

3D Vertex Model :: infinite sheet of cells

Initialization

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

```
Out[*]:= D:\LocalData\hashmial\3D vertex model - github\
```

importing mesh

mesh parameters and specifying precision

```
In[*]:= (*DumpGet["C:\\Users\\aliha\\Desktop\\optimize.mx"];*)
DumpGet["D:\\LocalData\\hashmial\\3D vertex model -
        github\\curved monolayer\\create monolayer - noisy\\noisymesh.mx"];
ptsToIndAssoc = KeyMap[SetPrecision[#, 8] &, ptsToIndAssoc];
indToPtsAssoc = SetPrecision[#, 8] & /@ indToPtsAssoc;
wrappedMat = SetPrecision[#, 8] & /@ wrappedMat;
faceListCoords = SetPrecision[#, 8] & /@ faceListCoords;
yLim = yLim - yLim[[1]];
```

```
In[*]:= Map[MinMax] @* Transpose @* Keys @ ptsToIndAssoc
```

```
Out[*]:= {{-0.10790911, 13.823121}, {-0.45677199, 15.736825}, {-1.6193558, 0.60661207}}
```

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

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

```
In[*]:= Needs["CompiledFunctionTools`"];
```

call dependencies/misc

Launch IGraphM







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

IGraph/M 0.5.1 (October 12, 2020)

```
Out[ ]:= Evaluate IGDocumentation[] to get started.
```

Launch Subkernels for parallel processing

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

```
Out[ ]:= {KernelObject[ Name: Local kernel KernelID: 1], KernelObject[ Name: Local kernel KernelID: 2],
KernelObject[ Name: Local kernel KernelID: 3], KernelObject[ Name: Local kernel KernelID: 4],
KernelObject[ Name: Local kernel KernelID: 5], KernelObject[ Name: Local kernel 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 =  $\delta t$  = 0.005; (*time param*)
celltypes = <| Thread[Range[400] → RandomInteger[{1, 2}, 400]] |>;
(*cell types defined here*)
surfCoeff = <| {1, 1} → 1., {2, 1} → 1., {1, 2} → 1., {2, 2} → 1., {0, 1} → 0.7,
  {0, 2} → 0.7, {-1, 1} → 0.7, {-1, 2} → 0.7 |>; (*params for surface tension*)
kcvAssoc = <| 1 → 14.0, 2 → 14.0 |>; (*volume elasticity and equilibrium volume*)
V0 = 0.528187;
Vgrowth =  $1.0 \times 10^{-3}$ ; (*volume growth-rate*)
 $\delta$  =  $1.0 \times 10^{-3}$ ; (*length threshold for topological transitions*)
growingcellIndices = {...} +;

```

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[]:=

```

Clear@getLocalTopology;
With[{xlim1 = xLim[[1]], xlim2 = xLim[[2]], ylim1 = yLim[[1]], ylim2 = yLim[[2]]},
  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[(D ~ RegionMember ~ Most[vert]),
      (* vertex inside bounds of the simulation domain *)
      (*Print["vertex inside bounds"];*)
      v = vert;
      With[{x = v[[1]], y = v[[2]]},
        boundsCheck = (x == xlim1 || x == xlim2 || y == ylim1 || y == ylim2)];
      extractcellkeys = If[boundsCheck,
        {r11, r12} = {SetPrecision[v, 5], SetPrecision[v /. periodicRules, 5]};
        Block[{x$},
          rules = With[{r = r11, s = r12},
            DeleteDuplicates[HoldPattern[Equal[x$, r]] || HoldPattern[Equal[x$, s]]]
          ]
        ];
        Position @@ With[{rule = rules},
          Hold[wrappedMat, x_ /; ReleaseHold@rule, {3}]
        ],
        With[{p = SetPrecision[v, 5]},
          Position[wrappedMat, x_ /; Equal[x, p], {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, 5];
(* ----- CORE B ----- *)
(* find which cells the
  shifted vertex is a part of in the wrapped matrix *)
wrappedcells = With[{voffbounds = SetPrecision[vertOutOfBounds, 5]},
  Complement[
    Union@Cases[
      Position[wrappedMat,
        x_ /; Equal[x, shiftedPt] || Equal[x, voffbounds], {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,
      Print[pos];
      pos = FirstPosition[SetPrecision[
        cellcoords /. periodicRules, 5], SetPrecision[opt, 5]];
    ];
    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[cellconnected, wrappedcells];
  If[Length[ls] ≠ 3,
    pos = Position[faceListCoords, x_ /; Equal[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";
With[{xlim1 = xLim[[1]], xlim2 = xLim[[2]], ylim1 = yLim[[1]], ylim2 = yLim[[2]]},
  outerCellsFn[faceListCoords_, vertexToCell_, ptsToIndAssoc_] :=
    Block[{res, missing, nearestpt, ptc, p1, p2, p3, boole, picks},
      ptc = Keys[ptsToIndAssoc];
      {p1, p2, p3} = Transpose@ptc;
      boole = Unitize[Boole[Thread[p1 ≤ xlim1]] + Boole[Thread[p1 ≥ xlim2]] +
        Boole[Thread[p2 ≤ ylim1]] + Boole[Thread[p2 ≥ ylim2]]];
      picks = Pick[ptc, boole, 1];
      res = Union@Flatten@Lookup[vertexToCell,
        Lookup[ptsToIndAssoc, picks~Join~(picks /. periodicRules)]
      ];
      If[MemberQ[res, _Missing, {1}],
        missing = Cases[res, Missing[_], Missing[_], y_] => y, {1}];
        nearestpt = Lookup[ptsToIndAssoc,
          Function[p, SelectFirst[ptc, Chop[# - p, 10^-5] == {0, 0, 0} &]] /@missing];
        res = Union[DeleteMissing[res]~Join~Flatten@Lookup[vertexToCell, nearestpt]]
      ];
      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[listableMeanFaces];
listableMeanFaces=Compile[{{faces,_Real,2}},
  Block[{facepart,edgelen},
    facepart=Partition[faces,2,1];
    AppendTo[facepart,{facepart[[-1,-1]],faces[[1]]}];
    edgelen=Table[Norm[Chop[First[i]-Last[i],10^-5]],{i,facepart}];
    Total[edgelen*(Mean/@facepart)]/Total[edgelen]
  ],CompilationTarget->"C",RuntimeOptions->"Speed",RuntimeAttributes->{Listable},
  CompilationOptions->{"InlineExternalDefinitions" -> True}
]*)
```

```
In[ ]:= ClearAll[meanFaces];
meanFaces = Compile[{{faces,_Real,2}},
  Block[{facepart, edgelen},
    facepart = Partition[faces, 2, 1];
    AppendTo[facepart, {facepart[[-1, -1]], faces[[1]]}];
    edgelen = Table[Norm[Chop[First[i] - Last[i], 10^-5]], {i, facepart}];
    Total[edgelen * (Mean /@ facepart)] / Total[edgelen]
  ], CompilationTarget -> "C", RuntimeOptions -> "Speed",
  CompilationOptions -> {"InlineExternalDefinitions" -> True}
]
```

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

```
In[ ]:= CompilePrint[meanFaces]
```

```
Out[ ]:=
```

1 argument

```

12 Integer registers
3 Real registers
10 Tensor registers
Underflow checking off
Overflow checking off
Integer overflow checking off
RuntimeAttributes -> {}

```

```

T(R2)0 = A1
I7 = 0
I4 = 10
I5 = 5
I0 = 2
I2 = -1
I1 = 1
Result = T(R1)7

```

```

1  T(I1)5 = {I0}
2  T(I1)3 = {I1}
3  T(R3)2 = Partition[ T(R2)0, T(I1)5, T(I1)3 ]
4  T(R1)5 = Part[ T(R3)2, I2, I2]
5  T(R1)3 = Part[ T(R2)0, I1]
6  T(R2)8 = {T(R1)5, T(R1)3}
7  T(I1)5 = {I2}
8  T(I2)3 = {T(I1)5}
9  T(R3)1 = Insert[ T(R3)2, T(R2)8, T(I2)3 ]
10 T(R3)2 = CopyTensor[ T(R3)1 ]
11 I9 = Length[ T(R3)2 ]
12 I6 = I7
13 T(R1)1 = Table[ I9]
14 I3 = I7
15 goto 27
16 T(R2)3 = GetElement[ T(R3)2, I3]
17 T(R1)7 = Part[ T(R2)3, I1]
18 T(R1)8 = Part[ T(R2)3, I2]
19 T(R1)4 = - T(R1)8
20 T(R1)7 = T(R1)7 + T(R1)4
21 I10 = Power[ I4, I5]
22 R2 = I10
23 R0 = Reciprocal[ R2]
24 T(R1)4 = Chop[ T(R1)7, R0]
25 R0 = Norm[ T(R1)4, I0, I7 ]
26 Element[ T(R1)1, I6] = R0
27 if[ ++ I3 <= I9] goto 16
28 I3 = Length[ T(R3)2]
29 I8 = I2
30 T(R2)3 = Table[ I3, I8]

```

```

31   I10 = I7
32   goto 40
33   T(R2)4 = GetElement[ T(R3)2, I10]
34   T(R1)8 = Total[ T(R2)4, I7] ]
35   I11 = Length[ T(R2)4]
36   R0 = I11
37   R1 = Reciprocal[ R0]
38   T(R1)9 = R1 * T(R1)8
39   Element[ T(R2)3, I8] = T(R1)9
40   if[ ++ I10 <= I3] goto 33
41   T(R2)7 = T(R1)1 * T(R2)3
42   T(R1)3 = Total[ T(R2)7, I7] ]
43   R1 = TotalAll[ T(R1)1, I7] ]
44   R0 = Reciprocal[ R1]
45   T(R1)7 = R0 * T(R1)3
46   Return

```

In[]:=

```

ClearAll[triangulateToMesh];
triangulateToMesh::Information =
  "the function takes in cell faces and triangulates them";
triangulateToMesh[faces_] := Block[{mf, partfaces},
  (*mf=SetPrecision[listableMeanFaces@faces,8];*)
  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}]
];

```

In[]:=

```

ClearAll[tToMesh];
With[{meanFaces = meanFaces},
  tToMesh = Compile[{{face, _Real, 2}},
    Block[{mf, partface, $face = face},
      If[Length[$face] ≠ 3,
        mf = meanFaces[$face];
        partface = Partition[$face, 2, 1];
        AppendTo[partface, {Last[$face], First[$face]}];
        Map[Join[#, {mf}] &, partface],
        {$face}
      ]
    ],
    CompilationTarget → "C", RuntimeOptions → "Speed", CompilationOptions →
      {"InlineCompiledFunctions" → True, "ExpressionOptimization" → False}
  ]
]

```

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

In[]:= CompilePrint[tToMesh]

Out[]:=

```

1 argument
1 Boolean register
13 Integer registers
3 Real registers
13 Tensor registers
Underflow checking off
Overflow checking off
Integer overflow checking off
RuntimeAttributes -> {}

T(R2)0 = A1
I8 = 0
I5 = 10
I6 = 5
I2 = 2
I4 = -1
I3 = 1
I1 = 3
Result = T(R3)12

1 T(R2)1 = CopyTensor[ T(R2)0 ]
2 I9 = Length[ T(R2)1 ]
3 B0 = I9 == I1
4 B0 = ! B0

```

```

5  if[ !B0] goto 74
6  T(R2)3 = CopyTensor[ T(R2)1]]
7  T(I1)8 = {I2}
8  T(I1)2 = {I3}
9  T(R3)5 = Partition[ T(R2)3, T(I1)8, T(I1)2]]
10 T(R1)8 = Part[ T(R3)5, I4, I4]
11 T(R1)2 = Part[ T(R2)3, I3]
12 T(R2)10 = {T(R1)8, T(R1)2}
13 T(I1)8 = {I4}
14 T(I2)2 = {T(I1)8}
15 T(R3)9 = Insert[ T(R3)5, T(R2)10, T(I2)2]]
16 T(R3)5 = CopyTensor[ T(R3)9]]
17 I0 = Length[ T(R3)5]
18 I11 = I8
19 T(R1)9 = Table[ I0]
20 I9 = I8
21 goto 33
22 T(R2)2 = GetElement[ T(R3)5, I9]
23 T(R1)6 = Part[ T(R2)2, I3]
24 T(R1)10 = Part[ T(R2)2, I4]
25 T(R1)4 = - T(R1)10
26 T(R1)6 = T(R1)6 + T(R1)4
27 I10 = Power[ I5, I6]
28 R2 = I10
29 R0 = Reciprocal[ R2]
30 T(R1)4 = Chop[ T(R1)6, R0]
31 R0 = Norm[ T(R1)4, I2, I8]]
32 Element[ T(R1)9, I11] = R0
33 if[ ++ I9 <= I0] goto 22
34 I9 = Length[ T(R3)5]
35 I7 = I4
36 T(R2)2 = Table[ I9, I7]
37 I10 = I8
38 goto 46
39 T(R2)4 = GetElement[ T(R3)5, I10]
40 T(R1)10 = Total[ T(R2)4, I8]]
41 I12 = Length[ T(R2)4]
42 R0 = I12
43 R1 = Reciprocal[ R0]
44 T(R1)12 = R1 * T(R1)10
45 Element[ T(R2)2, I7] = T(R1)12
46 if[ ++ I10 <= I9] goto 39
47 T(R2)6 = T(R1)9 * T(R2)2
48 T(R1)2 = Total[ T(R2)6, I8]]
49 R1 = TotalAll[ T(R1)9, I8]]
50 R0 = Reciprocal[ R1]
51 T(R1)6 = R0 * T(R1)2

```

```

52   T(I1)3 = {I2}
53   T(I1)7 = {I3}
54   T(R3)9 = Partition[ T(R2)1, T(I1)3, T(I1)7 ]
55   T(R1)3 = Part[ T(R2)1, I4]
56   T(R1)7 = Part[ T(R2)1, I3]
57   T(R2)5 = {T(R1)3, T(R1)7}
58   T(I1)3 = {I4}
59   T(I2)7 = {T(I1)3}
60   T(R3)2 = Insert[ T(R3)9, T(R2)5, T(I2)7 ]
61   T(R3)9 = CopyTensor[ T(R3)2 ]
62   I0 = Length[ T(R3)9]
63   I9 = I4
64   T(R3)2 = Table[ I0, I3, I9]
65   I7 = I8
66   goto 71
67   T(R2)5 = GetElement[ T(R3)9, I7]
68   T(R2)7 = {T(R1)6}
69   T(R2)12 = Join[ T(R2)5, T(R2)7 ]
70   Element[ T(R3)2, I9] = T(R2)12
71   if[ ++ I7 <= I0] goto 67
72   T(R3)12 = CopyTensor[ T(R3)2 ]
73   goto 76
74   T(R3)3 = {T(R2)1}
75   T(R3)12 = CopyTensor[ T(R3)3 ]
76   Return

```

In[]:=

```

ClearAll[localTriFunc];
With[{meanFaces = meanFaces},
  localTriFunc = Compile[{{ls, _Real, 2}, {pt, _Real, 1}},
    Block[{$face, k = 0, unit, len, mf, u},
      $face = ls;
      unit = Chop[Transpose[Transpose[$face] - pt], 10^-5];
      len = Length[unit];
      Do[
        If[unit[[i]] == {0, 0, 0},
          k = i;
          Break[];
        ], {i, len}
      ];
      If[k != 0,
        If[len == 3,
          $face,
          mf = meanFaces[$face];
          Which[
            k == 1,
            u = Compile`GetElement[$face, 1];
            {Compile`GetElement[$face, len], u, mf, u, Compile`GetElement[$face, 2], mf},
            k == len,
            u = Compile`GetElement[$face, len];
            {Compile`GetElement[$face, len - 1], u, mf, u, Compile`GetElement[$face, 1], mf},
            True,
            u = Compile`GetElement[$face, k];
            {Compile`GetElement[$face, k - 1], u, mf, u, Compile`GetElement[$face, k + 1], mf}
          ]
        ],
        {{0}}
      ]
    ], CompilationTarget -> "C", RuntimeOptions -> "Speed",
    RuntimeAttributes -> {Listable}, Parallelization -> True,
    CompilationOptions -> {"InlineCompiledFunctions" -> True}
  ]
]

```

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

In[]:= CompilePrint[localTriFunc]

Out[]:=

```

2 arguments
4 Boolean registers
16 Integer registers
4 Real registers

```

```

16 Tensor registers
Underflow checking off
Overflow checking off
Integer overflow checking off
RuntimeAttributes -> {Listable}

```

```

T(R2)0 = A1
T(R1)1 = A2
I0 = 0
I1 = 10
I2 = 5
T(I1)6 = {0, 0, 0}
T(I2)15 = {{0}}
I9 = 2
I11 = -1
I10 = 1
I8 = 3
R2 = 0.
Result = T(R2)11

```

```

1  I3 = I0
2  T(R2)5 = CopyTensor[ T(R2)0 ]
3  T(R2)2 = Transpose[ T(R2)5 ]
4  T(R1)3 = - T(R1)1
5  T(R2)4 = T(R2)2 + T(R1)3
6  T(R2)2 = Transpose[ T(R2)4 ]
7  I4 = Power[ I1, I2]
8  R0 = I4
9  R1 = Reciprocal[ R0]
10 T(R2)4 = Chop[ T(R2)2, R1]
11 I4 = Length[ T(R2)4]
12 I6 = I4
13 I7 = I0
14 goto 21
15 T(R1)2 = Part[ T(R2)4, I7]
16 B0 = CompareTensor[ I2, R2, T(R1)2, T(I1)6 ]
17 if[ !B0] goto 21
18 I3 = I7
19 goto 22
20 goto 21
21 if[ ++ I7 <= I6] goto 15
22 B0 = I3 == I0
23 B0 = ! B0
24 if[ !B0] goto 105
25 B1 = I4 == I8
26 if[ !B1] goto 29
27 T(R2)7 = CopyTensor[ T(R2)5 ]

```



```

28  goto 103
29  T(R2)2 = CopyTensor[ T(R2)5 ]
30  T(I1)10 = {I9}
31  T(I1)8 = {I10}
32  T(R3)7 = Partition[ T(R2)2, T(I1)10, T(I1)8 ]
33  T(R1)10 = Part[ T(R3)7, I11, I11 ]
34  T(R1)8 = Part[ T(R2)2, I10 ]
35  T(R2)13 = {T(R1)10, T(R1)8}
36  T(I1)10 = {I11}
37  T(I2)8 = {T(I1)10}
38  T(R3)3 = Insert[ T(R3)7, T(R2)13, T(I2)8 ]
39  T(R3)7 = CopyTensor[ T(R3)3 ]
40  I13 = Length[ T(R3)7 ]
41  I7 = I0
42  T(R1)3 = Table[ I13 ]
43  I6 = I0
44  goto 56
45  T(R2)8 = GetElement[ T(R3)7, I6 ]
46  T(R1)12 = Part[ T(R2)8, I10 ]
47  T(R1)13 = Part[ T(R2)8, I11 ]
48  T(R1)9 = - T(R1)13
49  T(R1)12 = T(R1)12 + T(R1)9
50  I14 = Power[ I1, I2 ]
51  R3 = I14
52  R1 = Reciprocal[ R3 ]
53  T(R1)9 = Chop[ T(R1)12, R1 ]
54  R1 = Norm[ T(R1)9, I9, I0 ]
55  Element[ T(R1)3, I7 ] = R1
56  if[ ++ I6 <= I13 ] goto 45
57  I6 = Length[ T(R3)7 ]
58  I12 = I11
59  T(R2)8 = Table[ I6, I12 ]
60  I14 = I0
61  goto 69
62  T(R2)9 = GetElement[ T(R3)7, I14 ]
63  T(R1)13 = Total[ T(R2)9, I0 ]
64  I15 = Length[ T(R2)9 ]
65  R1 = I15
66  R0 = Reciprocal[ R1 ]
67  T(R1)14 = R0 * T(R1)13
68  Element[ T(R2)8, I12 ] = T(R1)14
69  if[ ++ I14 <= I6 ] goto 62
70  T(R2)12 = T(R1)3 * T(R2)8
71  T(R1)8 = Total[ T(R2)12, I0 ]
72  R0 = TotalAll[ T(R1)3, I0 ]
73  R1 = Reciprocal[ R0 ]
74  T(R1)12 = R1 * T(R1)8

```

```

75   B2 = I3 == I10
76   if[ !B2] goto 83
77   T(R1)2 = GetElement[ T(R2)5, I10]
78   T(R1)11 = GetElement[ T(R2)5, I4]
79   T(R1)3 = GetElement[ T(R2)5, I9]
80   T(R2)7 = {T(R1)11, T(R1)2, T(R1)12, T(R1)2, T(R1)3, T(R1)12}
81   T(R2)8 = CopyTensor[ T(R2)7]]
82   goto 102
83   B3 = I3 == I4
84   if[ !B3] goto 93
85   T(R1)11 = GetElement[ T(R2)5, I4]
86   T(R1)2 = CopyTensor[ T(R1)11]]
87   I14 = I4 + I11
88   T(R1)11 = GetElement[ T(R2)5, I14]
89   T(R1)3 = GetElement[ T(R2)5, I10]
90   T(R2)8 = {T(R1)11, T(R1)2, T(R1)12, T(R1)2, T(R1)3, T(R1)12}
91   T(R2)11 = CopyTensor[ T(R2)8]]
92   goto 101
93   T(R1)11 = GetElement[ T(R2)5, I3]
94   T(R1)2 = CopyTensor[ T(R1)11]]
95   I14 = I3 + I11
96   T(R1)11 = GetElement[ T(R2)5, I14]
97   I14 = I3 + I10
98   T(R1)3 = GetElement[ T(R2)5, I14]
99   T(R2)14 = {T(R1)11, T(R1)2, T(R1)12, T(R1)2, T(R1)3, T(R1)12}
100   T(R2)11 = CopyTensor[ T(R2)14]]
101   T(R2)8 = CopyTensor[ T(R2)11]]
102   T(R2)7 = CopyTensor[ T(R2)8]]
103   T(R2)11 = CopyTensor[ T(R2)7]]
104   goto 107
105   T(R2)14 = CoerceTensor[ I8, T(I2)15]]
106   T(R2)11 = CopyTensor[ T(R2)14]]
107   Return

```

In[]:=

```

Clear[areaTriFn];
areaTriFn::Information = "areaTriFn finds the area of a triangle";
areaTriFn = Compile[{{face, _Real, 2}},
  Block[{v1, v2, f = face, firstelem, v1x, v1y, v1z, v2x, v2y, v2z, cross, vx, vy, vz},
    firstelem = Compile`GetElement[f, 1];
    v2 = Compile`GetElement[f, 2] - firstelem;
    v1 = Compile`GetElement[f, 3] - firstelem;
    {v1x, v1y, v1z} = v1;
    {v2x, v2y, v2z} = v2;
    (*0.5Norm[Cross[v2,v1]]*)
    {vx, vy, vz} = {v1z v2y - v1y v2z, -v1z v2x + v1x v2z, v1y v2x - v1x v2y};
    0.5 Sqrt[vx * vx + vy * vy + vz * vz]
  ], CompilationTarget -> "C", RuntimeOptions -> "Speed"
]

```

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

In[]:= CompilePrint[areaTriFn]

Out[]:=

```

1 argument
1 Boolean register
4 Integer registers
13 Real registers
6 Tensor registers
Underflow checking off
Overflow checking off
Integer overflow checking off
RuntimeAttributes -> {}

T(R2)0 = A1
I1 = 2
I0 = 1
I2 = 3
R11 = 0.5
Result = R9

1 T(R2)1 = CopyTensor[ T(R2)0 ]
2 T(R1)2 = GetElement[ T(R2)1, I0]
3 T(R1)3 = GetElement[ T(R2)1, I1]
4 T(R1)4 = - T(R1)2
5 T(R1)3 = T(R1)3 + T(R1)4
6 T(R1)4 = GetElement[ T(R2)1, I2]
7 T(R1)5 = - T(R1)2
8 T(R1)4 = T(R1)4 + T(R1)5
9 T(R1)5 = CopyTensor[ T(R1)4 ]

```

```

10   I3 = Length[ T(R1)5]
11   B0 = I3 == I2
12   B0 = ! B0
13   if[ !B0] goto 16
14   Return Error
15   goto 16
16   R0 = GetElement[ T(R1)5, I0]
17   R1 = GetElement[ T(R1)5, I1]
18   R2 = GetElement[ T(R1)5, I2]
19   T(R1)5 = CopyTensor[ T(R1)3]
20   I3 = Length[ T(R1)5]
21   B0 = I3 == I2
22   B0 = ! B0
23   if[ !B0] goto 26
24   Return Error
25   goto 26
26   R3 = GetElement[ T(R1)5, I0]
27   R4 = GetElement[ T(R1)5, I1]
28   R5 = GetElement[ T(R1)5, I2]
29   R6 = R2 * R4
30   R7 = R1 * R5
31   R8 = - R7
32   R6 = R6 + R8
33   R8 = - R2
34   R8 = R8 * R3
35   R7 = R0 * R5
36   R8 = R8 + R7
37   R7 = R1 * R3
38   R9 = R0 * R4
39   R10 = - R9
40   R7 = R7 + R10
41   T(R1)5 = {R6, R8, R7}
42   I3 = Length[ T(R1)5]
43   B0 = I3 == I2
44   B0 = ! B0
45   if[ !B0] goto 48
46   Return Error
47   goto 48
48   R6 = GetElement[ T(R1)5, I0]
49   R8 = GetElement[ T(R1)5, I1]
50   R7 = GetElement[ T(R1)5, I2]
51   R9 = R6 * R6
52   R10 = R8 * R8
53   R12 = R7 * R7
54   R9 = R9 + R10 + R12
55   R10 = Sqrt[ R9]
56   R9 = R11 * R10

```

57 Return

```
(*ClearAll[meanTri];
meanTri::Information="meanTri returns the centroid of the triangle";
meanTri=Compile[{{faces,_Real,2}},
  Mean@faces,
  CompilationTarget->"C",RuntimeAttributes->{Listable},RuntimeOptions->"Speed"
] *)
```

```
(*ClearAll[meanTri];
meanTri::Information="meanTri returns the centroid of the triangle";
meanTri=Compile[{{faces,_Real,2}},
  Block[{vec},
    vec=faces;
    (Compile`GetElement[vec,1]+
      Compile`GetElement[vec,2]+Compile`GetElement[vec,3])/3.0
  ],
  CompilationTarget->"C",RuntimeAttributes->{Listable},RuntimeOptions->"Speed"
] *)
```

In[]:=

```
ClearAll[meanTri];
meanTri::Information = "meanTri returns the centroid of the triangle";
meanTri = Compile[{{faces, _Real, 2}},
  Total[faces] / 3.0,
  CompilationTarget -> "C", RuntimeAttributes -> {Listable}, RuntimeOptions -> "Speed"
]
```

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

```
In[ ]:= CompilePrint[meanTri]
```

```
Out[ ]:=
```

```

1 argument
1 Integer register
1 Real register
3 Tensor registers
Underflow checking off
Overflow checking off
Integer overflow checking off
RuntimeAttributes -> {Listable}

T(R2)0 = A1
I0 = 0
R0 = 0.3333333333333333
Result = T(R1)2

1 T(R1)1 = Total[ T(R2)0, I0]
2 T(R1)2 = R0 * T(R1)1
3 Return
```

```

(*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"
]*)
```

In[]:=

```

ClearAll[triNormal];
triNormal::Information = "triNormal returns the normal of a triangle face";
triNormal = Compile[{{ls, _Real, 2}},
  Block[{vecs = ls, p1, v, w, vx, vy, vz, wx, wy, wz},
    p1 = Compile`GetElement[vecs, 1];
    v = Compile`GetElement[vecs, 2] - p1;
    w = Compile`GetElement[vecs, 3] - p1;
    vx = Compile`GetElement[v, 1];
    vy = Compile`GetElement[v, 2];
    vz = Compile`GetElement[v, 3];
    wx = Compile`GetElement[w, 1];
    wy = Compile`GetElement[w, 2];
    wz = Compile`GetElement[w, 3];
    {vy wz - vz wy, vz wx - vx wz, vx wy - vy wx}
  ], CompilationTarget -> "C", RuntimeOptions -> "Speed"
]

```

Out[]:=

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

```
In[ ]:= CompilePrint[triNormal]
```

```
Out[ ]:=
```

```

1 argument
3 Integer registers
11 Real registers
6 Tensor registers
Underflow checking off
Overflow checking off
Integer overflow checking off
RuntimeAttributes -> {}

T(R2)0 = A1
I1 = 2
I0 = 1
I2 = 3
Result = T(R1)5

1  T(R2)1 = CopyTensor[ T(R2)0 ]
2  T(R1)2 = GetElement[ T(R2)1, I0]
3  T(R1)3 = GetElement[ T(R2)1, I1]
4  T(R1)4 = - T(R1)2
5  T(R1)3 = T(R1)3 + T(R1)4
6  T(R1)4 = GetElement[ T(R2)1, I2]
7  T(R1)5 = - T(R1)2
8  T(R1)4 = T(R1)4 + T(R1)5
9  R0 = GetElement[ T(R1)3, I0]
10 R1 = GetElement[ T(R1)3, I1]
11 R2 = GetElement[ T(R1)3, I2]
12 R3 = GetElement[ T(R1)4, I0]
13 R4 = GetElement[ T(R1)4, I1]
14 R5 = GetElement[ T(R1)4, I2]
15 R6 = R1 * R5
16 R7 = R2 * R4
17 R8 = - R7
18 R6 = R6 + R8
19 R8 = R2 * R3
20 R7 = R0 * R5
21 R9 = - R7
22 R8 = R8 + R9
23 R9 = R0 * R4
24 R7 = R1 * R3
25 R10 = - R7
26 R9 = R9 + R10
27 T(R1)5 = {R6, R8, R9}
28 Return
```


In[]:=

```

ClearAll[normNormal];
normNormal::Information =
  "normNormal returns the normalized normal of a triangle face";
normNormal = Compile[{{ls, _Real, 1}},
  Block[{vecs = ls, x, y, z},
    x = Compile`GetElement[vecs, 1];
    y = Compile`GetElement[vecs, 2];
    z = Compile`GetElement[vecs, 3];
    {x, y, z} / Sqrt[(x x) + (y y) + (z z)]
  ], CompilationTarget -> "C", RuntimeOptions -> "Speed"
]

```

Out[]:=

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

```
In[ ]:= CompilePrint[normNormal]
```

```
Out[ ]:=
```

```

1 argument
3 Integer registers
6 Real registers
4 Tensor registers
Underflow checking off
Overflow checking off
Integer overflow checking off
RuntimeAttributes -> {}

T(R1)0 = A1
I1 = 2
I0 = 1
I2 = 3
Result = T(R1)3

1 T(R1)1 = CopyTensor[ T(R1)0 ]
2 R0 = GetElement[ T(R1)1, I0]
3 R1 = GetElement[ T(R1)1, I1]
4 R2 = GetElement[ T(R1)1, I2]
5 T(R1)2 = {R0, R1, R2}
6 R3 = R0 * R0
7 R4 = R1 * R1
8 R5 = R2 * R2
9 R3 = R3 + R4 + R5
10 R4 = Sqrt[ R3]
11 R3 = Reciprocal[ R4]
12 T(R1)3 = R3 * T(R1)2
13 Return

```

```
In[ ]:=
```

```

Clear[lenEdge];
lenEdge::Information = "lenEdge returns the length of an edge";
lenEdge = Compile[{{edge1, _Real, 1}, {edge2, _Real, 1}},
  Norm[Chop[edge1 - edge2, 10^-5]],
  CompilationTarget -> "C", CompilationOptions -> {"InlineExternalDefinitions" -> True},
  RuntimeOptions -> "Speed"
]

```

```
Out[ ]:= CompiledFunction[
```



Argument count: 2

Argument types: {{_Real, 1}, {_Real, 1}}

```
In[ ]:= CompilePrint[lenEdge]
```

```
Out[ ]:=
```

```

2 arguments
5 Integer registers
2 Real registers
5 Tensor registers
Underflow checking off
Overflow checking off
Integer overflow checking off
RuntimeAttributes -> {}

T(R1)0 = A1
T(R1)1 = A2
I4 = 0
I0 = 10
I1 = 5
I3 = 2
Result = R0

1  T(R1)3 = - T(R1)1
2  T(R1)4 = T(R1)0 + T(R1)3
3  I2 = Power[ I0, I1]
4  R1 = I2
5  R0 = Reciprocal[ R1]
6  T(R1)3 = Chop[ T(R1)4, R0]
7  R0 = Norm[ T(R1)3, I3, I4]
8  Return

```

centroid/volume for polyhedral cells

```
In[ ]:=
```

```

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

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

```

```

*)

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

ClearAll@volumePolyhedHelper;
volumePolyhedHelper = Compile[{{triFaces, _Real, 2}},
  Block[{V1, V2, V3, v1x, v1y, v1z, v2x, v2y, v2z},
    {V1, V2, V3} = triFaces;
    {v1x, v1y, v1z} = V1;
    {v2x, v2y, v2z} = V2;
    (-v1z v2y + v1y v2z) Compile`GetElement[V3, 1] + (v1z v2x - v1x v2z)
      Compile`GetElement[V3, 2] + (-v1y v2x + v1x v2y) Compile`GetElement[V3, 3]
  ], CompilationTarget -> "C", RuntimeAttributes -> {Listable},
  RuntimeOptions -> "Speed"
];
*)

ClearAll[volumePolyhedra];
volumePolyhedra::Information =
  "returns the volume of the triangulated polyhedral cell";
volumePolyhedra[facecollec_] :=
  1. / 6 Total[Flatten[Map[volumePolyhedHelper, facecollec, {2}]]];

ClearAll@volumePolyhedHelper;
volumePolyhedHelper = Compile[{{triFaces, _Real, 2}},
  Block[{V1, V2, V3, v1x, v1y, v1z, v2x, v2y, v2z},
    {V1, V2, V3} = triFaces;
    {v1x, v1y, v1z} = V1;
    {v2x, v2y, v2z} = V2;
    (-v1z v2y + v1y v2z) Compile`GetElement[V3, 1] + (v1z v2x - v1x v2z)
      Compile`GetElement[V3, 2] + (-v1y v2x + v1x v2y) Compile`GetElement[V3, 3]
  ], CompilationTarget -> "C", RuntimeOptions -> "Speed"
];

```

In[]:= CompilePrint[volumePolyhedHelper]

Out[]:=

```

1 argument
1 Boolean register
4 Integer registers
10 Real registers
5 Tensor registers
Underflow checking off
Overflow checking off
Integer overflow checking off

```

```

RuntimeAttributes -> {}

T(R2)0 = A1
I3 = 2
I2 = 1
I1 = 3
Result = R6

1  T(R2)1 = CopyTensor[ T(R2)0 ]
2  I0 = Length[ T(R2)1 ]
3  B0 = I0 == I1
4  B0 = ! B0
5  if[ !B0] goto 8
6  Return Error
7  goto 8
8  T(R1)2 = GetElement[ T(R2)1, I2]
9  T(R1)3 = GetElement[ T(R2)1, I3]
10 T(R1)4 = GetElement[ T(R2)1, I1]
11 T(R1)1 = CopyTensor[ T(R1)2 ]
12 I0 = Length[ T(R1)1 ]
13 B0 = I0 == I1
14 B0 = ! B0
15 if[ !B0] goto 18
16 Return Error
17 goto 18
18 R0 = GetElement[ T(R1)1, I2]
19 R1 = GetElement[ T(R1)1, I3]
20 R2 = GetElement[ T(R1)1, I1]
21 T(R1)1 = CopyTensor[ T(R1)3 ]
22 I0 = Length[ T(R1)1 ]
23 B0 = I0 == I1
24 B0 = ! B0
25 if[ !B0] goto 28
26 Return Error
27 goto 28
28 R3 = GetElement[ T(R1)1, I2]
29 R4 = GetElement[ T(R1)1, I3]
30 R5 = GetElement[ T(R1)1, I1]
31 R6 = - R2
32 R6 = R6 * R4
33 R7 = R1 * R5
34 R6 = R6 + R7
35 R7 = GetElement[ T(R1)4, I2]
36 R6 = R6 * R7
37 R7 = R2 * R3
38 R8 = R0 * R5
39 R9 = - R8

```

```
40    R7 = R7 + R9
41    R9 = GetElement[ T(R1)4, I3]
42    R7 = R7 * R9
43    R9 = - R1
44    R9 = R9 * R3
45    R8 = R0 * R4
46    R9 = R9 + R8
47    R8 = GetElement[ T(R1)4, I1]
48    R9 = R9 * R8
49    R6 = R6 + R7 + R9
50    Return
```

In[]:=

```

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

ClearAll@centroidPolyhedraHelper;
With[{triNormal=triNormal},
  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"
  ]
]
]*)

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

ClearAll@centroidPolyhedraHelper;
With[{triNormal = triNormal},
  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", RuntimeOptions -> "Speed"
  ]
]
]

```

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

```
In[ ]:= CompilePrint[centroidPolyhedraHelper]
```

```
Out[ ]:=
```

```

1 argument
1 Boolean register
4 Integer registers
9 Tensor registers
Underflow checking off
Overflow checking off
Integer overflow checking off
RuntimeAttributes -> {}

T(R2)0 = A1
I3 = 2
I2 = 1
I1 = 3
Result = T(R1)7

1  T(R2)1 = CopyTensor[ T(R2)0 ]
2  I0 = Length[ T(R2)1 ]
3  B0 = I0 == I1
4  B0 = ! B0
5  if[ !B0] goto 8
6  Return Error
7  goto 8
8  T(R1)2 = GetElement[ T(R2)1, I2]
9  T(R1)3 = GetElement[ T(R2)1, I3]
10 T(R1)4 = GetElement[ T(R2)1, I1]
11 T(R1)1 = LibraryFunction[<>,
    compiledFunction5, {{Real, 2, Constant}}, {Real, 1}} [ T(R2)0 ]
12 T(R1)5 = T(R1)2 + T(R1)3
13 T(R1)6 = Square[ T(R1)5]
14 T(R1)5 = T(R1)3 + T(R1)4
15 T(R1)7 = Square[ T(R1)5]
16 T(R1)5 = T(R1)4 + T(R1)2
17 T(R1)8 = Square[ T(R1)5]
18 T(R1)6 = T(R1)6 + T(R1)7 + T(R1)8
19 T(R1)7 = T(R1)1 * T(R1)6
20 Return
```


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
];

```

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 = SetPrecision[polyhedCentAssoc[#] + shiftvec[key][#], 8];
              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[{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}]
```

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

In[]:= CompilePrint[rotateSourceVertOrderFn]

Out[]=

```

2 arguments
4 Boolean registers
15 Integer registers
3 Real registers
9 Tensor registers
Underflow checking off
Overflow checking off
Integer overflow checking off
RuntimeAttributes -> {}

T(R1)0 = A1
T(R3)1 = A2
I0 = 0
T(R1)5 = {0., 0., 0.}
I6 = 10
I7 = 8
I9 = 5
I10 = 2
I12 = -1
I4 = 1
I11 = 3
R2 = 0.
Result = T(R3)3

1  I5 = I0
2  I8 = Length[ T(R3)1]
3  I13 = I12
4  T(R3)3 = Table[ I8, I4, I13]
5  I3 = I0
6  goto 46
7  T(R2)4 = GetElement[ T(R3)1, I3]
8  T(R1)6 = GetElement[ T(R2)4, I4]
9  T(R1)8 = - T(R1)0
10 T(R1)6 = T(R1)6 + T(R1)8
11 I14 = Power[ I6, I7]
12 R0 = I14
13 R1 = Reciprocal[ R0]
14 T(R1)8 = Chop[ T(R1)6, R1]
15 B0 = CompareTensor[ I9, R2, T(R1)8, T(R1)5]]
16 if[ !B0] goto 19
17 T(R2)8 = CopyTensor[ T(R2)4]]
18 goto 45
19 T(R1)8 = GetElement[ T(R2)4, I10]
20 T(R1)6 = - T(R1)0
21 T(R1)8 = T(R1)8 + T(R1)6
22 I14 = Power[ I6, I7]

```

```

23   R1 = I14
24   R0 = Reciprocal[ R1]
25   T(R1)6 = Chop[ T(R1)8, R0]
26   B1 = CompareTensor[ I9, R2, T(R1)6, T(R1)5] ]
27   if[ !B1] goto 30
28   I5 = I10
29   goto 41
30   T(R1)6 = GetElement[ T(R2)4, I11]
31   T(R1)8 = - T(R1)0
32   T(R1)6 = T(R1)6 + T(R1)8
33   I14 = Power[ I6, I7]
34   R0 = I14
35   R1 = Reciprocal[ R0]
36   T(R1)8 = Chop[ T(R1)6, R1]
37   B2 = CompareTensor[ I9, R2, T(R1)8, T(R1)5] ]
38   if[ !B2] goto 41
39   I5 = I11
40   goto 41
41   I14 = I5 + I12
42   T(I1)8 = { I14}
43   T(R2)6 = RotateLeft[ T(R2)4, T(I1)8] ]
44   T(R2)8 = CopyTensor[ T(R2)6] ]
45   Element[ T(R3)3, I13] = T(R2)8
46   if[ ++ I3 <= I8] goto 7
47   Return


```

In[]:=

```

ClearAll@surfaceGradHelper;
surfaceGradHelper::Information = "surfaceGradHelper ...";
surfaceGradHelper =
Compile[{{point, _Real, 1}, {tri, _Real, 3}, {normals, _Real, 2}, {coeffs, _Real, 1}},
Block[
{ptTri, source = point, rotatedTris,
normal, u2, u1, cross, sumVec = {0., 0., 0.}, coeff},
rotatedTris = rotateSourceVertOrderFn[source, tri];
Do[
ptTri = rotatedTris[[i]];
normal = normals[[i]];
coeff = coeffs[[i]];
{u1, u2} = {ptTri[[2]], ptTri[[1]]};
cross = Cross[normal, u2 - u1];
sumVec += (coeff * (1./2.) cross), {i, Length@normals}];
sumVec
], CompilationTarget -> "C", RuntimeOptions -> "Speed",
CompilationOptions ->
{"ExpressionOptimization" -> True, "InlineExternalDefinitions" -> True}]

```

Out[]:= CompiledFunction[ Argument count: 4
Argument types: {{_Real, 1}, {_Real, 3}, {_Real, 2}, {_Real, 1}}]

In[]:= CompilePrint[surfaceGradHelper]

Out[]:=

```

4 arguments
1 Boolean register
9 Integer registers
5 Real registers
15 Tensor registers
Underflow checking off
Overflow checking off
Integer overflow checking off
RuntimeAttributes -> {}

T(R1)0 = A1
T(R3)1 = A2
T(R2)2 = A3
T(R1)3 = A4
I3 = 0
R1 = 1.
T(R1)5 = {0., 0., 0.}
R2 = 2.

```

```

I5 = 2
I6 = -1
I8 = 1
Result = T(R1)9

1  T(R1)11 = CopyTensor[ T(R1)0 ]
2  T(R1)9 = CopyTensor[ T(R1)5 ]
3  T(R3)10 = LibraryFunction[<>, compiledFunction10, {{Real,
1, Constant}}, {Real, 3, Constant}}, {Real, 3}][ T(R1)11, T(R3)1 ]
4  I2 = Length[ T(R2)2 ]
5  I4 = I3
6  goto 30
7  T(R2)8 = Part[ T(R3)10, I4]
8  T(R1)6 = Part[ T(R2)2, I4]
9  R0 = Part[ T(R1)3, I4]
10 T(R1)4 = Part[ T(R2)8, I5]
11 T(R1)7 = Part[ T(R2)8, I6]
12 T(R2)13 = {T(R1)4, T(R1)7}
13 I7 = Length[ T(R2)13]
14 B0 = I7 == I5
15 B0 = ! B0
16 if[ !B0] goto 19
17 Return Error
18 goto 19
19 T(R1)4 = GetElement[ T(R2)13, I8]
20 T(R1)7 = GetElement[ T(R2)13, I5]
21 T(R1)13 = - T(R1)4
22 T(R1)12 = T(R1)7 + T(R1)13
23 T(R1)13 = Cross[ T(R1)6, T(R1)12 ]
24 R4 = Reciprocal[ R2]
25 R3 = R1 * R4
26 R4 = R0 * R3
27 T(R1)12 = R4 * T(R1)13
28 T(R1)14 = T(R1)9 + T(R1)12
29 T(R1)9 = CopyTensor[ T(R1)14 ]
30 if[ ++ I4 <= I2] goto 7
31 Return

```

In[]:=

```

Clear[makeSurfaceType];
makeSurfaceType[sign_, type_] := Block[{}],
  If[sign == 1, {0, type}, {-1, type}]
];

Clear@surfaceGrad;
surfaceGrad[point_, assocTriAssoc_, normalAssoc_] :=
Block[{triSurfClassifier, key, vals, surfclasscoeff, norms, triangs, temp},
  triSurfClassifier = (key = First[Keys@#]; vals = Values@#;
    key → vals
  ) & /@ GatherBy[Normal@AssociationMap[Thread@*Reverse, assocTriAssoc],
    Intersection@*Keys] // Flatten;
  (*triSurfClassifier = (keys=Keys@#;vals=Values@#;
    Flatten@If[Length@keys≠1, #→vals&/@keys, keys→vals]
  ) & /@ GatherBy[Normal@AssociationMap[Thread@*Reverse, assocTriAssoc],
    Intersection@*Keys] // Flatten;
  triSurfClassifier = DeleteDuplicatesBy[triSurfClassifier, Intersection@*Keys]; *)
  triSurfClassifier = Map[
    (key = Keys[#]; vals = Values[#];
      temp = Lookup[celltypes, vals];
      If[Length[vals] == 1,
        key → makeSurfaceType[Sign[normalAssoc[key][[3]]], First@temp], key → temp]
    ) &, triSurfClassifier, {1}];
  surfclasscoeff = Lookup[surfCoeff, Values@triSurfClassifier];
  triangs = Keys[triSurfClassifier];
  norms = Lookup[normalAssoc, triangs];
  surfaceGradHelper[point, triangs, norms, surfclasscoeff]
]

```

volumetric ▽

In[]:=

```
Clear@volumeGrad;
volumeGrad::Information = "volumeGrad computes the volume gradient about a point";
Clear@volumeGrad;
volumeGrad[areaAssoc_, triNormAssoc_, polyVols_, cellids_] :=
  Block[{gradV, vol, growindkeys, kcvals, inters},
    gradV =  $\left(\frac{1.0}{3}\right)$  Total /@ (areaAssoc * triNormAssoc);
    kcvals = AssociationThread[cellids → (kcvAssoc~Lookup~Lookup[celltypes, cellids])];
    vol = AssociationThread[cellids → ConstantArray[Vo, Length@cellids]];
    inters = Intersection[cellids, growingcellIndices];
    If[inters ≠ {},
      growindkeys = Replace[inters, k_Integer → {Key[k]}, {1}]
    ];
    If[growindkeys ≠ {},
      vol = MapAt[(1 + Vgrowth time) # &, vol, growindkeys]
    ];
    Total[ $\left(\frac{kcvals}{vol}\right) \left(\frac{polyVols}{vol} - 1\right) gradV$ ]
  ]
```


vertex ▽

In[]:=

```
Clear@gradientVertex;
gradientVertex::Information =
  "determines the sum of the gradients of all the potentials about a vertex";
gradientVertex[indToPtsAssoc_, triareaAssoc_, triNormalassoc_,
  associatedtris_, topoF_, polyhedVol_] := Block[{vkeys = Keys@indToPtsAssoc,
  triNormalSubAssoc, areaSubAssoc, associatedTri, triNormalpairings, cellKeys,
  associatedtrisAssoc, polyvol, volGrad, pt, surfGrad, normalsAssoc},
  Table[
    pt = indToPtsAssoc[i];
    areaSubAssoc = triareaAssoc[i];
    triNormalSubAssoc = triNormalassoc[i];
    associatedtrisAssoc = associatedtris[i];
    associatedTri = Flatten[Values@associatedtrisAssoc, 1];
    triNormalpairings = AssociationThread[associatedTri → Values@triNormalSubAssoc];
    cellKeys = Keys@First@topoF[i];
    polyvol = AssociationThread[cellKeys → Lookup[polyhedVol, cellKeys]];
    normalsAssoc = Map[Lookup[triNormalpairings, #] &, associatedtrisAssoc];
    volGrad = volumeGrad[areaSubAssoc, normalsAssoc, polyvol, cellKeys];
    surfGrad = surfaceGrad[pt, associatedtrisAssoc, triNormalpairings];
    - (volGrad + surfGrad),
    {i, vkeys}
  ]
];
```

pair boundary points for computing ∇

In[]:=

```

Clear@boundaryPtsPairing;
With[{xlim1 = xLim[[1]], xlim2 = xLim[[2]], ylim1 = yLim[[1]], ylim2 = yLim[[2]]},
  boundaryPtsPairing::Information =
    "the function pairs the points at the boundaries with
    corresponding mirror points";
  boundaryPtsPairing[vertexToCell_, indToPtsAssoc_, ptsToIndAssoc_] := Block[{outerpts,
    mirrorpairs, pt, posits, boundarypts, mirror, keys = Keys[ptsToIndAssoc], fpos},
    outerpts = Keys@Select[vertexToCell, Length[#] ≠ 3 &];
    mirrorpairs = <|
      (pt = Lookup[indToPtsAssoc, #];
      If[
        pt[[1]] ≤ xlim1 || pt[[1]] ≥ xlim2 || pt[[2]] ≤ ylim1 || pt[[2]] ≥ ylim2,
        mirror = Lookup[ptsToIndAssoc, {pt /. periodicRules},
          (fpos = With[{insert = pt /. periodicRules},
            FirstPosition[Chop[# - insert & /@ keys, 10^-5], {0, 0, 0}]
          ]);
        ptsToIndAssoc[Extract[keys, fpos]])
      ];
      # → mirror,
      Nothing]) & /@ outerpts
    |> // KeySort;
  boundarypts =
    Map[Sort@*Flatten][List @@@ Normal@GroupBy[Normal@mirrorpairs, Last → First]];
  (*(*option 1:*)
  DeleteDuplicates[ReverseSortBy[boundarypts, Length], Length[#1 ∩ #2] == 2 &];*)
  (*option 2:*)
  posits = Position[boundarypts, x_ /; Length[x] == 3];
  Complement[boundarypts,
    Flatten[Map[Permutations[#, {2}] &, Extract[boundarypts, posits]], 1]
  ]
];

```

integrating mesh

```
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[ ]:= partition = Compile[{{ls, _Real, 2}},
  Partition[ls, 3],
  CompilationTarget -> "C", RuntimeOptions -> "Speed"
];

With[{faceorientedQ = OptionValue[paramFinder, "OrientedFaces"]},
  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, $triAreaAssoc},
    (*Adjust params for the the new geometry M-X*)
    $faceListAssoc = Map[Lookup[indToPtsAssoc, #] &, $CVG, {2}];
    $topo = cellTranslator[$faceListAssoc, keyslocaltopo, shiftVecAssoc, vertKeys];
    $assoctri = With[{keys = Keys[indToPtsAssoc]},
      SetPrecision[
        AssociationThread[keys,
          Table[Map[Splice@partition@localTriFunc[#, indToPtsAssoc[i]] &,
            $topo[i][[1]], {2}], {i, keys}]], 8]
      ];
    opencloseTri = Flatten[Values@#, 1] & /@ $assoctri;
    (*trimesh=triangulateToMesh/@$faceListAssoc;*)
    trimesh = Map[tToMesh, $faceListAssoc, {2}];
    If[faceorientedQ,
      $polyhedVol = volumePolyhedra /@ trimesh;
      polyhedcent = <|
```

```

    KeyValueMap[#1 → 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 = Values /@ Map[triNormal, $assoctri, {3}];
  normNormals = Map[normNormal, normals, {3}];
  signednormals = normNormals;
  (*centTri=Map[SetPrecision[#,8]&, <|#→meanTri[Values[$assoctri@#]] & /@ vertKeys |>];
  normNormals=Map[Normalize,normals,{3}];
  signednormals=AssociationThread[vertKeys,
    Map[
      MapThread[
        #2 Sign@MapThread[Function[{x,y}, (y-#1).x], {#2,#3}] &,
        {cellcent[#,normNormals[#,centTri[#]]} &, vertKeys]]];*)
  If[Not@faceorientedQ,
    centTri =
      Map[SetPrecision[#, 8] &, <|# → meanTri[Values[$assoctri@#]] & /@ vertKeys |>];
    signs = AssociationThread[Keys@indToPtsAssoc,
      Map[
        MapThread[
          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}]
      ];
  ];
  $triAreaAssoc = Map[areaTriFn, $assoctri, {3}];
  $vertTriNormpair = <|
    # → <| Thread[opencloseTri[#] → Flatten[signednormals@#, 1]] |> & /@ vertKeys |>;
  $triDistAssoc = (GroupBy[GatherBy[#, Intersection], Length, Flatten[#, 1] &]) & /@
    opencloseTri;
  {$triAreaAssoc, $triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol}
]
];

```

Integrator

In[]:=

```
Clear@RK4Integrator;
RK4Integrator::Information = "the module comprises
  the Runge-Kutta Order 5 (RK5) scheme for integrating the mesh";
RK4Integrator[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,
  $triAreaAssoc, $indToPtsAssocOrig = indToPtsAssoc, bptpairs, vals},
(*computed once at the start of the computation*)
keyslocaltopo = Keys@*First /@ topo;
shiftVecAssoc = Association /@
  Map[Apply[Rule], Thread /@ Select[ (#[[2 ;; 3]]) & /@ topo, # ≠ {{}}, {} &], {2}];
bptpairs = boundaryPtsPairing[$vertexToCell, indToPtsAssoc, ptsToIndAssoc];
(*If[time==1*δt || time==2δt || time==3*δt ,Print@bptpairs];*)
{$triAreaAssoc, $triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol} =
  paramFinder[$indToPtsAssoc, $CVG, keyslocaltopo, shiftVecAssoc, vertKeys];
(*compute the gradient of the original mesh M1 *)
grad1 = AssociationThread[vertKeys,
  SetPrecision[gradientVertex[$indToPtsAssoc,
    $triAreaAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol], 8]];
grad1 = adjustGrad[grad1, bptpairs];
(*displace vertices by the numerical gradient*)
$indToPtsAssoc = <|
  KeyValueMap[#1 → SetPrecision[#2 + 0.5 grad1[#1] δt, 8] &, $indToPtsAssocOrig] |>;
(*adjust and compute params for the intermediate geometry M2*)
{$triAreaAssoc, $triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol} =
  paramFinder[$indToPtsAssoc, $CVG, keyslocaltopo, shiftVecAssoc, vertKeys];
(*compute the gradient for M2*)
grad2 = AssociationThread[vertKeys,
  SetPrecision[gradientVertex[$indToPtsAssoc,
    $triAreaAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol], 8]];
grad2 = adjustGrad[grad2, bptpairs];
(*displace vertices by the numerical gradient*)
$indToPtsAssoc = <|
  KeyValueMap[#1 → SetPrecision[#2 + 0.5 (grad2[#]) δt, 8] &, $indToPtsAssocOrig] |>;
(*adjust and compute params for the intermediate geometry M3*)
{$triAreaAssoc, $triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol} =
  paramFinder[$indToPtsAssoc, $CVG, keyslocaltopo, shiftVecAssoc, vertKeys];
(*compute the gradient for M3*)
grad3 = AssociationThread[vertKeys,
  SetPrecision[gradientVertex[$indToPtsAssoc,
    $triAreaAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol], 8]];
grad3 = adjustGrad[grad3, bptpairs];
```

```

(*displace vertices by the numerical gradient*)
$indToPtsAssoc = <|
  KeyValueMap[#1 → SetPrecision[#2 + (grad3[#]) δt, 8] &, $indToPtsAssocOrig] |>;
(*adjust and compute params for the intermediate geometry M4*)
{$triAreaAssoc, $triDistAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol} =
  paramFinder[$indToPtsAssoc, $CVG, keyslocaltopo, shiftVecAssoc, vertKeys];
(*compute the gradient for M4*)
grad4 = AssociationThread[vertKeys,
  SetPrecision[gradientVertex[$indToPtsAssoc,
    $triAreaAssoc, $vertTriNormpair, $assoctri, $topo, $polyhedVol], 8]];
grad4 = adjustGrad[grad4, bptpairs];
(*displace vertices by the numerical gradient*)
$indToPtsAssoc = Map[SetPrecision[#, 8] &] [ <|
  KeyValueMap[#1 → #2 + (1. / 6) (grad1[#1] + 2. grad2[#] + 2. grad3[#] + grad4[#]) δt &,
    $indToPtsAssocOrig] |>]
];

```

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

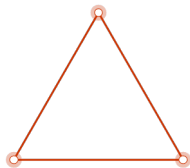
checks to determine whether α , β , γ or an invalid pattern is present in the graph

```

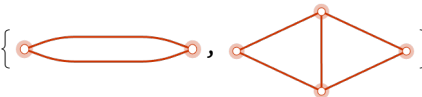
In[ ]:=
$invalidPatternsEdge = Graph[{1 ↔ 2, 2 ↔ 3, 3 ↔ 1}];
Echo[Graph[$invalidPatternsEdge, PlotTheme → "Web", ImageSize → Tiny],
  Style["invalid pattern (edge in Δ): ", Black]];
$invalidPatterns = {
  Graph[{1 ↔ 2, 2 ↔ 1}],
  Graph[{1 ↔ 2, 2 ↔ 3, 3 ↔ 1, 1 ↔ 4, 3 ↔ 4}]
};
Echo[Graph[#, PlotTheme →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

```

» invalid pattern (edge in Δ):



» invalid pattern (double edge || double Δ): {



```

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[]:=

```

ItoDelta[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 -  $\delta$ ], 1];
(*select edges that have length less than critical value  $\delta$ *)
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  $\alpha$  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 ItoΔ operation *)
  modifiednetwork = ItoΔoperation[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 ↦ 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]),
        {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[]:=

```

 $\Delta$ toIoperation[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]} :> {p, x}];
(*creating a graph from the current topology*)
graphCurrentTopology =
Graph@Replace[currentTopology, List -> UndirectedEdge, {2}, Heads -> True];
If[! InvalidTrigonalPatternQ[graphCurrentTopology],
(* transform the network topology by applying [H]->[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->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->Small];*)
rules = rules $\Delta$ toI[currentTopology, ptsTri, PtPartition];
Switch[
rules, _rules $\Delta$ toI, None,
_,
(updatedLocalNetwork =  $\Delta$ toIoperation[prevNetwork, rules];
celltopologicalChanges = bindCellsToNewTopology[adjoiningCells,
updatedLocalNetwork] /. _bindCellsToNewTopology -> {}];
If[celltopologicalChanges != {},
(*Print[Graphics3D[
{{Opacity[0.1],Blue,Polyhedron/@Values[celltopologicalChanges]},
{Red,Line[candidate~Append~First[candidate]]}},Axes->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}
];

```

In[]:= |

visualizing mesh

In[]:=

```

ClearAll[displayMesh];
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[triangulateToMesh@#, 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)

```
In[ ]:= $cellfaces = Map[Lookup[indToPtsAssoc, #] &, cellVertexGrouping, {2}];  
$mesh = Values[Polyhedron@Flatten[triangulateFaces@#, 1] & /@$cellfaces];
```

```
In[ ]:= Mean[Volume@$mesh]
```

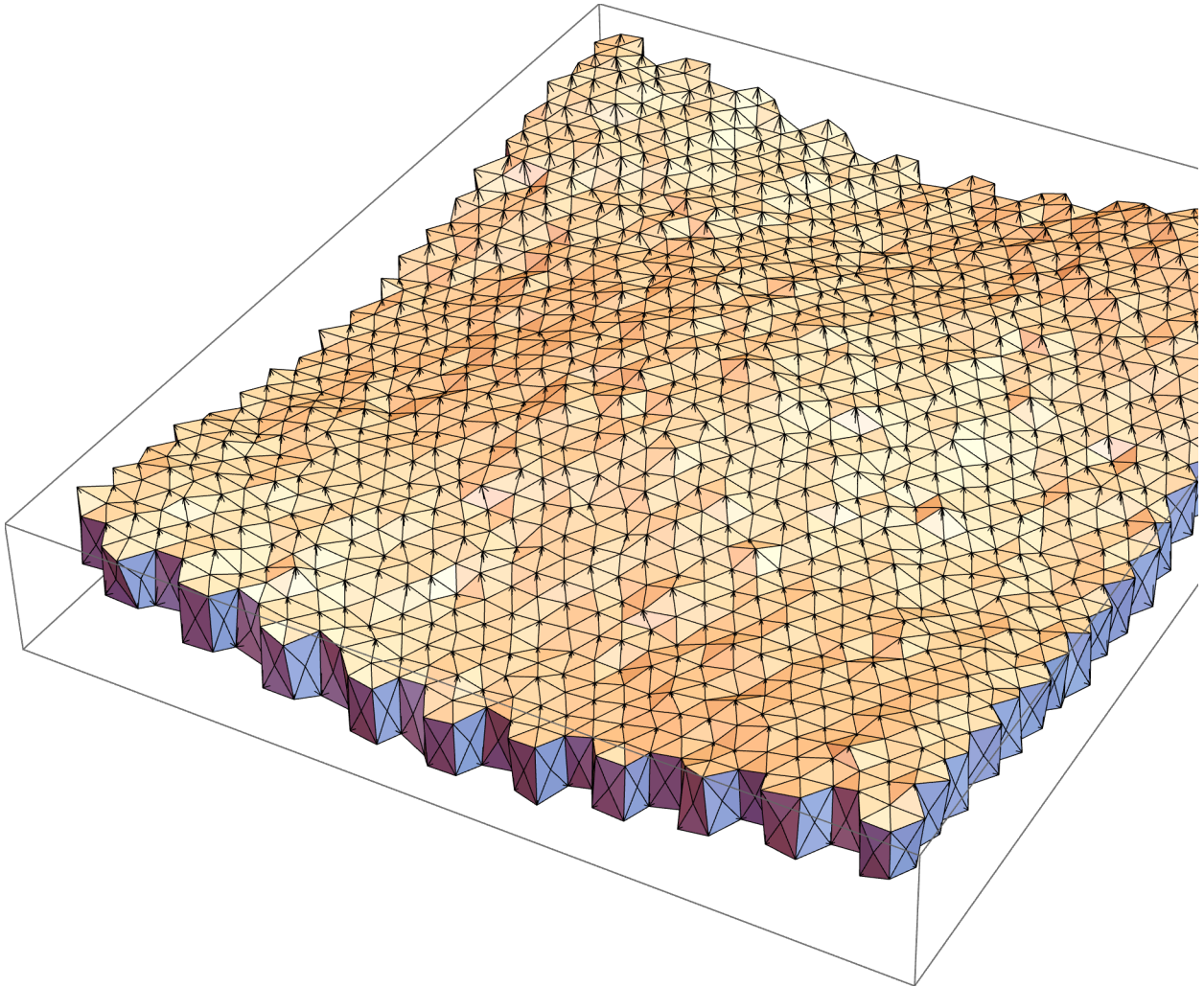
```
Out[ ]:= 0.525412
```

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 ($I \rightarrow \Delta$) & ($\Delta \rightarrow I$) transitions
 - perform RK4 mesh integration
 - time += δt
5. visualize mesh

starting mesh

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

Out[]:=



test module (single time-step)

termination condition:

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

```

In[ ]:= time = 1 * dt;
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["RK4 integration Started"];
      $indToPtsAssoc = RK4Integrator[$indToPtsAssoc,
        $topo, $cellVertexGroup, $vertexToCell, $ptsToIndAssoc];
      $ptsToIndAssoc = AssociationMap[Reverse, $indToPtsAssoc];
      Echo["Integration Done"];
      $indToPtsAssoc
    ]
  ];

```

```

In[ ]:= ClearSystemCache[];

```

```

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

```

```

» RK4 integration Started

```

```

» Integration Done

```

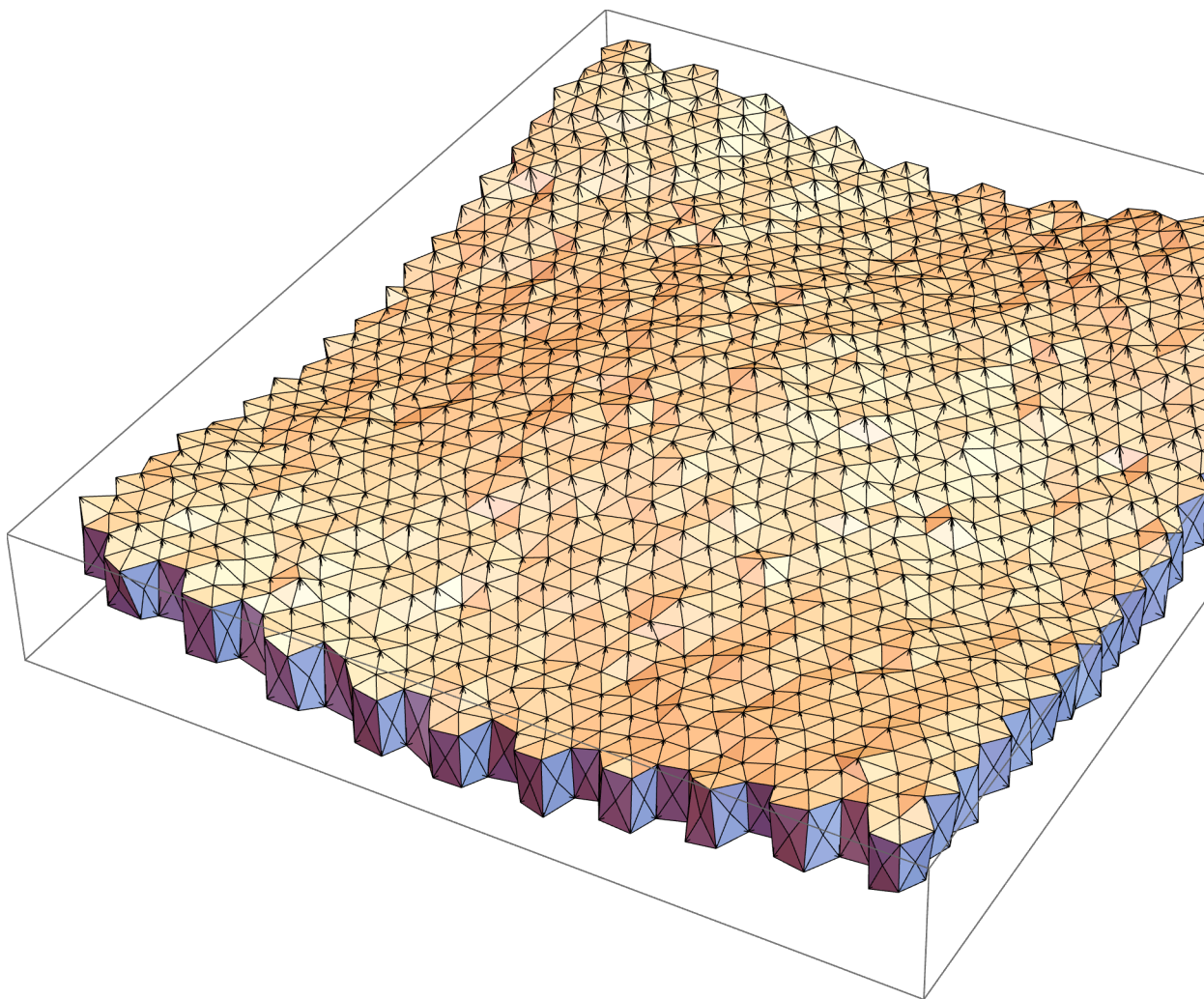
```

Out[ ]:= {8.67188, Null}

```

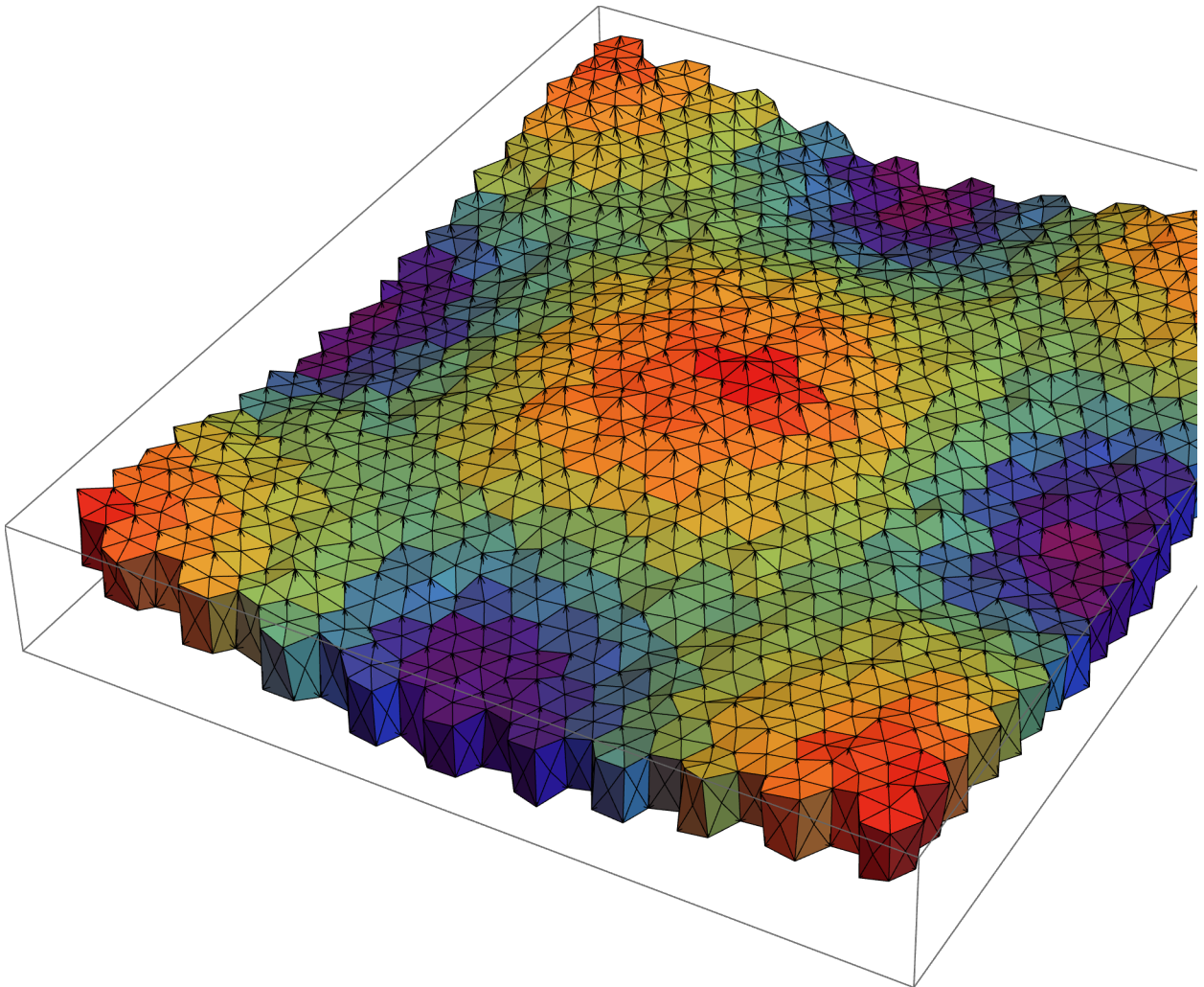
```
In[ ]:= resSingleStep~visualizeMesh~cellVertexGrouping
```

Out[]:=




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

Out[]:=

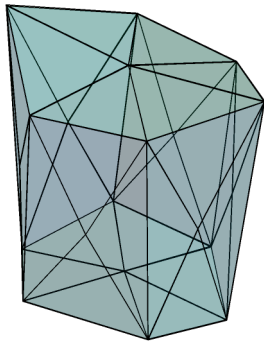


```

In[ ]:= 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]

```

Out[]:=



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 = 30},
time = counter *  $\delta t$ ;
Echo[time, "time: "];
While[counter ≤ endc,
Echo[counter, "counter: "];
$faceListCoords = Map[Lookup[$indToPtsAssoc, #] &, $cellVertexGroup, {2}];
$wrappedMat =
AssociationThread[cellIds → Map[Lookup[$indToPtsAssoc, #] /. periodicRules &,
Lookup[$cellVertexGroup, cellIds], {2}]];
PrintTemporary[" ----- construct local topology ----- "];
vertKeys = Keys@$indToPtsAssoc;
cellIds = Keys@$cellVertexGroup;
boundarycells = outerCellsFn[$faceListCoords, $vertexToCell, $ptsToIndAssoc];
(*Print@boundarycells;*)
$wrappedMatBound = $wrappedMat ~KeyTake~ boundarycells;
topo = <|# → (getLocalTopology[$ptsToIndAssoc,
  $indToPtsAssoc, $vertexToCell, $cellVertexGroup,
  $wrappedMatBound, $faceListCoords] [$indToPtsAssoc[#]]) & /@ vertKeys|>;
(*Print[Values@Map[Length@*First, topo] // Counts];*)
(*Print[
Union@Values[Length@DeleteDuplicates@Flatten[Values[First@#, 2] & /@ topo]]];*)
PrintTemporary[" ----- Integrate mesh (RK4) ----- "];
$indToPtsAssoc = RK4Integrator[$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 +=  $\delta t$ ; 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: 0.005
```

```
» 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
```

```
» counter: 21
```

```
» counter: 22
```

```
» counter: 23
```

```
» counter: 24
```

```
» counter: 25
```

```
» counter: 26
```

```
» counter: 27
```

```
» counter: 28
```

```
» counter: 29
```

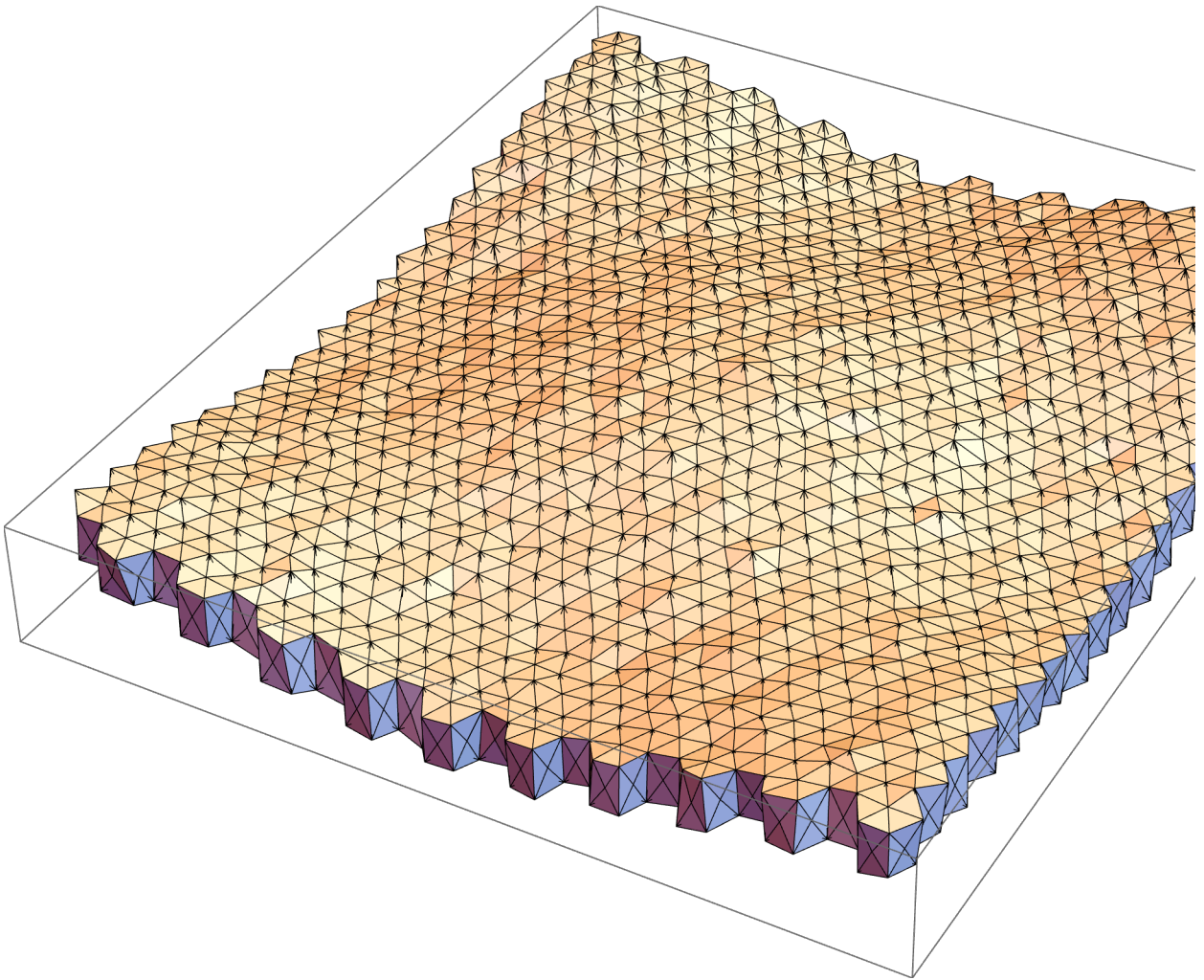
```
» counter: 30
```

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

```
Out[8]= {278.631, Null}
```

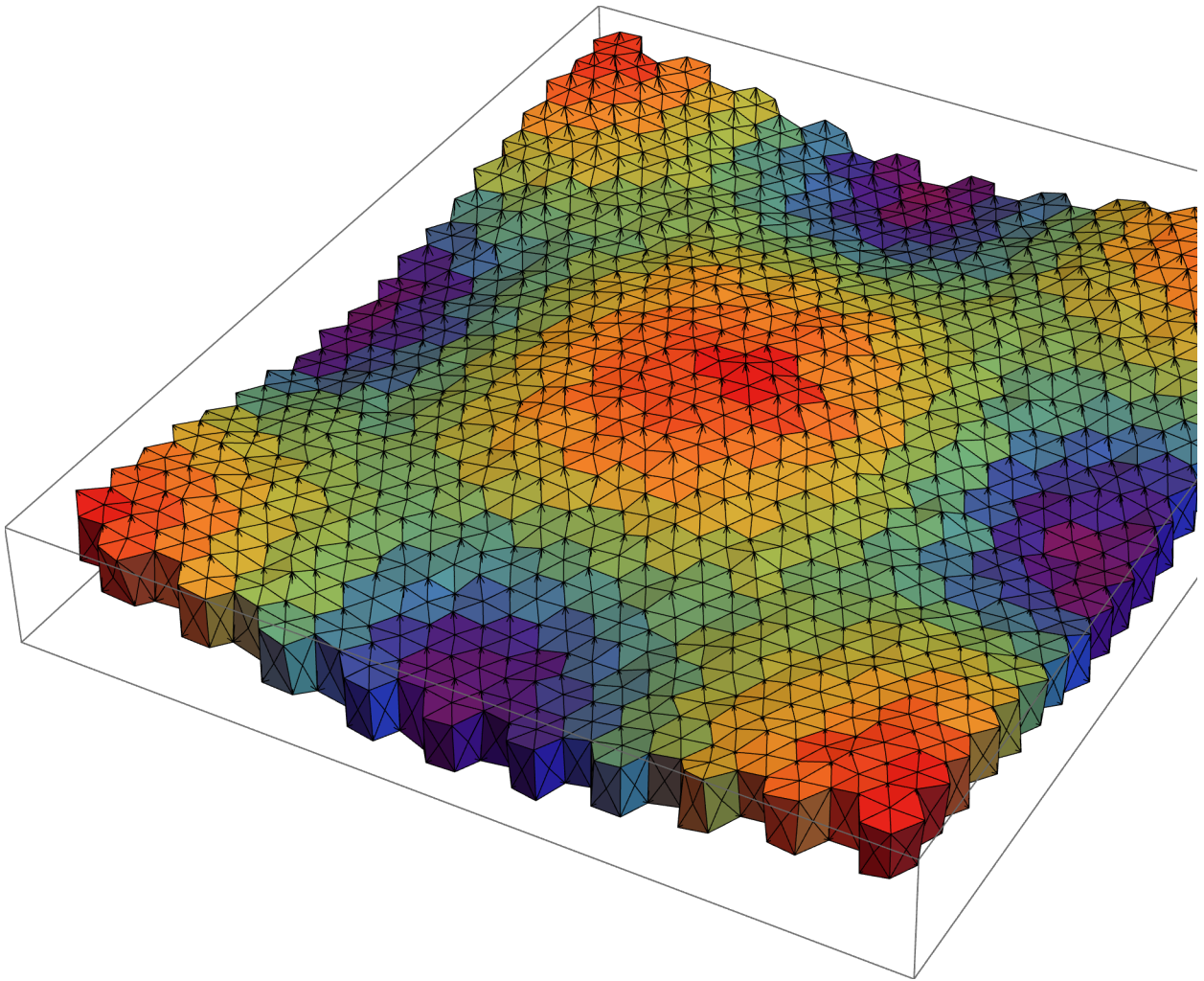
```
In[ ]:= res2~visualizeMesh~res4
```

Out[]:=



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

Out[]:=

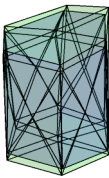


```

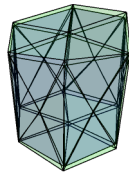
In[ ]:= CircleDot = MemberQ;
SeedRandom[1];
Grid[
  Partition[
    Function[ind,
      If[growingcellIndices ⊙ ind, Style[ind, {Bold, Red}], Style[ind, {Bold, Black}]] →
      With[{CVG = cellVertexGrouping[ind]},
        Show[Graphics3D[{Opacity[0.12], Blue,
          (Polyhedron@Flatten[triangulateToMesh[#], 1] &@ (Lookup[res2, #] & /@ CVG))}],
          Graphics3D[{Opacity[0.12], Green, (Polyhedron@Flatten[triangulateToMesh[#], 1] &@
            (Lookup[indToPtsAssoc, #] & /@ CVG))}],
          ImageSize → Tiny, Boxed → False]
      ]
    ] /@ Sort@RandomSample[Range[400], 30], 10]
]

```

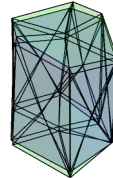
10 →



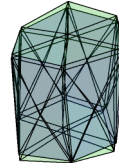
20 →



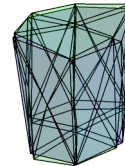
22 →



32 →



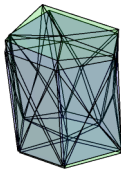
48 →



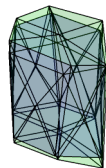
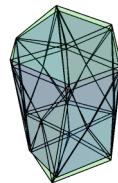
68 →



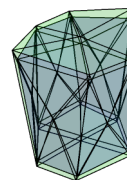
135 →



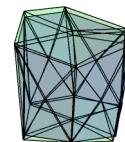
138 →

**163** →

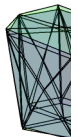
170 →



174 →

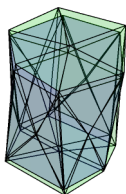


175 →

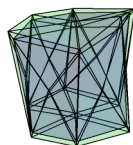
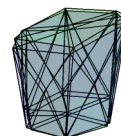


Out[]:=

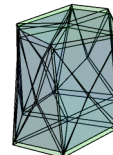
229 →



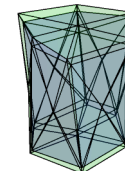
258 →

**300** →

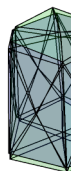
301 →



318 →



324 →



```

In[ ]:= $cellfaces = Map[Lookup[res2, #] &, cellVertexGrouping, {2}];
$mesh = Values[Polyhedron@Flatten[triangulateFaces@#, 1] & /@ $cellfaces];

```

```
In[ ]:= Mean[Volume@$mesh]
```

```
Out[ ]:= 0.469525
```