

# 3D Vertex Model :: infinite sheet of cells

## Initialization

### importing mesh

*mesh parameters and specifying precision*

```
(*DumpGet["C:\\Users\\aliha\\Desktop\\optimize.mx"];*)
```

```
In[ ]:= DumpGet["C:\\Users\\aliha\\Desktop\\wolfram-vertex-3D\\add  
noise to mesh\\infinitesheet-noise.mx"];  
ptsToIndAssoc = KeyMap[SetPrecision[#, 8] &, ptsToIndAssoc];  
indToPtsAssoc = SetPrecision[#, 8] & /@ indToPtsAssoc;  
wrappedMat = SetPrecision[#, 8] & /@ wrappedMat;  
faceListCoords = SetPrecision[#, 8] & /@ faceListCoords;
```

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

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

### call dependencies/misc

*Launch IGraphM*

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

IGraph/M 0.3.108 (December 17, 2018)

Evaluate IGDokumentation[] to get started.

*Launch Subkernels for parallel processing*

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

```
Out[ ]:= {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 = 21/1000.;  $\delta t$  = 0.022; (*time params*)
 $\epsilon_{cc}$  = 1.;  $\epsilon_{co}$  = 1.; (*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};
```

### *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";
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 < 0., y_ /; y ≤ ylim1, z_} ⇒ SetPrecision[{x + xlim2, y + ylim2, z}, 8],
    {x_ /; x < 0., y_ /; ylim1 < y < ylim2, z_} ⇒ SetPrecision[{x + xlim2, y, z}, 8],
    {x_ /; x < 0., y_ /; y > ylim2, z_} ⇒ SetPrecision[{x + xlim2, y - ylim2, z}, 8],
    {x_ /; 0. < 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 < 0, y_ /; y ≤ ylim1, _} ⇒ SetPrecision[{xlim2, ylim2, 0}, 8],
  {x_ /; x < 0, y_ /; ylim1 < y < ylim2, _} ⇒ SetPrecision[{xlim2, 0, 0}, 8],
  {x_ /; x < 0, y_ /; y > ylim2, _} ⇒ SetPrecision[{xlim2, -ylim2, 0}, 8],
  {x_ /; 0 < 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]],xlim2=xLim[[2]]},
  Union[First/@Position[origcellOrient,
    {x_/;x≥xlim2,___}|{x_/;x<0,___}|{_,y_/;y>ylim2,___}|{_,y_/;y≤ylim1,___}]/.
    Key[x_]⇒ x]
  ];
wrappedMat=AssociationThread[
  Keys[$cellVertexGrouping]→ Map[Lookup[interIndToPtsAssoc,#]/.periodicRules&,
    Lookup[$cellVertexGrouping,Keys[$cellVertexGrouping]],{2}]]];
*)
```

In[ ]:=

```
getLocalTopology::Information =
  "given a vertex or a vertex-pair the function yields the local cell topology";
getLocalTopology[ptsToIndAssoc_, indToPtsAssoc_, vertexToCell_,
  cellVertexGrouping_, wrappedMat_, faceListCoords_][vertices_] :=
  Module[{localTopology = <|>, wrappedcellList = {}, vertcellconns,
    localcellunion, vertInBounds, v, wrappedcellpos, vertcs = vertices,
    transVector, wrappedcellCoords, wrappedcells, vertOutOfBounds,
    shiftedPt, transvecList = {}, $faceListCoords = Values@faceListCoords,
```

```

vertexQ},
vertexQ = MatchQ[vertices, {__?NumberQ}];
If[vertexQ,
vertcellconns =
  AssociationThread[{#}, {vertexToCell[ptsToIndAssoc[#]]}] &@vertices;
verts = {vertices};
localcellunion = Flatten[Values@vertcellconns],
(* this will yield vertex → cell indices connected in the local mesh *)
vertcellconns =
  AssociationThread[#, Lookup[vertexToCell, Lookup[ptsToIndAssoc, #]]] &@vertices;
localcellunion = Union@Flatten[Values@vertcellconns];
];
(* condition to be an internal
edge: both vertices should have 3 or more neighbours *)
(*Print["All topology known"];*)
(* the cells in the local mesh define the entire network topology →
no wrapping required *)
(* else cells need to be wrapped because other cells are
connected to the vertices → periodic boundary conditions *)
With[{vert = #},
  If[(D~RegionMember~Most[vert]) &&
    ! (vert[[1]] == xLim[[2]] || vert[[2]] == yLim[[2]])],
    (* the vertex has less than 3 neighbouring cells but
the vertex is within bounds *)
    (*Print["vertex inside bounds with fewer than 3 cells"];*)
    v = vertInBounds = vert;
    (* find cell indices that are attached to the vertex in wrappedMat *)
    wrappedcellpos = DeleteDuplicatesBy[
      Cases[Position[wrappedMat, x_ /; SameQ[x, v], {3}],
        {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[$faceListCoords,
          Replace[#, {p_, q__} :> {Key[p], q}, {1}]]] & /@wrappedcellpos, 8],
        (*the main function is enquiring an edge and not a vertex*)
        transVector =
          SetPrecision[(v - Extract[$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];
(*local topology here only has wrapped cell *)
],
(*Print["vertex out of bounds"];*)
(* else vertex is out of bounds *)
vertOutOfBounds = vert;
(* translate the vertex back into mesh *)
transVector = vertOutOfBounds /. transformRules;
shiftedPt = SetPrecision[vertOutOfBounds + transVector, 8];
(* find which cells the vertex is a part of in the wrapped matrix *)
wrappedcells = Complement[
  Union@Cases[Position[wrappedMat, x_ /; SameQ[x, shiftedPt], {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)},
    Block[{pos, vertref, transvec},
      Do[
        With[{cellcoords = wrappedcellCoords[cell]},
          pos = FirstPosition[cellcoords /. periodicRules, opt];
          vertref = Extract[cellcoords, pos];
          transvec = SetPrecision[vertOutOfBounds - vertref, 8];
          AppendTo[transvecList, transvec];
          AppendTo[localtopology, cell ->
            Map[SetPrecision[# + transvec, 8] &, cellcoords, {2}]]];
        ], {cell, wrappedcells}];
    ];
  ];
] & /@vertcs;
If[localcellunion != {},
  AppendTo[localtopology,
    Thread[localcellunion ->
      Map[Lookup[indToPtsAssoc, #] &, cellVertexGrouping /@ localcellunion, {2}]]
  ];
];

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[{ylim1 = yLim[[1]], ylim2 = yLim[[2]], xlim2 = xLim[[2]]},
    Block[{boundaryCells, bcells, temp, res},
      temp = <|MapIndexed[First[#2] → #1 &, faceListCoords]|>;
      boundaryCells = Union[First/@Position[temp,
        {x_ /; x ≥ xlim2, __} | {x_ /; x < 0, __} |
        {_, 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 < 0,
              __} | {_, 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",
  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"
]

```

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
]

```

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}
]

```

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}
]

```

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}
]

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}
]

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
];
```

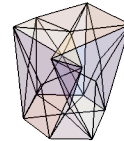
# polyhedron collision avoidance

## removal of self-intersections

```
In[ ]:= Clear@selfIntersectMod;
```

```
In[ ]:= selfIntersectMod::Information =
```

"prevents such cases in which a polyhedron can self-intersect



;

We essentially find degenerate polyhedrons and  
move them back to their initial morphology";

```
In[ ]:= selfIntersectMod[iToPAssoc_, iToPAssocOrig_, CVG_] :=
Block[{polyhed, $iToPAssoc = iToPAssoc, keysdegen, repRules, ptoIAssoc},
polyhed = (Polyhedron@Flatten[triangulateToMesh[#, 1] & /@
Map[Lookup[$iToPAssoc, #] &, CVG, {2}]]);
keysdegen = Keys@Select[(RegionQ /@ polyhed), # == False &];
If[keysdegen != {},
repRules = Flatten[
Thread[# -> Lookup[iToPAssocOrig, #]] &@Union@Flatten@Lookup[CVG, keysdegen]];
AppendTo[$iToPAssoc, repRules];
];
ptoIAssoc = AssociationMap[Reverse, $iToPAssoc];
{$iToPAssoc, ptoIAssoc}
];
```

## removal of cross-intersections

In[6] :=

```

pointcloudFn[bpts_, bcells_, itoPAssoc_, CVG_, thresh_ : 0.2] :=
Block[{ $bpts, firstInnerptsInds, firstInnerpts, firstInnerptsTab},
  $bpts = Sort@Flatten@bpts;
  firstInnerptsInds =
    Union@Flatten[Complement[Union@Flatten[CVG[#]], $bpts] & /@ bcells];
  firstInnerpts = Map[Lookup[itoPAssoc, #] &, firstInnerptsInds];
  firstInnerptsTab = With[{t = thresh},
    With[{ylim = yLim[[2]], xlim = xLim[[2]], ymax = yLim[[2]] (1 + t),
      ymin = -yLim[[2]] t, xmax = xLim[[2]] (1 + t), xmin = -xLim[[2]] t},
      Flatten[Map[Distribute[{
        If[{#[[2]] + ylim} < ymax, SetPrecision[# + {0., ylim, 0.}, 8], Nothing],
        If[{#[[2]] - ylim} > ymin, SetPrecision[# - {0., ylim, 0.}, 8], Nothing],
        If[{#[[1]] + xlim} < xmax, SetPrecision[# + {xlim, 0., 0.}, 8], Nothing],
        If[{#[[1]] - xlim} > xmin, SetPrecision[# - {xlim, 0., 0.}, 8], Nothing],
        If[{#[[1]] - xlim} > xmin && {#[[2]] - ylim} > ymin,
          SetPrecision[# - {xlim, ylim, 0.}, 8], Nothing],
        If[{#[[1]] + xlim} < xmax && {#[[2]] + ylim} < ymax,
          SetPrecision[# + {xlim, ylim, 0.}, 8], Nothing],
        If[{#[[1]] + xlim} < xmax && {#[[2]] - ylim} > ymin,
          SetPrecision[# + {xlim, -ylim, 0.}, 8], Nothing],
        If[{#[[1]] - xlim} > xmin && {#[[2]] + ylim} < ymax,
          SetPrecision[# + {-xlim, ylim, 0.}, 8], Nothing]
      } → {#}, List, Rule] &, firstInnerpts],
      1]
    ] // Association;
  firstInnerptsTab
];

```

```
ClearAll[crossIntersectMod];
```

```
crossIntersectMod::Information =
```

"prevents cases in which a vertex of one polyhedron passes  
into a neighbouring polyhedron;

the vertices that cross over into neighbouring polyhedron are  
forced back to their initial positions";

```

crossIntersectMod[table_, itoPAssoc_, ptoIAssoc_, itoPAssocOrig_, CVG_] :=
Block[{
  $iToPAssoc = itoPAssoc, ptscloud, regPolyhed,
  regCuboid, ptsinCuboid, signedDistPolyFn, ptsinPoly,
  repRules, IndsToReplace, $ptoIAssoc, picks, repPicks},
ptscloud = (Keys[table] ~Join~ Values[$iToPAssoc]);
regPolyhed = (BoundaryMeshRegion@Polyhedron@Flatten[triangulateToMesh[#, 1] & /@
  Map[Lookup[$iToPAssoc, #] &, CVG, {2}]] // Values;
regCuboid = ParallelMap[RegionMember[Cuboid@@Transpose@RegionBounds@#] &,
  regPolyhed];
picks =
  Flatten[
    MapThread[
      (signedDistPolyFn = SignedRegionDistance[#2];
       ptsinCuboid = Pick[ptscloud, #1[ptscloud], True];
       ptsinPoly = Pick[ptsinCuboid, Sign[signedDistPolyFn@ptsinCuboid], -1]
      ) &, {regCuboid, regPolyhed}] /. {} -> Nothing
    , 1];
repPicks = If[picks != {},
  Replace[Lookup[table, picks], Missing["KeyAbsent", x_] -> x, {1}], {}];
If[repPicks != {},
  IndsToReplace = Lookup[ptoIAssoc, repPicks];
  repRules = Thread[IndsToReplace -> Lookup[itoPAssocOrig, IndsToReplace]];
  AppendTo[$iToPAssoc, repRules];
];
$ptoIAssoc = AssociationMap[Reverse, $iToPAssoc];
{$iToPAssoc, $ptoIAssoc}
];

```

## 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@surfaceGrad;
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, target, facept, cross,
      openS = {0., 0., 0.}, closedS = {0., 0., 0.}},
      Do[
        ptTri = opentr[[i]];
        normal = normal0[[i]];
        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]
        ];
        openS += (0.5 * cross), {i, Length@normal0}];
      Do[
        ptTri = closedtr[[j]];
        normal = normalC[[j]];
        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]
        ];
        closedS += (0.5 * cross), {j, Length@normalC}];
      epcc * closedS + epco * openS
    ], CompilationTarget -> "C",
    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[#] × 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 ≠ {}, MapAt[(1 + Vgrowth time) # &, vol, growindkeys], vol];
    kcv Total[((polyhedVols / vol) - 1) 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, normO, 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]};
  {normO, normC} = {Lookup[normalAssoc, opentri], Lookup[normalAssoc, closetri]};
  sgterm = surfaceGrad[pt, opentri, normO, closetri, normC];
  vgterm = volumeGrad[normalAssoc, keys, assocTri, polyvol, growingcellIndices];
  sgterm + vgterm
]

```



# pair boundary points for computing $\nabla$

```

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},
    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,
        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  $\mapsto$  (
      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, 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  $\mapsto$  <|GroupBy[Flatten[#, 1],
      MemberQ[indToPtsAssoc[vert]]][True] & /@ localtrimesh[vert] |>) /@ vertKeys];
  trimesh = triangulateToMesh /@ $faceListAssoc;
  If[faceorientedQ,
    $polyhedVol = volumePolyhedra /@ trimesh;
    polyhedcent = <|
      KeyValueMap[#1  $\rightarrow$  centroidPolyhedra[#2, $polyhedVol[#1]] &, trimesh] |>,
    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]]];
$vertTriNormpair = <|
  # → <|Thread[Flatten[Values@$assoctri[#], 1] → Flatten[signednormals@#, 1]]|> & /@
  vertKeys|>;
opencloseTri = Flatten[Values@#, 1] & /@$assoctri;
$triDistAssoc = (trianglemembers = #;
  GroupBy[GatherBy[trianglemembers, Intersection], Length, Flatten[#, 1] &]) & /@
  opencloseTri;
{$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,
    $indToPtsAssocOrig = indToPtsAssoc, btpairs, vals},
    (*computed once at the start of the computation*)
    keyslocaltopo = Keys@*First /@ topo;
    shiftVecAssoc = Association /@
      Map[Apply[Rule], Thread /@ Select[(#[[2 ;; 3]]) & /@ topo, # ≠ {}], {2}];
    btpairs = boundaryPtsPairing[$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, btpairs];

```

```

(*displace vertices by the numerical gradient*)
$indToPtsAssoc = <|
  KeyValueMap[#1 → SetPrecision[#2 + 0.25 grad1[#1]  $\delta t$ , 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[#])  $\delta t$ , 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[#])  $\delta t$ , 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[#])  $\delta t$ , 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 $[\Delta \leftrightarrow I] \wedge [I \leftrightarrow \Delta]$

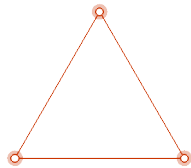
checks to determine whether  $\alpha, \beta, \gamma$  or an invalid pattern is present in the graph

```

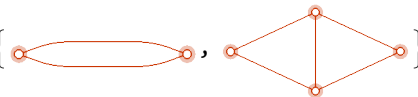
In[ ]:=
$invalidPatternsEdge = Graph[{1 ↔ 2, 2 ↔ 3, 3 ↔ 1}];
Echo[Graph[$invalidPatternsEdge, PlotTheme → "Web", ImageSize → Tiny],
    Style["invalid pattern (edge in Δ): ", Black]];
$invalidPatterns = {
    Graph[{1 ↔ 2, 2 ↔ 1}],
    Graph[{1 ↔ 2, 2 ↔ 3, 3 ↔ 1, 1 ↔ 4, 3 ↔ 4}]
};
Echo[Graph[#, PlotTheme → "Web", ImageSize → Tiny] & /@ $invalidPatterns,
    Style["invalid pattern (double edge || double Δ): ", Black]];

```

» invalid pattern (edge in  $\Delta$ ):



» invalid pattern (double edge || double  $\Delta$ ): {



```

In[ ]:=
edgeinTrianglePatternQ[graph_] := IGSubisomorphicQ[$invalidPatternsEdge, graph];
InvalidEdgePatternQ[graph_] := AnyTrue[$invalidPatterns, IGSubisomorphicQ[#, graph] &];
InvalidTrigonalPatternQ[graph_] :=
    AnyTrue[$invalidPatterns, IGSubisomorphicQ[#, graph] &];

```

```
In[ ]:=
faceIntersections[polyhed_] := AnyTrue[
  Length /@ (Intersection@@@ Replace[Subsets[Partition[#, 2, 1, 1] & /@ polyhed, {2}],
    List → orderlessHead, {4}, Heads → True]), # ≥ 2 &];

gammaPatternFreeQ[polyhedList_] := Not[Or@@ (faceIntersections /@ polyhedList)];
```

## I → Δ operator

```
In[ ]:=
ItoΔpreprocess1::description =
  "the F[x] extracts the vertex pairings from the local topology i.e. {r10,r11}
    and {r1,r4} & {r2,r5} & {r3,r6} → (points attached to r10,r11)";
ItoΔpreprocess1[candidate_, currentTopology_, localTopology_] :=
  Block[{r10, r11, ptsPartitioned, vertAttached,
    cellsPartOf, cellsElim, ptsAttached},
    {r10, r11} = candidate; (* edge unpacked into vertices: r10,r11 *)
    (* r10 → {vertices attached with r10}, r11 → {vertices attached with r11} *)
    ptsPartitioned = If[Keys[#,
      r10 → Flatten[Last@#, 1], r11 → Flatten[Last@#, 1]] & /@ (
      Normal@KeySortBy[
        GroupBy[
          (currentTopology /. {OrderlessPatternSequence[r11, r10]} → Sequence[]),
          MemberQ[#, r10] &], MatchQ[False]] /. {r10 | r11 → Sequence[]}]);
    (* the code below creates pairings between vertices
      such that r1 is packed with r4, r2 with r5 & r3 with r6 *)
    vertAttached = Flatten[Values@ptsPartitioned, 1];
    cellsPartOf =
      Union[Position[localTopology, #, {3}] /. {Key[x_], __} => x] & /@ vertAttached;
    cellsElim = Complement[Union@Flatten[cellsPartOf],
      Union@Flatten@#[[1]] ∩ Union@Flatten@#[[2]]] &@TakeDrop[cellsPartOf, 3];
    If[cellsElim ≠ {},
      cellsPartOf = cellsPartOf /. Alternatives@@ cellsElim → Sequence[]
    ];
    ptsAttached = Values@GroupBy[Thread[vertAttached → cellsPartOf], Last → First];
    {r10, r11, ptsAttached}
  ];
```

In[ ]:=

```

ItoΔpreprocess2::description =
  "the F[x] computes {r7,r8,r9} vertices from the vertices
    attached with r10 & r11 i.e. (r1-r6)";
ItoΔpreprocess2[ptsAttached_, {r10_, r11_}] :=
  Block[{r01, u1T, r1, r4, r2, r5, r3, r6, w07, w08, w09,
    v07, v08, v09, lmax, r7, r8, r9},
    r01 = Mean[{r10, r11}];
    u1T = (r10 - r11) / Norm[r10 - r11];
    {{r1, r4}, {r2, r5}, {r3, r6}} = ptsAttached;
    w07 = 0.5 ((r1 - r01) / Norm[r1 - r01] + (r4 - r01) / Norm[r4 - r01]);
    w08 = 0.5 ((r2 - r01) / Norm[r2 - r01] + (r5 - r01) / Norm[r5 - r01]);
    w09 = 0.5 ((r3 - r01) / Norm[r3 - r01] + (r6 - r01) / Norm[r6 - r01]);
    v07 = w07 - (w07.u1T) u1T;
    v08 = w08 - (w08.u1T) u1T;
    v09 = w09 - (w09.u1T) u1T;
    lmax = Max[Norm[v08 - v07], Norm[v09 - v08], Norm[v07 - v09]];
    r7 = SetPrecision[r01 + (δ / lmax) v07, 8];
    r8 = SetPrecision[r01 + (δ / lmax) v08, 8];
    r9 = SetPrecision[r01 + (δ / lmax) v09, 8];
    {r1, r2, r3, r4, r5, r6, r7, r8, r9}
  ];

```

In[ ]:=

```

insertTrigonalFace::description = "the module inserts the trigonal face into the cell";
insertTrigonalFace[topology_, r7_, r8_, r9_, r10_, r11_] := Block[{posInserts},
  posInserts = Position[
    FreeQ[#, {___, OrderlessPatternSequence[r10, r11], ___}] & /@ topology, True];
  If[posInserts ≠ {},
    Insert[topology, {r7, r8, r9}, Flatten[{#, -1}] & /@ posInserts],
    topology]
];

```

In[ ]:=

```

Clear@corrTriOrientationHelper;
corrTriOrientationHelper[topology_, trigonalface_] :=
  Block[{allTri, selectTriAttached, selectTriSharedEdge, selectTri,
    partTri, partAttachedTri},
    partTri = Partition[trigonalface, 2, 1, 1];
    allTri = Flatten[triangulateToMesh@topology, 1];
    selectTriAttached =
      Cases[allTri, {OrderlessPatternSequence[___, Alternatives @@ trigonalface]}];
    selectTriSharedEdge = Select[selectTriAttached,
      Length[Intersection[#, trigonalface]] == 2 &];
    selectTri = RandomChoice@selectTriSharedEdge;
    partAttachedTri = Partition[selectTri, 2, 1, 1];
    If[Intersection[partAttachedTri, partTri] ≠ {},
      topology /. trigonalface => Reverse@trigonalface,
      topology
    ]
  ];

```

In[ ]:=

```

Clear@corrTriOrientation;
corrTriOrientation::description =
  "the function corrects the orientation of the added trigonal
    face, such that it is oriented c.c.w";
corrTriOrientation[localtopology_, trigonalface_] := Block[{cells, affectedIDs, topo},
  cells = Map[DeleteDuplicates@* (Flatten[#, 1] &), localtopology, {2}];
  affectedIDs = Partition[First /@ Position[cells, trigonalface], 1];
  topo = MapAt[corrTriOrientationHelper[#1, trigonalface] &, cells, affectedIDs];
  Map[Partition[#, 2, 1, 1] &, topo, {2}]
];

```

In[ ]:=

```

ItoDeltaoperation::description =
  "the module removes vertices r10,r11 and connects the points
    r1-r6 with the new points r7-r9";
ItoDeltaoperation[graphnewLocalTopology_, cellCoords_, r1_, r2_, r3_, r4_,
  r5_, r6_, r7_, r8_, r9_, r10_, r11_] := Block[{mat},
  mat = insertTrigonalFace[cellCoords, r7, r8, r9, r10, r11];
  Map[Partition[#, 2, 1, 1] &, mat, {2}] /. {
    {OrderlessPatternSequence[r11, r10]} => Sequence[],
    {PatternSequence[r11, q : r4 | r5 | r6]} =>
      Switch[q, r4, {r7, r4}, r5, {r8, r5}, r6, {r9, r6}],
    {PatternSequence[q : r4 | r5 | r6, r11]} =>
      Switch[q, r4, {r4, r7}, r5, {r5, r8}, r6, {r6, r9}],
    {PatternSequence[r10, q : r1 | r2 | r3]} =>
      Switch[q, r1, {r7, r1}, r2, {r8, r2}, r3, {r9, r3}],
    {PatternSequence[q : r1 | r2 | r3, r10]} =>
      Switch[q, r1, {r1, r7}, r2, {r2, r8}, r3, {r3, r9}]
  ] /; (! InvalidEdgePatternQ[graphnewLocalTopology]);

```

In[ ]:=

```

bindCellsToNewTopology::description =
  "the module pairs modified cell topologies with cell IDs if 'γ' pattern is absent";
bindCellsToNewTopology[adjoiningCells_, network_, func_ : Identity] /;
  gammaPatternFreeQ[network] := Thread[adjoiningCells -> func[network]];

```



In[ ]:=

```

modifier::description = "the module makes
  modifications to the datastructures after topological transitions";
modifier[candidate_, adjoiningCells_, indToPtsAssoc_,
  ptsToIndAssoc_, cellVertexGrouping_,
vertexToCell_, celltopologicalChanges_, updatedLocalNetwork_, newAdditions_] :=
  Block[{dropVertInds, $ptsToIndAssoc = ptsToIndAssoc,
    $indToPtsAssoc = indToPtsAssoc, $cellVertexGrouping = cellVertexGrouping,
    $vertexToCell = vertexToCell},
    dropVertInds = Lookup[$ptsToIndAssoc, candidate];
    KeyDropFrom[$ptsToIndAssoc, candidate];
    KeyDropFrom[$indToPtsAssoc, dropVertInds];
    {AssociateTo[$ptsToIndAssoc, #~Reverse~2], AssociateTo[$indToPtsAssoc, #]} &@
      newAdditions;
    AssociateTo[$cellVertexGrouping, MapAt[$ptsToIndAssoc,
      celltopologicalChanges, {All, 2, All, All}]];
    KeyDropFrom[$vertexToCell, Sort@dropVertInds];
    AssociateTo[$vertexToCell,
      (First[#] → Part[adjoiningCells, Union[
        First/@Position[updatedLocalNetwork, Last@#, {3}]]]) &/@newAdditions];
    {$indToPtsAssoc, $ptsToIndAssoc, $cellVertexGrouping, $vertexToCell}
  ];

```

In[ ]:=

```

ItoΔ[edges_, faceListCoords_, indToPtsAssoc_,
  ptsToIndAssoc_, cellVertexGrouping_, vertexToCell_, wrappedMat_] :=
  Block[{edgelen, edgesel, candidate, graphCurrentTopology, currentTopology, z, ž,
    localTopology = {}, adjoiningcells, cellCoords, r10, r11, ptsAttached, r1, r2, r3,
    r4, r5, r6, r7, r8, r9, newLocalTopology, graphnewLocalTopology, modifiednetwork,
    cellTopologicalChanges, maxVnum, wrappedcells, celltransvecAssoc, newAdditions,
    transvec, ls, vpt, cellTopologicalChangesBeforeShift, positions, cellspartof,
    vertices, $indToPtsAssoc = indToPtsAssoc, $ptsToIndAssoc = ptsToIndAssoc,
    $cellVertexGrouping = cellVertexGrouping, $vertexToCell = vertexToCell,
    $edges = edges, $wrappedMat = wrappedMat, $faceListCoords = faceListCoords},

    edgelen = lenEdge@@@$edges; (*here we check the length of all the edges,
    we can also use built-in EuclideanDistance[]*)
    edgesel = Pick[$edges, 1 - UnitStep[edgelen - δ], 1];
    (*select edges that have length less than critical value δ*)
    Scan[
      (candidate = #; (*candidate edge*)
        vertices = DeleteDuplicates@Flatten[$edges, 1];
        If[AllTrue[candidate, MemberQ[vertices, #] &],
          (*this means that the edge exists in the network.
            If there are two adjacent edges
            that need to be transformed and one gets transformed first
            then the second one will not exist*)
          (* get all edges that are connected to our edge of interest *)
          currentTopology = Cases[$edges,
            {OrderlessPatternSequence[x_, p : Alternatives@@candidate]} → {p, x}];
          (* this part of code takes care of border cells *)

```

```

If[Length[currentTopology] < 7,
  (*Print[" # of edges is < than 7 "];*)
  (* here we get the local topology of our network *)
  {localTopology, wrappedCells, transvec} =
    getLocalTopology[$ptsToIndAssoc, $indToPtsAssoc, $vertexToCell,
      $cellVertexGrouping, $wrappedMat, $faceListCoords][candidate];
  (*Print[Keys@localTopology];*)
  (* this yields all the unique edges
    in the localTopology and extract vertex pairs, such that
    {candidate_vertex, vertex attached to candidate} *)
  With[{edg = DeleteDuplicatesBy[
    Flatten[Map[Partition[#, 2, 1, 1] &, Values@localTopology, {2}], 2], Sort]},
    currentTopology = Cases[edg,
      {OrderlessPatternSequence[x_, p : Alternatives@@ candidate]} -> {p, x}];
  ];
];
(*creating a graph from the current topology*)
graphCurrentTopology =
  Graph@Replace[currentTopology, List -> UndirectedEdge, {2}, Heads -> True];
If[edgeinTrianglePatternQ@graphCurrentTopology,
  (*edge is part of a trigonal face and
    hence nothing is to be done. this prevents `α` pattern *)
  None,
  {z, ž} = candidate; (* edge vertices unpacked *)
  If[localTopology == {},
    (* here we get the local topology of our network *)
    {localTopology, wrappedCells, transvec} =
      getLocalTopology[$ptsToIndAssoc, $indToPtsAssoc, $vertexToCell,
        $cellVertexGrouping, $wrappedMat, $faceListCoords][candidate];
  ];
  {adjoiningCells, cellCoords} = {Keys@#, Values@#} &@localTopology;
  (* adjoining cells and their vertices *)
  (*Print[adjoiningCells];*)
  (* label vertices joining the candidate edge *)
  {r10, r11, ptsAttached} =
    ItoΔpreprocess1[candidate, currentTopology, localTopology];
  (* getting all vertices for transformation including (r7,r8,r9) *)
  {r1, r2, r3, r4, r5, r6, r7, r8, r9} = ItoΔpreprocess2[ptsAttached, {r10, r11}];
  (*
  (*print old and predicted topology *)
  Print[
    Graphics3D[{PointSize[0.025], Red, Point@{r1, r4, r7},
      Green, Point@{r2, r5, r8}, Blue, Point@{r3, r6, r9}, Purple,
      Point@r10, Pink, Point@r11, Black, Line@currentTopology, Dashed,
      Line[{r1, r7}], Line[{r4, r7}], Line[{r2, r8}], Line[{r5, r8}],
      Line[{r3, r9}], Line[{r6, r9}], Purple, Line@ptsAttached}, ImageSize -> Small]
  ];
  *)
  If[! IGSubisomorphicQ[$invalidPatternsEdge, graphCurrentTopology],
    (* at least no α pattern will be generated. I think
      this has been checked in the If statement prior to this *)

```

```

(* Scheme: apply [I]→[H]; check if the new topology is valid (i.e. no  $\alpha, \beta$ );
check if the new topology is free of  $\gamma$  pattern;
replace network architecture *)
(*forming new topology and graph*)
newLocalTopology = {r1  $\leftrightarrow$  r7, r4  $\leftrightarrow$  r7,
  r2  $\leftrightarrow$  r8, r5  $\leftrightarrow$  r8, r3  $\leftrightarrow$  r9, r6  $\leftrightarrow$  r9, r7  $\leftrightarrow$  r8, r8  $\leftrightarrow$  r9, r9  $\leftrightarrow$  r7};
graphnewLocalTopology = Graph@newLocalTopology;
(* apply Ito $\Delta$  operation *)
modifiednetwork = Ito $\Delta$ operation[graphnewLocalTopology,
  cellCoords, r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11];
modifiednetwork = corrTriOrientation[modifiednetwork, {r7, r8, r9}];
(*bind cells with their new topology if  $\gamma$  pattern is absent*)
cellTopologicalChanges = bindCellsToNewTopology[adjoiningcells,
  modifiednetwork, Map[Map[DeleteDuplicates@Flatten[#, 1] &]]];
(*
(*print topology post operation *)
If[(cellTopologicalChanges != {})] ||
  (Head[cellTopologicalChanges] != bindCellsToNewTopology),
  Print[ind→Graphics3D[{{Opacity[0.1], Blue, Polyhedron/@
    Values[cellTopologicalChanges]},
    {Red, Line@candidate}}, Axes→True]]
];
*)
If[(cellTopologicalChanges != {})] ||
  (Head[cellTopologicalChanges] != bindCellsToNewTopology),
  (*if you are here then it means that cell topology was altered *)
  modifiednetwork = Values@cellTopologicalChanges;
  (*vertex coordinates of the modified topology*)
  maxVnum = Max[Keys@$indToPtsAssoc]; (*maximum
  number of vertices so far*)
  If[wrappedcells != {},
    (* if there are wrapped
    cells send them back to their respective positions *)
    (* wrapped cells with their respective vectors for translation *)
    celltransvecAssoc = AssociationThread[wrappedcells, transvec];
    cellTopologicalChangesBeforeShift = cellTopologicalChanges;
    (* here we send the cells
    back to their original positions → unwrapped state *)
    cellTopologicalChanges = (x  $\mapsto$  With[{p = First[x]},
      If[MemberQ[wrappedcells, p],
        p  $\rightarrow$  Map[SetPrecision[# - celltransvecAssoc[p], 8] &, Last[x], {2}], x]
    ]) /@ cellTopologicalChanges;

  ls = {};
  Scan[
    vpt  $\mapsto$ 
    (positions = Position[cellTopologicalChangesBeforeShift, vpt];
    positions = DeleteDuplicates[{First[#]} & /@ positions];
    cellspartof = Extract[adjoiningcells, positions];
    Fold[

```

```

Which[MemberQ[wrappedcells, #2],
      AppendTo[ls, SetPrecision[vpt - celltransvecAssoc[#2], 8] ],
      True, If[! MemberQ[ls, vpt], AppendTo[ls, vpt]]] &, ls, cellspartof]),
{r7, r8, r9}];

newAdditions = Thread[(Range[Length@ls] + maxVnum) → ls],
newAdditions = Thread[(Range[3] + maxVnum) → {r7, r8, r9}]
(* labels for new vertices *)
];

(* appropriate changes are made to the datastruct *)
{$indToPtsAssoc, $ptsToIndAssoc, $cellVertexGrouping, $vertexToCell} =
modifier[candidate, adjoiningcells, $indToPtsAssoc,
          $ptsToIndAssoc, $cellVertexGrouping,
          $vertexToCell, cellTopologicalChanges, modifiednetwork, newAdditions];

$faceListCoords = Map[Lookup[$indToPtsAssoc, #] &, $cellVertexGrouping, {2}];

$edges =
Flatten[Map[Partition[#, 2, 1, 1] &, Map[Lookup[$indToPtsAssoc, #] &, Values[
          $cellVertexGrouping], {2}], {2}], 2] // DeleteDuplicatesBy[Sort];

$wrappedMat = With[{temp = Keys[$cellVertexGrouping]},
AssociationThread[temp → Map[Lookup[$indToPtsAssoc, #] /. periodicRules &,
Lookup[$cellVertexGrouping, temp], {2}]]
];
];
];
];
]) &, edgesel];
{$edges, $indToPtsAssoc,
 $ptsToIndAssoc, $cellVertexGrouping, $vertexToCell, $wrappedMat}
];

```

## $\Delta \rightarrow I$ operator

*In[ ]:=*

```

pickTriangulatedFaces::description = "pick candidate  $\Delta$  faces to transform";
pickTriangulatedFaces[faceListCoords_] :=
Block[{triangleCandidates, triangleCandidatesSel},
  triangleCandidates = Cases[faceListCoords, x_ /; Length[x] == 3, {2}];
  (* yield all  $\Delta$  faces from the mesh & retain
   those that pass Satoru's 2nd condition *) triangleCandidatesSel =
    AllTrue[lenEdge @@@ Partition[#, 2, 1, 1], # ≤  $\delta$  &] & /@ triangleCandidates;
  Pick[triangleCandidates, triangleCandidatesSel, True]
];

```

```

In[ ]:=
ΔtoIoperation[network_, rules_] := Block[{ruleapply},
  ruleapply =
    ((network /. rules) /. Line[] → Sequence[]) /. {Line → Sequence, {} → Sequence[]};
  Map[DeleteDuplicates@Flatten[#, 1] &, ruleapply, {2}]
];

```

```

In[ ]:=
rulesΔtoI[currentTopology_, ptsTri_, ptPartition_] :=
  Block[{attachedEdges, triedges, reconnectRules, rules},
    (*edges connected with face*)
    attachedEdges = DeleteCases[currentTopology,
      Alternatives@@({OrderlessPatternSequence@@#} & /@ Partition[ptsTri, 2, 1, 1])];
    triedges = Complement[currentTopology, attachedEdges];
    (* only edges that form the trigonal face *)
    reconnectRules = Flatten[Cases[attachedEdges,
      q : {y_, p : Alternatives@@Last[#]} => q → {First@#, p}] & /@ ptPartition];
    rules = Dispatch[reconnectRules~Join~Reverse[reconnectRules, {3}]~Join~(
      {OrderlessPatternSequence@@#} → Sequence[] & /@ triedges)];
    rules
  ] /; (!InvalidTrigonalPatternQ[
    Graph@Replace[currentTopology, List → UndirectedEdge, {2}, Heads → True]]);

```

```

In[ ]:=
ΔtoIpreprocess[ptsTri_, currentTopology_] :=
  Block[{sortptsTri, uTH, r1, r2, PtsAcrossFaces,
    ptsAttached, newPts, ptPartition, newLocalTopology, vec},
    With[{r0H = Mean@ptsTri},
      sortptsTri = SortBy[ptsTri, ArcTan[# - r0H] &];
      (* arrange the points in an clockwise || anti-clockwise manner *) uTH = Function[
        Cross[#2 - #1, #3 - #1] / (Norm[#2 - #1] Norm[#3 - #1])] [Sequence@@ sortptsTri];
      r1 = SetPrecision[r0H + 0.5 δ * uTH, 8];
      r2 = SetPrecision[r0H - 0.5 δ * uTH, 8];
      vec = Normalize[r1 - r2];
      ptsAttached = DeleteCases[currentTopology~Flatten~1, Alternatives@@ptsTri];
      (* are points above or below the Δ *)
      PtsAcrossFaces = GroupBy[ptsAttached, Sign[vec.(r0H - #)] &];
      (* compute the 2 new pts from the trio of old pts *)
      newPts = <|Sign[vec.(r0H - #)] → # & /@ {r1, r2}|>;
      ptPartition = Values@Merge[{newPts, PtsAcrossFaces}, Identity];
      (* which points belong with r1 and which points with r2 *) newLocalTopology =
        Flatten[Map[x ↦ First[x] ↦ # & /@ Last[x], ptPartition], 1]~Join~{r1 → r2};
      {ptPartition, newLocalTopology, r1, r2}]
  ];

```

```

In[ ]:=
ΔtoI[edges_, faceListCoords_, indToPtsAssoc_,
  ptsToIndAssoc_, cellVertexGrouping_, vertexToCell_, wrappedMat_] :=
  Block[{selectTriangle, candidate, currentTopology, ptsAttached, graphCurrentTopology,
    ptsTri, PtPartition, newLocalTopology, adjoiningCells, prevNetwork,
    updatedLocalNetwork, rules, celltopologicalChanges, r1, r2, maxVnum, newAdditions,

```

```

localTopology, wrappedCells, transvec, cellCoords, ls, positions, cellTransvecAssoc,
cellTopologicalChangesBeforeShift, cellspartof, $faceListCoords = faceListCoords,
$ptsToIndAssoc = ptsToIndAssoc, $indToPtsAssoc = indToPtsAssoc,
$vertexToCell = vertexToCell, $cellVertexGrouping = cellVertexGrouping,
$wrappedMat = wrappedMat, vpt, $edges = edges, selectTriangles},
selectTriangles = pickTriangulatedFaces@*Values@$faceListCoords;
Scan[
(candidate = #;
If[And@@(KeyMemberQ[$ptsToIndAssoc, #] & /@ candidate),
(* get local network topology from the  $\Delta$ 
face: basically which coordinates the face is linked to *)
{localTopology, wrappedCells, transvec} =
getLocalTopology[$ptsToIndAssoc, $indToPtsAssoc, $vertexToCell,
$cellVertexGrouping, $wrappedMat, $faceListCoords][candidate];
{adjoiningCells, cellCoords} = {Keys@#, Values@#} &@localTopology;
(* adjoining cells and their vertices *)
prevNetwork = Map[Partition[#, 2, 1, 1] &, cellCoords, {2}];
(* this yields all the unique edges
in the current topology and extract vertex pairs, such that
{candidate_vertex, vertex_attached with candidate} *)
currentTopology = Cases[DeleteDuplicatesBy[Flatten[prevNetwork, 2], Sort],
{OrderlessPatternSequence[x_, p : Alternatives@@ candidate]}  $\Rightarrow$  {p, x}];
(*creating a graph from the current topology*)
graphCurrentTopology =
Graph@Replace[currentTopology, List  $\rightarrow$  UndirectedEdge, {2}, Heads  $\rightarrow$  True];
If[! InvalidTrigonalPatternQ[graphCurrentTopology],
(* transform the network topology by applying [H] $\rightarrow$ [I] operation *)
ptsTri = candidate; (* vertices of the faces *)
{PtPartition, newLocalTopology, r1, r2} =
 $\Delta$ toIpreprocess[ptsTri, currentTopology];
(*Print@Graphics3D[{Dashed,Thick,Opacity[0.6],Black,Line@currentTopology},
{Thick,Opacity[0.6],Darker@Blue,
Line[newLocalTopology/.UndirectedEdge $\rightarrow$  List]},
{Red,PointSize[0.035],Point@PtPartition[[2,-1]]},
{Blue,PointSize[0.035],Point@PtPartition[[1,-1]]},
{Orange,PointSize[0.05],Point@candidate},{Darker@Green,
PointSize[0.05],Point@{r1,r2}}},ImageSize $\rightarrow$ Small];*)
rules = rules $\Delta$ toI[currentTopology, ptsTri, PtPartition];
Switch[
rules, _rules $\Delta$ toI, None,
_,
(updatedLocalNetwork =  $\Delta$ toIoperation[prevNetwork, rules];
celltopologicalChanges = bindCellsToNewTopology[adjoiningCells,
updatedLocalNetwork] /. _bindCellsToNewTopology  $\rightarrow$  {}];
If[celltopologicalChanges  $\neq$  {},
(*Print[Graphics3D[
{{Opacity[0.1],Blue,Polyhedron/@Values[celltopologicalChanges]},
{Red,Line[candidate~Append~First[candidate]]}},Axes $\rightarrow$ True]]];*)

maxVnum = Max[Keys@$indToPtsAssoc];

```

```

If[wrappedcells ≠ {},
  (* if there are wrapped
    cells send them back to their respective positions *)
  (* wrapped cells with their respective vectors for translation *)
  celltransvecAssoc = AssociationThread[wrappedcells, transvec];
  cellTopologicalChangesBeforeShift = celltopologicalChanges;
  (* here we send the cells
    back to their original positions → unwrapped state *)
  celltopologicalChanges = (x ↦ With[{p = First[x]},
    If[MemberQ[wrappedcells, p], p →
      Map[SetPrecision[# - celltransvecAssoc[p], 8] &, Last[x], {2}], x]
    ]) /@ celltopologicalChanges;

  ls = {};
  Scan[
    vpt ↦
    (positions = Position[cellTopologicalChangesBeforeShift, vpt];
     positions = DeleteDuplicates[{First[#]} & /@ positions];
     cellspartof = Extract[adjoiningCells, positions];
     Fold[
       Which[MemberQ[wrappedcells, #2],
         AppendTo[ls, SetPrecision[vpt - celltransvecAssoc[#2], 8] ],
         True, If[! MemberQ[ls, vpt], AppendTo[ls, vpt]]] &,
       ls, cellspartof]), {r1, r2}];

  newAdditions = Thread[(Range[Length@ls] + maxVnum) → ls],
  newAdditions = Thread[(Range[2] + maxVnum) → {r1, r2}]
  (* labels for new vertices *)
];
updatedLocalNetwork =
  Map[Partition[#, 2, 1, 1] &, Values[celltopologicalChanges], {2}];

{$indToPtsAssoc, $ptsToIndAssoc, $cellVertexGrouping, $vertexToCell} =
  modifier[candidate, adjoiningCells, $indToPtsAssoc,
    $ptsToIndAssoc, $cellVertexGrouping, $vertexToCell,
    celltopologicalChanges, updatedLocalNetwork, newAdditions];

$faceListCoords =
  Map[Lookup[$indToPtsAssoc, #] &, $cellVertexGrouping, {2}];

$edges = Flatten[Map[Partition[#, 2, 1, 1] &, Map[
  Lookup[$indToPtsAssoc, #] &, Values[$cellVertexGrouping],
  {2}], {2}], 2] // DeleteDuplicatesBy[Sort];

With[{temp = Keys[$cellVertexGrouping]},
  $wrappedMat = AssociationThread[
    temp → Map[Lookup[$indToPtsAssoc, #] /. periodicRules &,
      Lookup[$cellVertexGrouping, temp], {2}]
  ];
])

```

```

    ]
  ]
  ]) &, selectTriangle];
{$edges, $indToPtsAssoc,
 $ptsToIndAssoc, $cellVertexGrouping, $vertexToCell, $wrappedMat}
];

```

## 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", 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[ ]:= visualizeMesh[table_, CVG_] :=
  (Polyhedron@Flatten[triangulateToMesh[#, 1] & /@ Map[Lookup[table, #] &, CVG, {2}]] //
  Graphics3D[#, ImageSize → 750] &@*Values;

```

## main f(x)

1. instantiate geometry
2. seed growing cells in the tissue



3. initiate counter, and time-step =  $\delta t$
4. main loop (? termination condition is fulfilled):
  - counter += 1
  - if Mod[counter, 10] is 0 :
    - check for ( $I \rightarrow \Delta$ ) & ( $\Delta \rightarrow I$ ) transitions
  - perform RK5 mesh integration
  - time +=  $\delta t$
5. visualize mesh

## test module [single time-step]

### run vertex model

In[ ]:= time = 21 / 1000.;

```
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},
Echo[(*time=counter*delta t*)time, "time: "];
While[counter < 4,
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[" ----- checking topological operations ----- "];
$edges = Flatten[Map[Partition[#, 2, 1, 1] &, Map[Lookup[$indToPtsAssoc, #] &,
Values[$cellVertexGroup], {2}], {2}], 2] // DeleteDuplicatesBy[Sort];
(*I->Delta*)
{$edges, $indToPtsAssoc, $ptsToIndAssoc, $cellVertexGroup,
$vertexToCell, $wrappedMat} = ItoDelta[$edges, $faceListCoords, $indToPtsAssoc,
$ptsToIndAssoc, $cellVertexGroup, $vertexToCell, $wrappedMat];
$faceListCoords = Map[Lookup[$indToPtsAssoc, #] &, $cellVertexGroup, {2}];
(*Delta->I*)
{$edges, $indToPtsAssoc, $ptsToIndAssoc, $cellVertexGroup,
$vertexToCell, $wrappedMat} = DeltaToI[$edges, $faceListCoords, $indToPtsAssoc,
$ptsToIndAssoc, $cellVertexGroup, $vertexToCell, $wrappedMat];
$faceListCoords = Map[Lookup[$indToPtsAssoc, #] &, $cellVertexGroup, {2}],
Nothing
];

PrintTemporary[" ----- finding local topology ----- "];
(*we need wrappedMatBound*)
```

```

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[" ----- proceeding with RK5 integration ----- "];
(*make sure we have topo to proceed with this*)
$indToPtsAssoc = RK5Integrator[$indToPtsAssoc,
    topo, $cellVertexGroup, $vertexToCell, $ptsToIndAssoc];
(*here we compute ptstoind *)
$ptsToIndAssoc = AssociationMap[Reverse, $indToPtsAssoc];
(*avoiding polyhedral collisions: self and cross *)
{$indToPtsAssoc, $ptsToIndAssoc} =
    selfIntersectMod[$indToPtsAssoc, $indToPtsAssocOrig, $cellVertexGroup];
bptpairs = boundaryPtsPairing[$vertexToCell, $indToPtsAssoc, $ptsToIndAssoc];
outerptscloudTab =
    pointcloudFn[bptpairs, boundarycells, $indToPtsAssoc, $cellVertexGroup];
{$indToPtsAssoc, $ptsToIndAssoc} = crossIntersectMod[outerptscloudTab,
    $indToPtsAssoc, $ptsToIndAssoc, $indToPtsAssocOrig, $cellVertexGroup];

(*DumpSave["C:\\Users\\aliha\\Desktop\\tempdatastruct\\save"<>ToString[counter]<>
    ".mx",{$indToPtsAssoc,$ptsToIndAssoc,$cellVertexGrouping,$vertexToCell}];*)

(*update time*)
time += δt; counter += 1;
PrintTemporary["< updating 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
```

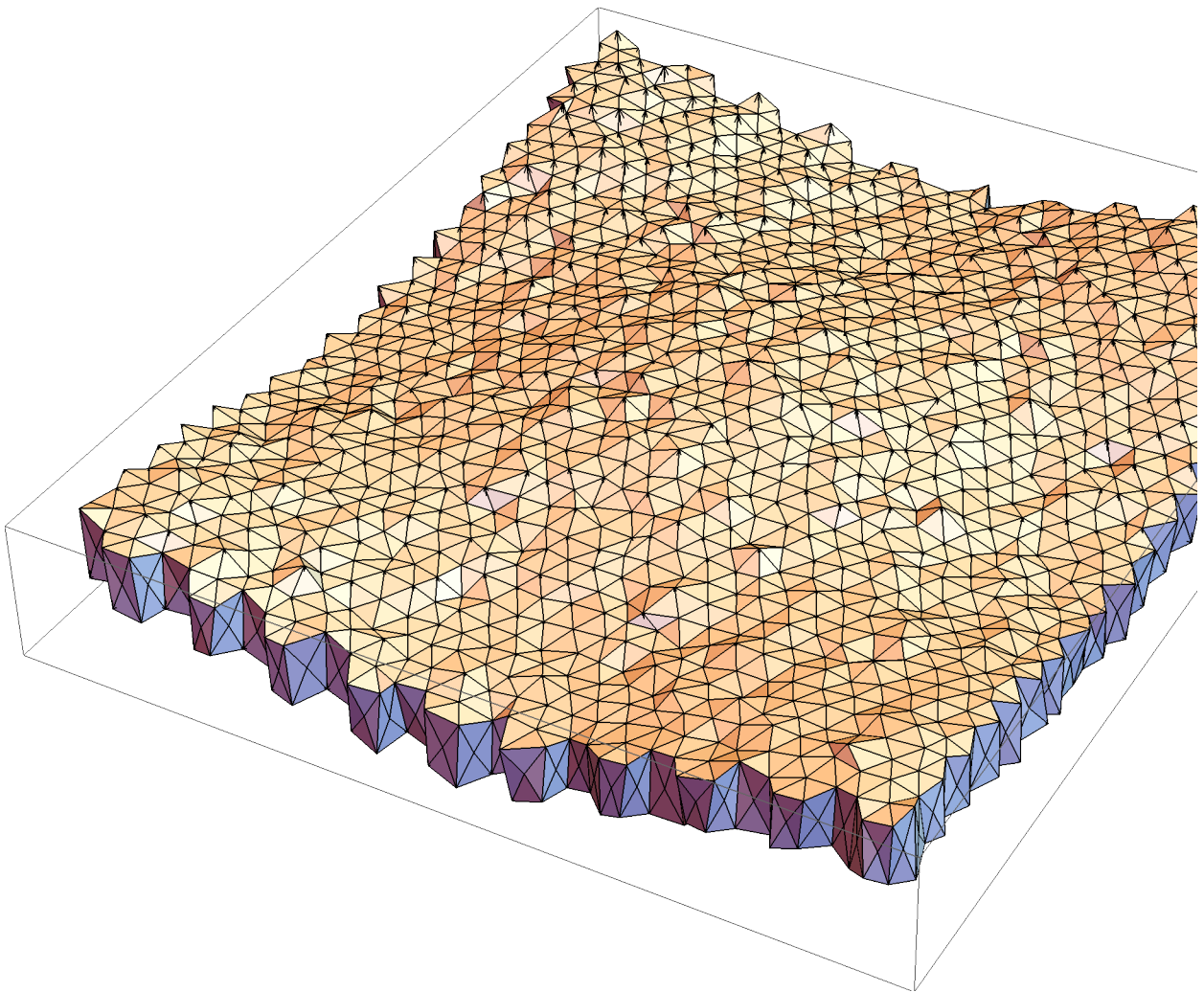
```
» counter: 3
```

```
----- iterations ended successfully -----
```

```
Out[ ]:= {95.2753, Null}
```

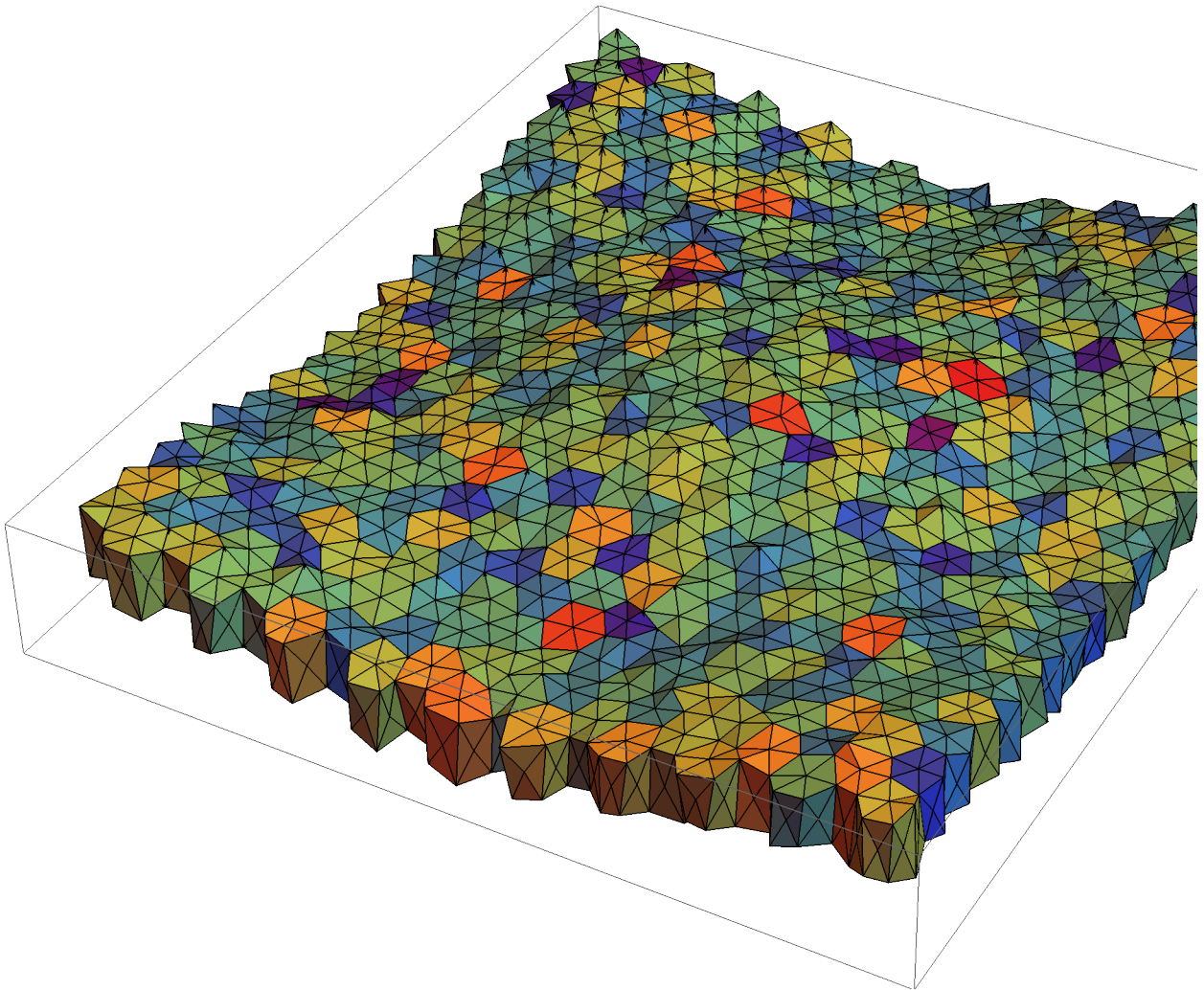
```
In[ ]:= visualizeMesh[res2, res4]
```

Out[ ]:=



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

Out[ ]:=



```

In[ ]:= Grid[
  Partition[
    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[res2, #] & /@ CVG))}],
          ImageSize -> Tiny, Boxed -> False]
        ]
      ] /@ RandomSample[Range[400], 30], 10]
]

```

Out[ ]:=

