assoctri;
trimesh = triangulateToMesh /@ $faceListAssoc;
If[faceorientedQ,
  $polyhedVol = volumePolyhedra /@ trimesh;
  polyhedcent = <|
    KeyValueMap[#1 → centroidPolyhedra[#2, $polyhedVol[#1]] &, trimesh] |>,
  (*else*)
  polyhed = Polyhedron@* (Flatten[#, 1] &) /@ trimesh;
  ckeys = Keys@trimesh;
  {polyhedcent, $polyhedVol} =
    AssociationThread[ckeys, #] & /@ parPolyhedProp[polyhed]
  (*polyhedcent=RegionCentroid/@polyhed;
  $polyhedVol=AssociationThread[Keys[polyhed]→Volume[Values@polyhed]]*)
];
cellcent = cellCentroids[polyhedcent, keyslocaltopo, shiftVecAssoc];
normals = Map[SetPrecision[#, 8] &, (triNormal@Values@# & /@ $assoctri)];
normNormals = Map[Normalize, normals, {3}];
centTri =
  Map[SetPrecision[#, 8] &, <|# → meanTri[Values[$assoctri@#]] & /@ vertKeys |>];
signednormals = AssociationThread[vertKeys,
  Map[
    MapThread[
      #2 Sign@MapThread[Function[{x, y}, (y - #1).x], {#2, #3}] &,
      {cellcent[#, normNormals[#, centTri[#]]} &, vertKeys];
  ];
  ];
opencloseTri =
  MapThread[MapAt[Function[coords, {coords[[1]], coords[[3]], coords[[2]]}],
    #1, Position[Flatten[#2, 1], -1]] &, {opencloseTri, signs}];

$assoctri = AssociationThread[
  vertKeys → MapThread[
    (dim = Values[Map[Length] [#1]);
    AssociationThread[Keys[#1] → TakeList[#2, dim]] &,
    {$assoctri, opencloseTri}]
  ];
];

$vertTriNormpair = <|
  # → <|Thread[opencloseTri[#] → Flatten[signednormals@#, 1]] |> & /@ vertKeys |>;
$triDistAssoc = (trianglemembers = #;
  GroupBy[GatherBy[trianglemembers, Intersection],
    Length, Flatten[#, 1] &]) & /@ opencloseTri;
{$triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol}
];

```


Integrator

In[]:=

```
Clear@RK5Integrator;
RK5Integrator::Information = "the module comprises the
  Runge-Kutta Order 5 (RK5) scheme for integrating the mesh";
RK5Integrator[indToPtsAssoc_, topo_, cellVertG_, $vertexToCell_, ptsToIndAssoc_] :=
  Block[{grad1, grad2, grad3, grad4, grad5, grad6, $indToPtsAssoc = indToPtsAssoc,
    $triDistAssoc, $vertTriNormpair, $assoctri, $topo = topo, $polyhedVol,
    $CVG = cellVertG, shiftVecAssoc, keyslocaltopo, vertKeys = Keys@indToPtsAssoc,
    $indToPtsAssocOrig = indToPtsAssoc, bptpairs, vals},
    (*computed once at the start of the computation*)
    keyslocaltopo = Keys@*First /@ topo;
    shiftVecAssoc = Association /@
      Map[Apply[Rule], Thread /@ Select[(#[[2 ;; 3]]) & /@ topo, # ≠ {} &], {2}];
    bptpairs = boundaryPtsPairing[$vertexToCell, indToPtsAssoc, ptsToIndAssoc];
    {$triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol} =
      paramFinder[$indToPtsAssoc, $CVG, keyslocaltopo, shiftVecAssoc, vertKeys];
    (*compute the gradient of the original mesh M1 *)
    grad1 = AssociationThread[vertKeys,
      SetPrecision[Table[gradientVertex[i, $indToPtsAssoc,
        $triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol],
        {i, vertKeys}], 8]];
    grad1 = adjustGrad[grad1, bptpairs];
    (*displace vertices by the numerical gradient*)
    $indToPtsAssoc = <|KeyValueMap[
      #1 → SetPrecision[#2 + 0.25 grad1[#1] δt, 8] &, $indToPtsAssocOrig] |>;
    (*adjust and compute params for the intermediate geometry M2*)
    {$triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol} =
      paramFinder[$indToPtsAssoc, $CVG, keyslocaltopo, shiftVecAssoc, vertKeys];
    (*compute the gradient for M2*)
    grad2 = AssociationThread[vertKeys,
      SetPrecision[Table[gradientVertex[i, $indToPtsAssoc,
        $triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol],
        {i, vertKeys}], 8]];
    grad2 = adjustGrad[grad2, bptpairs];
    (*displace vertices by the numerical gradient*)
    $indToPtsAssoc = <|
      KeyValueMap[#1 → SetPrecision[#2 + 0.125 (grad1[#1] + grad2[#]) δt, 8] &,
        $indToPtsAssocOrig] |>;
    (*adjust and compute params for the intermediate geometry M3*)
    {$triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol} =
      paramFinder[$indToPtsAssoc, $CVG, keyslocaltopo, shiftVecAssoc, vertKeys];
    (*compute the gradient for M3*)
    grad3 = AssociationThread[vertKeys,
      SetPrecision[Table[gradientVertex[i, $indToPtsAssoc,
        $triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol],
        {i, vertKeys}], 8]];
    grad3 = adjustGrad[grad3, bptpairs];
    (*displace vertices by the numerical gradient*)
    $indToPtsAssoc = <|
      KeyValueMap[#1 → SetPrecision[#2 - (0.5 grad2[#1] - grad3[#]) δt, 8] &,
        $indToPtsAssocOrig] |>;
    (*adjust and compute params for the intermediate geometry M4*)
    {$triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol} =
```

```

    paramFinder[$indToPtsAssoc, $CVG, keyslocaltopo, shiftVecAssoc, vertKeys];
    (*compute the gradient for M4*)
    grad4 = AssociationThread[vertKeys,
      SetPrecision[Table[gradientVertex[i, $indToPtsAssoc,
        $triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol],
        {i, vertKeys}], 8]];
    grad4 = adjustGrad[grad4, bptpairs];
    (*displace vertices by the numerical gradient*)
    $indToPtsAssoc = <|KeyValueMap[
      #1 → SetPrecision[#2 + ((3. / 16) grad1[#1] + (9. / 16) grad4[#]) δt, 8] &,
      $indToPtsAssocOrig] |>;
    (*adjust and compute params for the intermediate geometry M5*)
    {$triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol} =
      paramFinder[$indToPtsAssoc, $CVG, keyslocaltopo, shiftVecAssoc, vertKeys];
    (*compute the gradient for M5*)
    grad5 = AssociationThread[vertKeys,
      SetPrecision[Table[gradientVertex[i, $indToPtsAssoc,
        $triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol],
        {i, vertKeys}], 8]];
    grad5 = adjustGrad[grad5, bptpairs];
    (*displace vertices by the numerical gradient*)
    $indToPtsAssoc = <|KeyValueMap[#1 →
      SetPrecision[#2 + ((2. / 7) grad2[#1] - (3. / 7) grad1[#] +
        (12. / 7) grad3[#] - (12. / 7) grad4[#] + (8. / 7) grad5[#]) δt, 8] &,
      $indToPtsAssocOrig] |>;
    (*adjust and compute params for the intermediate geometry M6*)
    {$triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol} =
      paramFinder[$indToPtsAssoc, $CVG, keyslocaltopo, shiftVecAssoc, vertKeys];
    (*compute the gradient for M6*)
    grad6 = AssociationThread[vertKeys,
      SetPrecision[Table[gradientVertex[i, $indToPtsAssoc,
        $triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol],
        {i, vertKeys}], 8]];
    grad6 = adjustGrad[grad6, bptpairs];
    (*displace vertices to get next geometry*)
    $indToPtsAssoc = Map[SetPrecision[#, 8] &] [<|
      KeyValueMap[#1 → #2 + (1. / 90) (7. grad1[#1] + 32. grad3[#] + 12. grad4[#] +
        32. grad5[#] + 7. grad6[#]) δt &, $indToPtsAssocOrig] |>]
];

```

topological network operations $[\Delta \leftrightarrow I] \wedge [I \leftrightarrow \Delta]$

visualizing mesh

```
In[ ]:= Options[displayMesh] = Options[Graphics3D] ~ Join ~ {"opacity" → Opacity[1]};
displayMesh::"header" =
  "display the mesh of polyhedrons and colour by various properties;
  colourmaps include:
    \"volume\", \"surfacearea\", \"height\", \"centroid\", \"faces\", \"surface/volume
    \", \"vertices\", \"edges\";
  use None for displaying mesh without colouring";
displayMesh[indToPtsAssoc_, cellVertexG_, colourmap_ : "volume",
  opts : OptionsPattern[]] := Block[{mesh, col, plt, cellfaces, func},
  func = Function[x, ColorData["Rainbow"][x], Listable];
  cellfaces = Map[Lookup[indToPtsAssoc, #] &, cellVertexG, {2}];
  mesh = Values[Polyhedron@Flatten[triangulateFaces@#, 1] & /@ cellfaces];
  plt = If[colourmap != None,
    col = func@Rescale@Switch[colourmap,
      "volume", Volume@mesh,
      "surfacearea" | "surface", SurfaceArea@mesh,
      "height", Values[Max[#[[All, All, 3]]] & /@ cellfaces],
      "centroid", (RegionCentroid /@ mesh) [[All, 3]],
      "faces", Values[Map[Length]@cellfaces],
      "surface/volume", (SurfaceArea@mesh) / (Volume@mesh),
      "vertices",
      Values[Length*DeleteDuplicates@Flatten[#, 1] & /@ cellfaces],
      "edges", Values[(x ↦ Length@DeleteDuplicatesBy[Flatten[
        Partition[#, 2, 1, 1] & /@ x, 1], Sort]) /@ cellfaces]
    ];
  Thread[{col, mesh}],
  mesh
];
Graphics3D[{OptionValue["opacity"], plt}, ImageSize → OptionValue[ImageSize]]
];
```

```
In[ ]:= convertToPolyhed[table_, CVG_] := (Polyhedron@Flatten[triangulateToMesh[#, 1] & /@
  Map[Lookup[table, #] &, CVG, {2}]]);
```

```
In[ ]:= visualizeMesh[table_, CVG_] :=
  Graphics3D[Values@convertToPolyhed[table, CVG], ImageSize → 750];
```

main f(x)

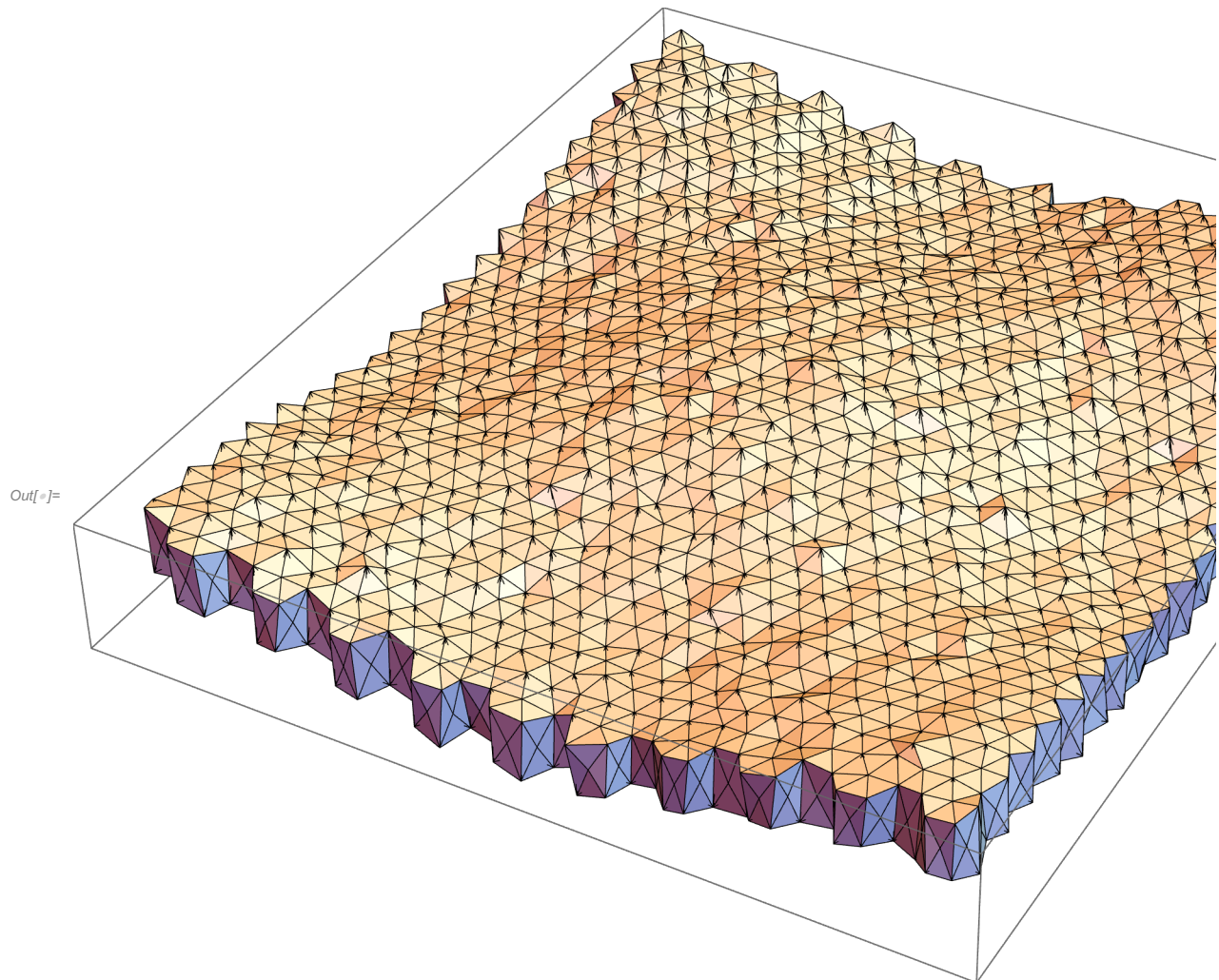
1. instantiate geometry
2. seed growing cells in the tissue
3. initiate counter, and time-step = δt
4. main loop (? termination condition is fulfilled):
 - counter += 1
 - if Mod[counter, 10] is 0 :
 - check for ($l \rightarrow \Delta$) & ($\Delta \rightarrow l$) transitions

- perform *RK5* mesh integration
- time += δt

5. visualize mesh

starting mesh

```
In[ ]:= visualizeMesh[indToPtsAssoc, cellVertexGrouping]
```



test module (single time-step)

termination condition:

$\text{Abs}[\eta(\text{pts-coords @ } t + 1 - \text{pts-coords @ } t)/\delta t] - \nabla$ at pts @ $t < \text{threshold}$
 → the condition should hold true for all pts

(* initialize time *)

```
In[ ]:= time = 1 * δt;
```

In[]:=

```

Clear@runSingleStep;
runSingleStep::Information = "test vertex model by running a single time step";
runSingleStep[ptsToIndAssoc_, indToPtsAssoc_, vertexToCell_, cellVertexGroup_] :=
  With[{ylim1 = yLim[[1]],
        ylim2 = yLim[[2]], xlim2 = xLim[[2]]},
    Block[{$ptsToIndAssoc = ptsToIndAssoc,
           $indToPtsAssoc = indToPtsAssoc, $vertexToCell = vertexToCell,
           $cellVertexGroup = cellVertexGroup, $wrappedMat,
           $wrappedMatTrim, $faceListCoords, $topo, vertKeys,
           cellKeys, boundarycells, bptpairs, outerptscloudTab,
           $indToPtsAssocOrig = indToPtsAssoc},
      cellKeys = Keys[$cellVertexGroup];
      vertKeys = Keys@$indToPtsAssoc;
      $faceListCoords = Map[Lookup[$indToPtsAssoc, #] &, $cellVertexGroup, {2}];
      $wrappedMat = With[{keys = Keys[$cellVertexGroup]},
        AssociationThread[keys → Map[Lookup[$indToPtsAssoc, #] /. periodicRules &,
          Lookup[$cellVertexGroup, keys], {2}]]];
      boundarycells = outerCellsFn[$faceListCoords, $vertexToCell, $ptsToIndAssoc];
      $wrappedMatTrim = $wrappedMat ~KeyTake~ boundarycells;
      $topo = <| # → (getLocalTopology[$ptsToIndAssoc,
        $indToPtsAssoc, $vertexToCell, $cellVertexGroup,
        $wrappedMatTrim, $faceListCoords] [$indToPtsAssoc[#]]) & /@ vertKeys |>;
      Echo["RK5 integration Started"];
      $indToPtsAssoc = RK5Integrator[$indToPtsAssoc,
        $topo, $cellVertexGroup, $vertexToCell, $ptsToIndAssoc];
      $ptsToIndAssoc = AssociationMap[Reverse, $indToPtsAssoc];
      Echo["RK5 integration Done"];
      (*
      (* avoiding polyhedral collisions: self and cross *)
      {$indToPtsAssoc,$ptsToIndAssoc}=
        selfIntersectMod[$indToPtsAssoc,$indToPtsAssocOrig,$cellVertexGroup];
      Echo["checked self-intersection"];
      bptpairs=boundaryPtsPairing[$vertexToCell,$indToPtsAssoc,$ptsToIndAssoc];
      outerptscloudTab=
        pointcloudFn[bptpairs,boundarycells,$indToPtsAssoc,$cellVertexGroup];
      {$indToPtsAssoc,$ptsToIndAssoc}=crossIntersectMod1[outerptscloudTab,
        $indToPtsAssoc,$ptsToIndAssoc,$indToPtsAssocOrig,$cellVertexGroup];
      Echo["checked cross-intersection"];
      *)
      $indToPtsAssoc
    ]
  ];

```

```
(*Needs["CompiledFunctionTools`"]*)
```

```

In[ ]:= resSingleStep = runSingleStep[ptsToIndAssoc,
  indToPtsAssoc, vertexToCell, cellVertexGrouping]; // AbsoluteTiming

```

```
» RK5 integration Started
```

```
» RK5 integration Done
```

```
Out[ ]:= {20.6426, Null}
```


In[]:=



20.5953 (2000/0.021) sec to days

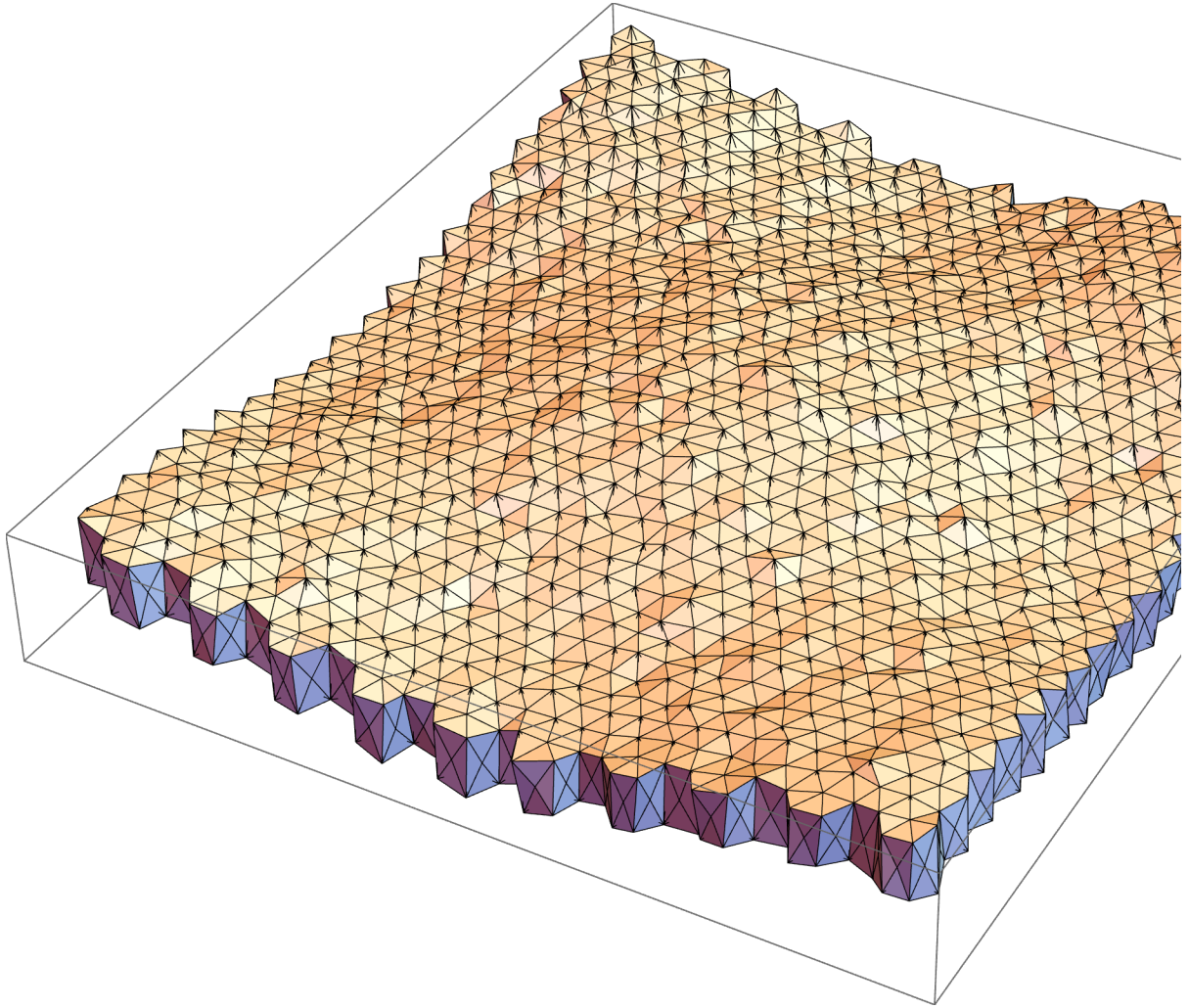


$$\text{UnitConvert}\left[20.5953 \times \frac{2000}{0.021} \text{ s}, \text{"Days"}\right]$$

Out[]:= 22.7021 days

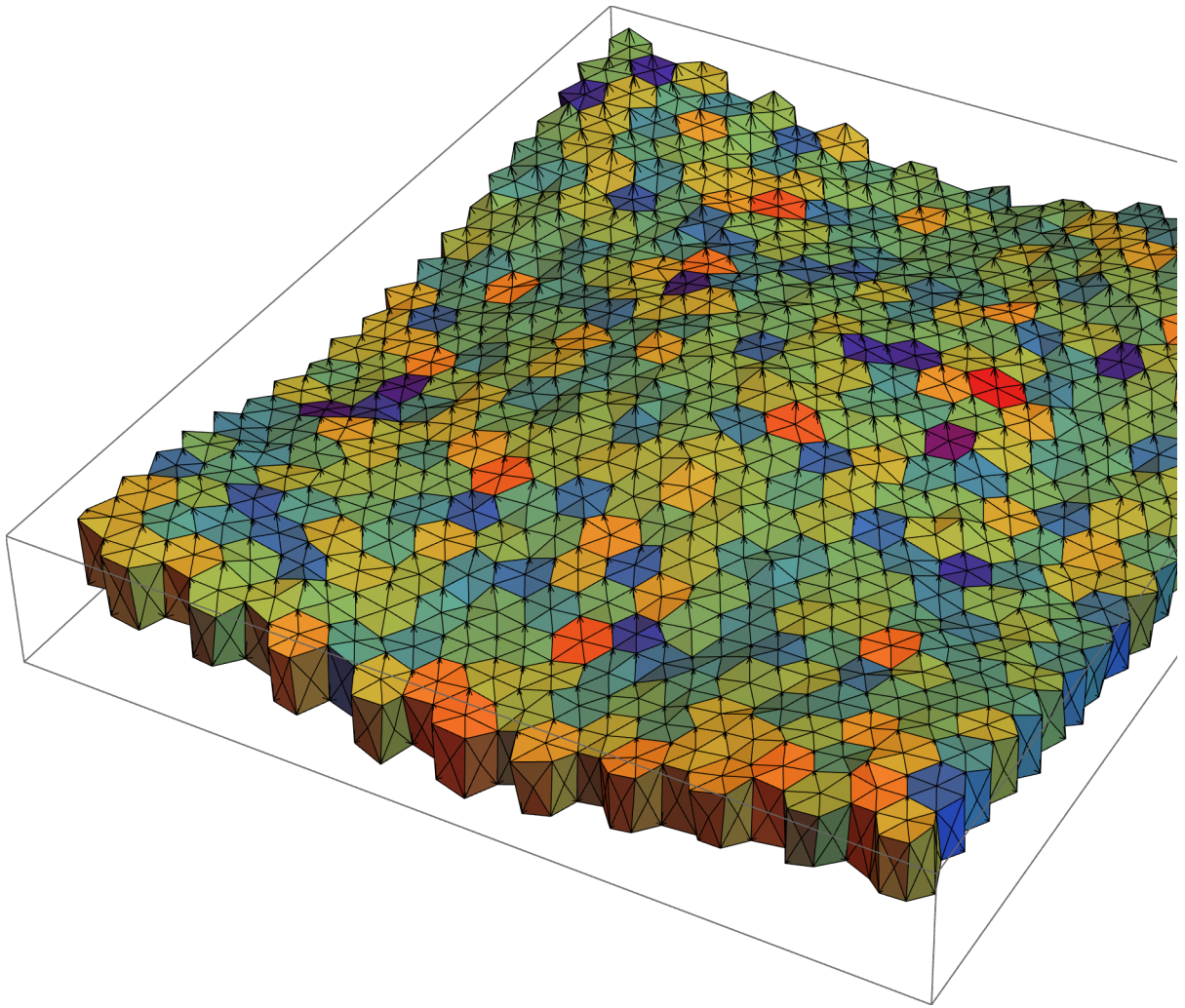
In[]:= resSingleStep~visualizeMesh~cellVertexGrouping

Out[]:=



```
In[ ]:= displayMesh[ressSingleStep, cellVertexGrouping, "volume", ImageSize -> {750, Automatic}]
```

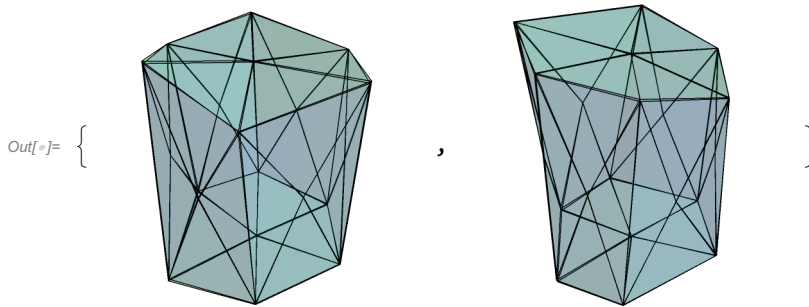
Out[]:=



```

In[ ]:= SeedRandom[10];
Function[ind,
  With[{CVG = cellVertexGrouping[ind]},
    Show[
      Graphics3D[{Opacity[0.12], Green, (Polyhedron@Flatten[triangulateToMesh[#], 1] &@
        (Lookup[indToPtsAssoc, #] & /@ CVG))}],
      Graphics3D[{Opacity[0.12], Blue, (Polyhedron@Flatten[triangulateToMesh[#], 1] &@
        (Lookup[resSingleStep, #] & /@ CVG))}],
      ImageSize → Small, Boxed → False]
    ]
  ] /@ {1, RandomChoice[growingcellIndices]}

```



run model (without collision detection)

Initialize time by 'counter' and assign value to the stop counter 'endc'. If your last run stopped at counter 7, then the starting value for the next run will be 8;

```

In[ ]:= Clear@runModel;
runModel[ptsToIndAssoc_, indToPtsAssoc_, vertexToCell_, cellVertexGroup_] :=
  Block[{counter = 1, $indToPtsAssoc = indToPtsAssoc, $ptsToIndAssoc = ptsToIndAssoc,
    $cellVertexGroup = cellVertexGroup, $vertexToCell = vertexToCell,
    $wrappedMat, $faceListCoords, cellIds = Keys@cellVertexGroup,
    vertKeys, topo, boundarycells, $edges, $wrappedMatBound,
    $indToPtsAssocOrig = indToPtsAssoc, outerptscloudTab, bptpairs, endc = 20},
    time = counter * dt;
    Echo[time, "time: "];

    While[counter ≤ endc,
      Echo[counter, "counter: "];
      $faceListCoords = Map[Lookup[$indToPtsAssoc, #] &, $cellVertexGroup, {2}];
      $wrappedMat =
        AssociationThread[cellIds → Map[Lookup[$indToPtsAssoc, #] /. periodicRules &,
          Lookup[$cellVertexGroup, cellIds], {2}]];

      If[Mod[counter, 10] == 0,
        PrintTemporary[" ----- perform topological operations ----- "];
        $edges = Flatten[Map[Partition[#, 2, 1, 1] &, Map[Lookup[$indToPtsAssoc, #] &,
          Values[$cellVertexGroup], {2}], {2}], 2] // DeleteDuplicatesBy[Sort];
        (*I→Δ*)
        {$edges, $indToPtsAssoc, $ptsToIndAssoc, $cellVertexGroup, $vertexToCell,
          $wrappedMat} = ItoΔ[$edges, $faceListCoords, $indToPtsAssoc,
            $ptsToIndAssoc, $cellVertexGroup, $vertexToCell, $wrappedMat];
        $faceListCoords = Map[Lookup[$indToPtsAssoc, #] &, $cellVertexGroup, {2}];

```



```

(*Δ→I*)
{$edges, $indToPtsAssoc, $ptsToIndAssoc, $cellVertexGroup, $vertexToCell,
  $wrappedMat} = ΔtoI[$edges, $faceListCoords, $indToPtsAssoc,
  $ptsToIndAssoc, $cellVertexGroup, $vertexToCell, $wrappedMat];
$faceListCoords = Map[Lookup[$indToPtsAssoc, #] &, $cellVertexGroup, {2}],
Nothing
];

PrintTemporary[" ----- construct local topology ----- "];
vertKeys = Keys@$indToPtsAssoc;
cellIds = Keys@$cellVertexGroup;
boundarycells = outerCellsFn[$faceListCoords, $vertexToCell, $ptsToIndAssoc];
$wrappedMatBound = $wrappedMat ~ KeyTake ~ boundarycells;
topo = <|# → (getLocalTopology[$ptsToIndAssoc,
  $indToPtsAssoc, $vertexToCell, $cellVertexGroup,
  $wrappedMatBound, $faceListCoords] [$indToPtsAssoc[#]]) & /@
  vertKeys|>;
(*Print[Union@Values[Length@DeleteDuplicates@Flatten[Values[First@#],2]&/@
  topo]]];*)

PrintTemporary[" ----- Integrate mesh (RK5) ----- "];
$indToPtsAssoc = RK5Integrator[$indToPtsAssoc,
  topo, $cellVertexGroup, $vertexToCell, $ptsToIndAssoc];
$ptsToIndAssoc = AssociationMap[Reverse, $indToPtsAssoc];

(*DumpSave["C:\\Users\\aliha\\Desktop\\vertmodelres\\datasave_"<>
  ToString[counter]<>"_"<>ToString[time]<>".mx",
  {$indToPtsAssoc,$ptsToIndAssoc,$cellVertexGroup,$vertexToCell}]]];*)

time += δt; counter += 1;
PrintTemporary["< update time > : " <> ToString[time]];
];
Print[" ----- iterations ended successfully ----- "];
{$ptsToIndAssoc, $indToPtsAssoc, $vertexToCell, $cellVertexGroup}
];

```

```

In[ ]:= {res1, res2, res3, res4} = runModel[ptsToIndAssoc,
  indToPtsAssoc, vertexToCell, cellVertexGrouping]; // AbsoluteTiming

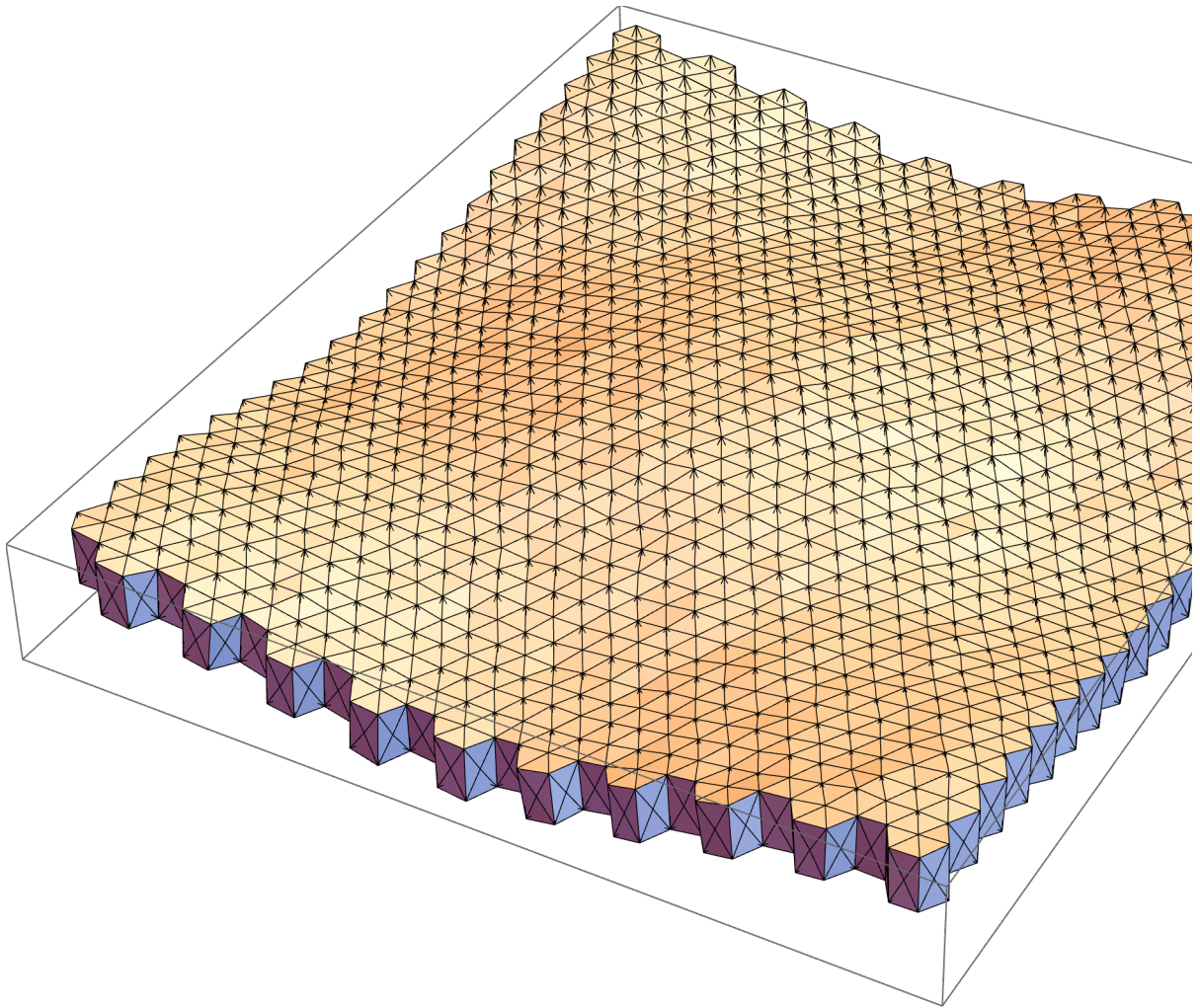
```

```
» time: 0.021
» counter: 1
» counter: 2
» counter: 3
» counter: 4
» counter: 5
» counter: 6
» counter: 7
» counter: 8
» counter: 9
» counter: 10
» counter: 11
» counter: 12
» counter: 13
» counter: 14
» counter: 15
» counter: 16
» counter: 17
» counter: 18
» counter: 19
» counter: 20

----- iterations ended successfully -----
Out[ ]= {461.014, Null}
```

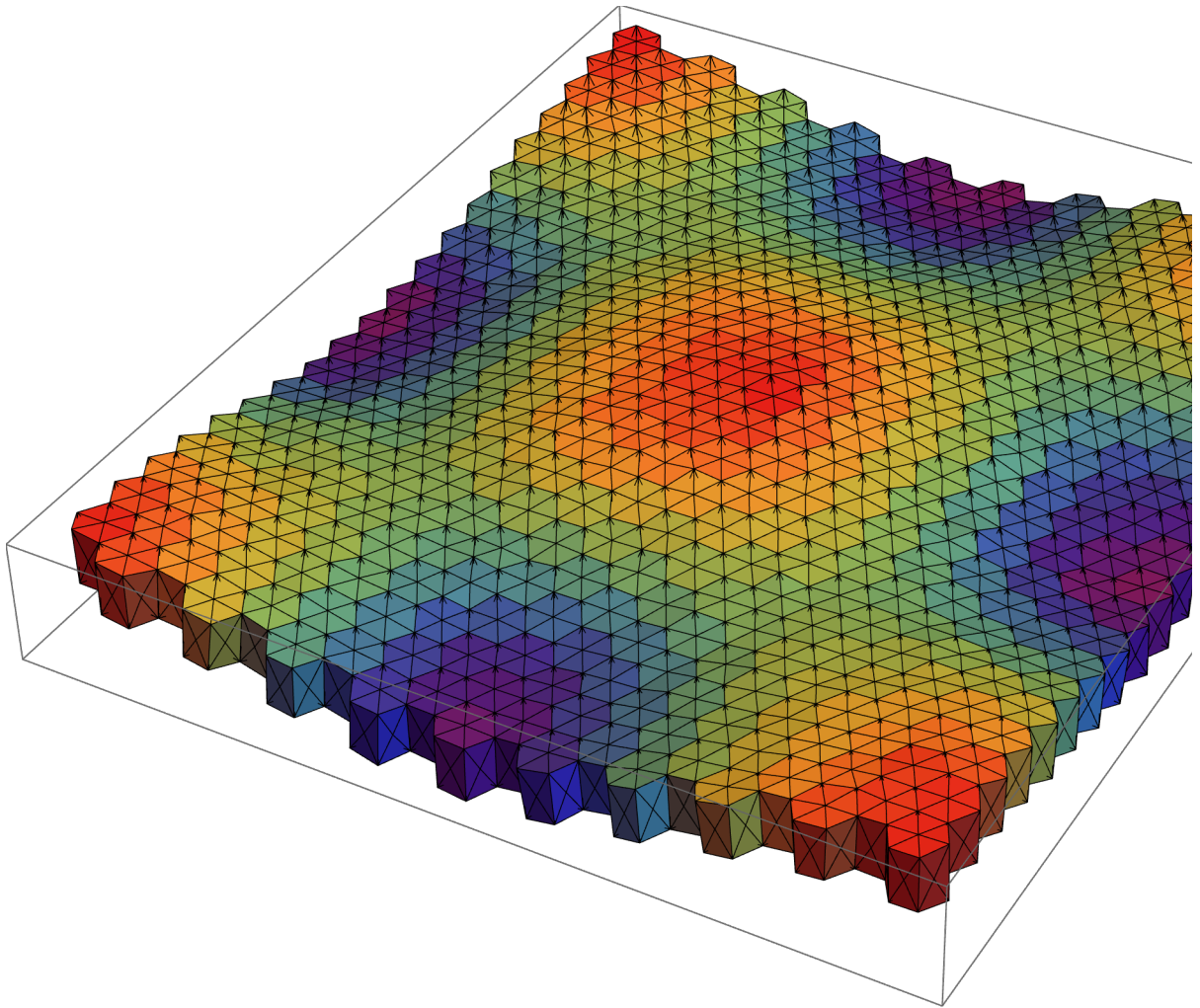
```
In[ ]:= visualizeMesh[res2, res4]
```

Out[]:=



```
In[ ]:= displayMesh[res2, res4, "centroid", ImageSize -> {750, Automatic}]
```

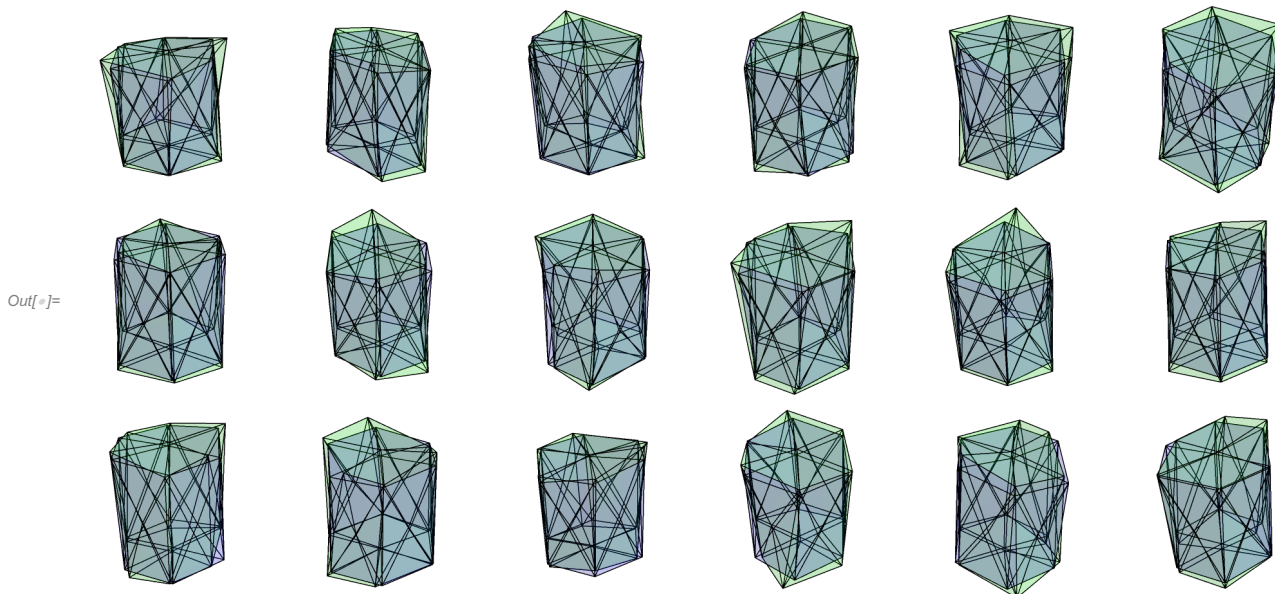
Out[]:=





```

In[ ]:= SeedRandom[1];
Grid[
  Partition[
    Function[ind,
      With[{CVG = cellVertexGrouping[ind]},
        Show[
          Graphics3D[{Opacity[0.12], Blue, (Polyhedron@Flatten[triangulateToMesh[#], 1] &@
            (Lookup[res2, #] & /@ CVG))}],
          Graphics3D[{Opacity[0.12], Green, (Polyhedron@Flatten[triangulateToMesh[#],
            1] &@ (Lookup[indToPtsAssoc, #] & /@ CVG))}],
          ImageSize -> Tiny, Boxed -> False]
        ]
      ] /@ RandomSample[Range[400], 30], 10]
]

```



In[]:=  454/20 (2000/0.021) sec to days 

UnitConvert[(454 / 20) * 95 238.1 s , "Days"]

Out[]:= 25.022 days

run model (collision detection module 1)

run model (collision detection module 2)