Module - computing Surface ∇

In[ ]:=
```
DumpGet["D:\\LocalData\\hashmial\\Github\\3D-Vertex-Model\\sanity
    checks for functions\\meshGen_noise.mx"];
```

In[ ]:=
```
yLim[[1]] = 0.;
edges = SetPrecision[edges, 10];
faceListCoords = SetPrecision[faceListCoords, 10];
(*convert faceListCoords into an association*)
indToPtsAssoc = SetPrecision[indToPtsAssoc, 10];
ptsToIndAssoc = KeyMap[SetPrecision[#, 10] &, ptsToIndAssoc];
xLim = SetPrecision[xLim, 10];
yLim = SetPrecision[yLim, 10];
faceListCoords = Map[Lookup[indToPtsAssoc, #] &, cellVertexGrouping, {2}];
```

In[ ]:=
```
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}, 10],
      {x_ /; x ≥ xlim2, y_ /; ylim1 < y < ylim2, z_} ⧴
       SetPrecision[{x - xlim2, y, z}, 10],
      {x_ /; xlim1 < x < xlim2, y_ /; y ≤ ylim1, z_} ⧴
       SetPrecision[{x, y + ylim2, z}, 10],
      {x_ /; x ≤ xlim1, y_ /; y ≤ ylim1, z_} ⧴
       SetPrecision[{x + xlim2, y + ylim2, z}, 10],
      {x_ /; x ≤ xlim1, y_ /; ylim1 < y < ylim2, z_} ⧴
       SetPrecision[{x + xlim2, y, z}, 10],
      {x_ /; x ≤ xlim1, y_ /; y ≥ ylim2, z_} ⧴
       SetPrecision[{x + xlim2, y - ylim2, z}, 10],
      {x_ /; xlim1 < x < xlim2, y_ /; y ≥ ylim2, z_} ⧴
       SetPrecision[{x, y - ylim2, z}, 10],
      {x_ /; x ≥ xlim2, y_ /; y ≥ ylim2, z_} ⧴ SetPrecision[{x - xlim2, y - ylim2, z}, 10]
     }];

  transformRules = Dispatch[{
      {x_ /; x ≥ xlim2, y_ /; y ≤ ylim1, _} ⧴ SetPrecision[{-xlim2, ylim2, 0}, 10],
      {x_ /; x ≥ xlim2, y_ /; ylim1 < y < ylim2, _} ⧴ SetPrecision[{-xlim2, 0, 0}, 10],
      {x_ /; xlim1 < x < xlim2, y_ /; y ≤ ylim1, _} ⧴ SetPrecision[{0, ylim2, 0}, 10],
      {x_ /; x ≤ xlim1, y_ /; y ≤ ylim1, _} ⧴ SetPrecision[{xlim2, ylim2, 0}, 10],
      {x_ /; x ≤ xlim1, y_ /; ylim1 < y < ylim2, _} ⧴ SetPrecision[{xlim2, 0, 0}, 10],
      {x_ /; x ≤ xlim1, y_ /; y ≥ ylim2, _} ⧴ SetPrecision[{xlim2, -ylim2, 0}, 10],
      {x_ /; xlim1 < x < xlim2, y_ /; y ≥ ylim2, _} ⧴ SetPrecision[{0, -ylim2, 0}, 10],
      {x_ /; x ≥ xlim2, y_ /; y ≥ ylim2, _} ⧴ SetPrecision[{-xlim2, -ylim2, 0}, 10],
      {___Real} ⧴ SetPrecision[{0, 0, 0}, 10]}];
 ];
```

In[ ]:=
```
origcellOrient = <|MapIndexed[First[#2] → #1 &, faceListCoords]|>;
boundaryCells = With[{ylim1 = yLim[[1]], ylim2 = yLim[[2]], xlim2 = xLim[[2]]},
   Union[First /@ Position[origcellOrient,
        {x_ /; x ≥ xlim2, __} | {x_ /; x ≤ 0, __} |
         {_, y_ /; y ≥ ylim2, _} | {_, y_ /; y ≤ ylim1, _}] /. Key[x_] ⧴ x]
  ];
wrappedMat = AssociationThread[
   Keys[cellVertexGrouping] → Map[Lookup[indToPtsAssoc, #] /. periodicRules &,
     Lookup[cellVertexGrouping, Keys[cellVertexGrouping]], {2}]];
```

In[ ]:=
```
meanTri = Compile[{{faces, _Real, 2}},
  Mean@faces,
  CompilationTarget → "C", RuntimeAttributes → {Listable},
  Parallelization → True
 ]
```

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

In[ ]:=
```
Clear[triNormal];
triNormal = Compile[{{ls, _Real, 2}},
  Block[{res},
   res = Partition[ls, 2, 1];
   Cross[res[[1, 1]] - res[[1, 2]], res[[2, 1]] - res[[2, 2]]]
  ], CompilationTarget → "C", RuntimeAttributes → {Listable}
 ]
```

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

In[ ]:=
```
Clear[meanFaces, triangulateToMesh];
meanFaces = Compile[{{faces, _Real, 2}},
  Block[{facepart, edgelen, mean},
   facepart = Partition[faces, 2, 1];
   AppendTo[facepart, {facepart[[-1, -1]], faces[[1]]}];
   edgelen = Table[Norm[SetPrecision[First@i - Last@i, 10]], {i, facepart}];
   mean = Total[edgelen * (Mean /@ facepart)] / Total[edgelen];
   mean],
  RuntimeAttributes → {Listable}, CompilationTarget → "C",
  CompilationOptions → {"InlineExternalDefinitions" → True}
 ]

triangulateToMesh[faces_] := Block[{mf, partfaces},
  mf = SetPrecision[meanFaces@faces, 10];
  partfaces = Partition[#, 2, 1, 1] & /@ faces;
  MapThread[
   If[Length[#] ≠ 3,
     Function[x, Join[x, {#2}]] /@ #1,
     {#[[All, 1]]}
    ] &, {partfaces, mf}]
 ]
```

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

In[ ]:=
```
Clear@cellCentroids;
cellCentroids[polyhedCentAssoc_, keystopo_, shiftvec_] :=
  Block[{assoc = <||>, regcent, counter},
   AssociationThread[Keys@keystopo →
     KeyValueMap[
      Function[{key, cellassoc},
       If[KeyFreeQ[shiftvec, key],
        Lookup[polyhedCentAssoc, cellassoc],
        If[KeyFreeQ[shiftvec[key], #],
           regcent = polyhedCentAssoc[#],
           regcent = polyhedCentAssoc[#] + shiftvec[key][#];
           regcent
          ] & /@ cellassoc
       ]
      ], keystopo]
   ]
  ];
```

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

```
ClearAll@getLocalTopology;
getLocalTopology[ptsToIndAssoc_, indToPtsAssoc_, vertexToCell_,
    cellVertexGrouping_, wrappedMat_, faceListCoords_][vertices_] :=
  Block[{localtopology = <||>, wrappedcellList = {}, vertcellconns,
    localcellunion, v, wrappedcellpos, vertcs = vertices, rl1, rl2,
    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]),
        (*Print["vertex inside bounds"];*)
        v = vert;
        With[{x = v[[1]], y = v[[2]]}, boundsCheck =
           (x == xLim[[1]] || x == xLim[[2]] || y == yLim[[1]] || y == yLim[[2]])];

        extractcellkeys = If[boundsCheck,
          {rl1, rl2} = {v, v /. periodicRules};
          rules = Block[{x$},
            With[{r = rl1, s = rl2},
             DeleteDuplicates[
              HoldPattern[SameQ[x$, r]] || HoldPattern[SameQ[x$, s]]]
            ]
           ];
          Position @@ With[{rule = rules},
            Hold[wrappedMat, x_ /; ReleaseHold@rule, {3}]
           ],
          Position[wrappedMat, x_ /; SameQ[x, v], {3}]
         ];
        (* find cell indices that are attached to the vertex in wrappedMat *)
        wrappedcellpos = DeleteDuplicatesBy[
          Cases[extractcellkeys,
           {Key[p : Except[Alternatives @@ Join[localcellunion,
                  Flatten@wrappedcellList]]], y__} ⧴ {p, y}],
          First];
        (*wrappedcellpos = wrappedcellpos/.
           {Alternatives@@Flatten[wrappedcellList],__} ⧴ Sequence[];*)
```

```
(* if a wrapped cell has not been considered earlier (i.e. is new)
 then we translate it to the position of the vertex *)
If[wrappedcellpos ≠ {},
 If[vertexQ,
  transVector = SetPrecision[(v - Extract[$faceListCoords, Replace[#,
          {p_, q__} :→ {Key[p], q}, {1}]]) & /@ wrappedcellpos, 10],
   (* call to function is enquiring an edge and not a vertex*)
  transVector =
   SetPrecision[(v - Extract[$faceListCoords, #]) & /@ wrappedcellpos, 10]
 ];
 wrappedcellCoords = MapThread[#1 → Map[Function[x,
       SetPrecision[x + #2, 10]], $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, 10];
(* ------------- CORE B ------------- *)
(* find which cells the
 shifted vertex is a part of in the wrapped matrix *)
wrappedcells = Complement[
  Union@Cases[Position[wrappedMat, x_ /;
       SameQ[x, shiftedPt] || SameQ[x, vertOutofBounds], {3}],
     x_Key :→ Sequence @@ x, {2}] /. Alternatives @@
     localcellunion → Sequence[],
  Flatten@wrappedcellList];

(*forming local topology now that we know the wrapped cells *)
If[wrappedcells ≠ {},
 AppendTo[wrappedcellList, Flatten@wrappedcells];
 wrappedcellCoords = AssociationThread[wrappedcells,
   Map[Lookup[indToPtsAssoc, #] &,
    cellVertexGrouping[#] & /@ wrappedcells, {2}]];
 With[{opt = (vertOutofBounds /. periodicRules) | vertOutofBounds},
  Block[{pos, vertref, transvec},
    Do[
     With[{cellcoords = wrappedcellCoords[cell]},
      pos = FirstPosition[cellcoords /. periodicRules, opt];
      If[Head[pos] === Missing,
       pos = FirstPosition[
          Chop[cellcoords /. periodicRules, 10^-6], Chop[opt, 10^-6]];
      ];
      vertref = Extract[cellcoords, pos];
      transvec = SetPrecision[vertOutofBounds - vertref, 10];
      AppendTo[transvecList, transvec];
      AppendTo[localtopology,
       cell → Map[SetPrecision[# + transvec, 10] &, cellcoords, {2}]];
     ], {cell, wrappedcells}]
```

```
          ];
         ];
        ];
        (* to detect wrapped cells not detected by CORE B*)
        (* ------------- CORE C ------------- *)
        Block[{pos, celllocs, ls, transvec, assoc, tvecLs = {}, ckey},
         ls = Union@Flatten@Join[cellsconnected, wrappedcells];
         If[Length[ls] ≠ 3,
          pos = Position[faceListCoords, x_ /; SameQ[x, shiftedPt], {3}];
          celllocs = DeleteDuplicatesBy[Cases[pos, Except[{Key[Alternatives @@ ls],
                __}]], First] /. {Key[x_], z__} ⧴ {Key[x], {z}};
          If[celllocs ≠ {},
           celllocs = Transpose@celllocs;
           assoc = <|
             MapThread[
              (transvec = SetPrecision[vertOutofBounds -
                   Extract[faceListCoords[Sequence @@ #1], #2], 10];
               ckey = Identity @@ #1;
               AppendTo[tvecLs, transvec];
               ckey → Map[SetPrecision[Lookup[indToPtsAssoc, #] + transvec,
                   10] &, 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}
 ];
```

Launch Kernels

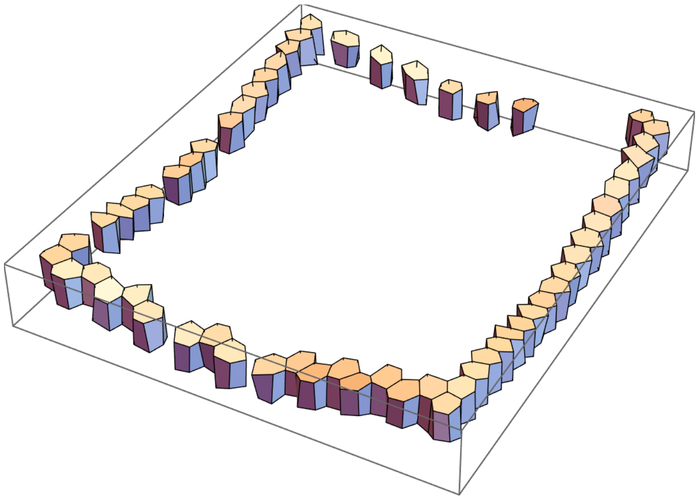
In[ ]:=
```
LaunchKernels[]
```

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

prerequisite run

In[∘]:=
```
Graphics3D[Polygon /@ (faceListCoords /@ boundaryCells)]
```

Out[∘]=



In[∘]:=
```
(*missing boundary cells need to be found *)
```

In[∘]:=
```
bcells = KeyTake[faceListCoords, boundaryCells];
```

In[∘]:=
```
Length@boundaryCells
```

Out[∘]=
```
60
```

In[∘]:=
```
keyLs = Union@(Flatten@Lookup[vertexToCell,
         Lookup[ptsToIndAssoc,
          With[{ylim1 = yLim[[1]],
             ylim2 = yLim[[2]], xlim1 = xLim[[1]], xlim2 = xLim[[2]]},
            DeleteDuplicates@Cases[bcells,
              {x_ /; x ≥ xlim2, __} | {x_ /; x ≤ xlim1, __} |
               {_, y_ /; y ≥ ylim2, _} | {_, y_ /; y ≤ ylim1, _}, {3}]
           ] /. periodicRules
         ]
        ] ~ Join ~ boundaryCells);
```

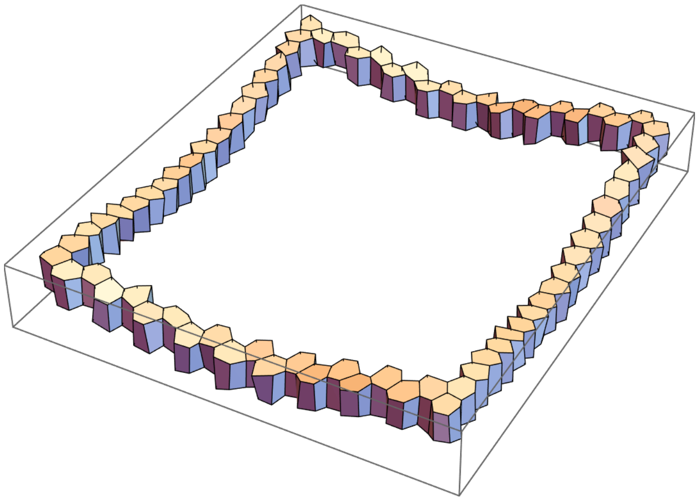
In[∘]:=
```
Length[keyLs] - Length[boundaryCells]
```

Out[∘]=
```
16
```

In[∘]:=
```
border = faceListCoords /@ keyLs;
```

In[∘]:=
```
Graphics3D[{Polygon /@ border}, ImageSize → Medium]
```

Out[∘]=



In[∘]:=
```
wrappedMatC = KeyTake[wrappedMat, keyLs];
```

In[∘]:=
```
vertKeys = Keys@indToPtsAssoc;
```

In[∘]:=
```
(
  topo = <|# → (getLocalTopology[ptsToIndAssoc, indToPtsAssoc,
            vertexToCell, cellVertexGrouping, wrappedMatC, faceListCoords][
           indToPtsAssoc[#]] // First) & /@ vertKeys
      |>;
 ) // AbsoluteTiming
```

Out[∘]=
```
{1.02689, Null}
```

finding triangles connected to a vertex

In[∘]:=
```
(trimesh = Map[triangulateToMesh, topo, {2}]); // AbsoluteTiming
```

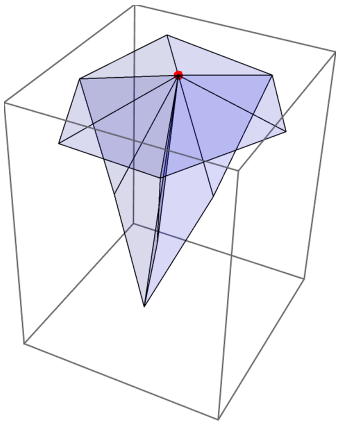
Out[∘]=
```
{1.76328, Null}
```

In[∘]:=
```
examplevertToTri =
   GroupBy[Flatten[Values@trimesh[#], 2], MemberQ[indToPtsAssoc[#]]][True] &[
    1]; // AbsoluteTiming
```

Out[∘]=
```
{0.0004543, Null}
```

In[∘]:=
```
(examplevertToTri =
    GroupBy[Flatten[Values@trimesh[#], 2], MemberQ[indToPtsAssoc[#]]][True];
   Graphics3D[{{Opacity[0.15], Blue, Triangle /@ examplevertToTri},
     Red, PointSize[0.03], Point@indToPtsAssoc[#]},
    ImageSize → Small]
  ) &[RandomInteger[Max@Keys@indToPtsAssoc]]
```

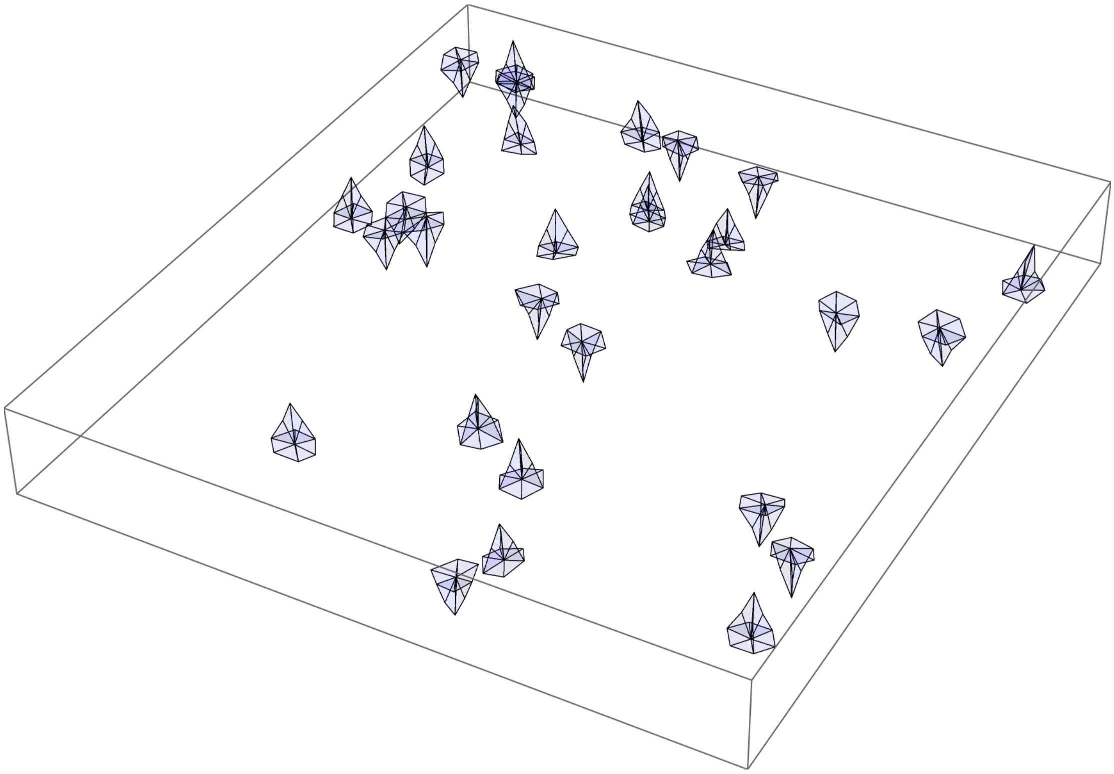
Out[∘]=



In[∘]:=
```
(
  associatedtri = With[{ItoPA = indToPtsAssoc, tmesh = trimesh},
     AssociationThread[vertKeys, Function[vert,
        With[{pt = Chop@ItoPA[vert]}, <|GroupBy[
              Flatten[#, 1], MemberQ[#, x_ /; Chop[x] === pt] &][True] & /@ tmesh[vert]|>
        ]] /@ vertKeys]
    ];
 ) // AbsoluteTiming
```

Out[∘]= {0.985904, Null}

In[]:=
```
SeedRandom[3];
Graphics3D[{Opacity[0.1], Blue, Triangle /@
   Flatten[Values@Values@RandomSample[associatedtri, 30], 2]}, ImageSize → Large]
```

Out[]=


In[]:=
```
(centTri = <|# → meanTri[Values[associatedtri@#]] & /@ Keys@indToPtsAssoc|>;) //
  AbsoluteTiming
```

Out[]= {0.305219, Null}

In[]:=
```
centTri = SetPrecision[#, 10] & /@ centTri;
```

In[]:=
```
(normals = Map[SetPrecision[#, 10] &, triNormal@Values@# & /@ associatedtri]); //
  AbsoluteTiming
```

Out[]= {0.395068, Null}

In[]:=
```
(normNormals = Map[Normalize, normals, {3}];) // AbsoluteTiming
```

Out[]= {0.0915531, Null}

In[]:=
```
(triangulatedmesh = triangulateToMesh /@ faceListCoords); // AbsoluteTiming
(polyhedra = Polyhedron@* (Flatten[#, 1] &) /@ triangulatedmesh;) // AbsoluteTiming
```

Out[]= {0.137013, Null}

Out[]= {0.0019697, Null}

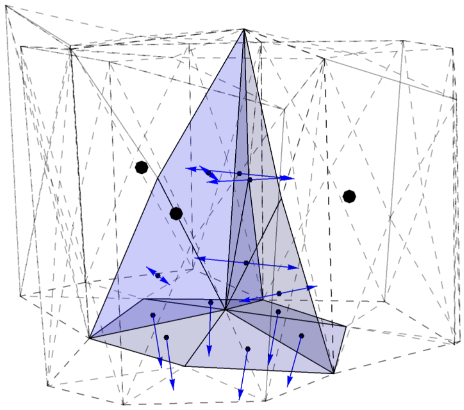
In[]:=
```
(polyhedcent = RegionCentroid /@ polyhedra); // AbsoluteTiming
```

Out[]= {3.78279, Null}

```
In[•]:= (
          topoF = <|# → (getLocalTopology[
                   ptsToIndAssoc, indToPtsAssoc, vertexToCell, cellVertexGrouping,
                   wrappedMatC, faceListCoords][indToPtsAssoc[#]]) & /@ vertKeys
             |>;
         ) // AbsoluteTiming

Out[•]= {1.04721, Null}

In[•]:= (keyslocaltopoF = Keys@*First /@ topoF); // AbsoluteTiming

Out[•]= {0.003086, Null}

In[•]:= (shiftVecAssoc = Association /@ Map[Apply[Rule],
              Thread /@ Select[(#[[2 ;; 3]]) & /@ topoF, # ≠ {{}, {}} &], {2}]); // AbsoluteTiming

Out[•]= {0.0048144, Null}

In[•]:= (cellcentroids = cellCentroids[polyhedcent, keyslocaltopoF, shiftVecAssoc]);

In[•]:= (signednormals = AssociationThread[Keys@indToPtsAssoc,
             Map[
              MapThread[
                #2 Sign@MapThread[Function[{x, y}, (y - #1).x], {#2, #3}] &,
                {cellcentroids[#], normNormals[#], centTri[#]}] &, Keys@indToPtsAssoc]
            ]
          ); // AbsoluteTiming

Out[•]= {0.165886, Null}

In[•]:= (signs = AssociationThread[Keys@indToPtsAssoc,
             Map[
              MapThread[
                Sign@MapThread[Function[{x, y}, (y - #1).x], {#2, #3}] &,
                {cellcentroids[#], normNormals[#], centTri[#]}] &, Keys@indToPtsAssoc]
            ]
          ); // AbsoluteTiming

Out[•]= {0.149697, Null}
```

In[•]:=
```
Function[key,
  Graphics3D[{{Opacity[0.2], Blue,
      Triangle /@ Flatten[Values@associatedtri[key], 1]}, Point /@ centTri[key],
     Black, PointSize[0.02], Point@cellcentroids[key], Blue, Arrowheads[Small],
     MapThread[Arrow[{#2, #2 + 0.2 #1}] &,
      {Flatten[signednormals[key], 1], Flatten[centTri[[key]], 1]}],
     {Opacity[0.4], Black, Dashed, Line /@ Flatten[Values@trimesh[key], 2]}
   }, ImageSize → Medium, Boxed → False]
 ][7]
```

Out[•]=



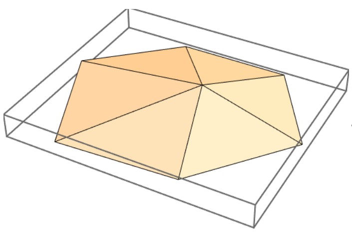
make sets of open/closed triangles

In[•]:=
```
opencloseTri = Flatten[Values@#, 1] & /@ associatedtri;
```

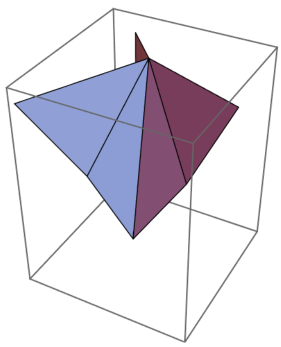
In[•]:=
```
opencloseTri =
  MapThread[MapAt[Function[coords, {coords[[1]], coords[[3]], coords[[2]]}],
     #1, Position[Flatten[#2, 1], -1]] &, {opencloseTri, signs}];
```

In[•]:=
```
Graphics3D /@ Map[Triangle,
  GroupBy[GatherBy[opencloseTri[1], Intersection], Length, Flatten[#, 1] &], {2}]
```

Out[•]= ⟨| 1 →



, 2 →



|⟩

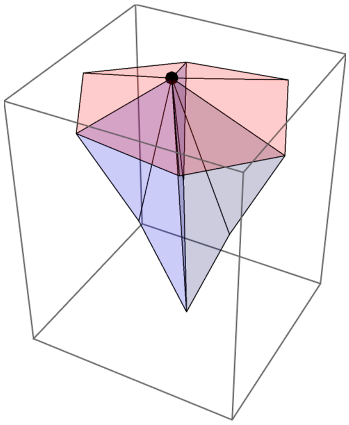
In[•]:=
```
triDistAssoc = Block[{trianglemembers},
   Map[
    (trianglemembers = #;
      GroupBy[GatherBy[trianglemembers, Intersection], Length, Flatten[#, 1] &]) &,
    opencloseTri]
  ];
```

In[∘]:=
```
pointind = 5;
```

In[∘]:=
```
{opentriExample, closedtriExample} =
  {triDistAssoc[pointind][1], triDistAssoc[pointind][2]};
```

In[∘]:=
```
Graphics3D[{{Opacity[0.2], Red,
   Map[Triangle][opentriExample], Blue, Map[Triangle][closedtriExample]},
  {Black, PointSize[0.04], Point@indToPtsAssoc[pointind]}}, ImageSize → Small]
```

Out[∘]=



associate normals with triangles

In[∘]:=
```
vertTriNormalpairings = <|
   # → <|Thread[opencloseTri[#] → Flatten[signednormals@#, 1]]|> & /@ vertKeys|>;
```

To associate the open/closed triangles with their respective normals we simply need to perform a lookup in the association for (vertex1,vertex2,vertex3) - a triangle face - and its normal.

In[∘]:=
```
normalsO = Lookup[vertTriNormalpairings[pointind], opentriExample];
```

In[∘]:=
```
normalsC = Lookup[vertTriNormalpairings[pointind], closedtriExample];
```

In[∘]:=
```
centLs = {};
arrow = Flatten@Map[Module[{tri, normal, cent, tricent},
       tri = Triangle[#[[2]]];
       cent = Region`Mesh`MeshCentroid[DiscretizeRegion@tri];
       AppendTo[centLs, cent];
       Arrow[{cent, cent + 0.15 #[[1]]}]
      ] &,
     {Thread[{normalsO, opentriExample}], Thread[{normalsC, closedtriExample}]}, {2}];
```

In[∘]:=
```
point = indToPtsAssoc[pointind];
```

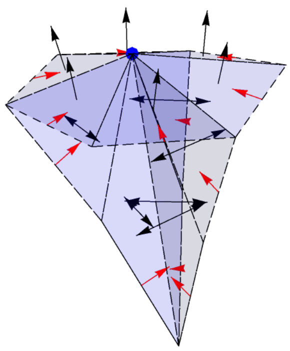
In[∘]:=
```
{crossprod, midpt} =
  Flatten[#, 1] & /@ Transpose[#, {2, 1}] &@ (Function[x, Transpose@MapThread[
        Block[{ptTri = #1, source = point, normal = #2, u2, u1, cross, pos},
          pos = First @@ Position[ptTri, source];
          Which[pos == 1,
           {u1, u2} = {ptTri[[2]], ptTri[[-1]]},
           pos == 2,
           {u2, u1} = {ptTri[[1]], ptTri[[-1]]},
           pos == 3,
           {u2, u1} = {ptTri[[2]], ptTri[[1]]}
          ];
          cross = Cross[normal, u2 - u1];
          {0.5 cross, (u2 + u1) / 2}
         ] &, x]] /@ {{opentriExample, normalsO}, {closedtriExample, normalsC}});
```

In[∘]:=
```
centLsPartition = TakeDrop[centLs, Length@opentriExample];
```

In[∘]:=
```
arrowtosource = Flatten@Map[
    Module[{cent = #[[1]], vec = #[[2]]},
      Arrow[{cent, cent + 0.4 vec}]
     ] &, Thread[{midpt, crossprod}]];
```

In[∘]:=
```
plt2 = Graphics3D[{{{Blue, Opacity[0.15], EdgeForm[Dashed],
      Triangle /@ opentriExample, Triangle /@ closedtriExample},
     {Blue, PointSize[0.04], Point@point}, {Arrowheads[Small], arrow},
     {Red, Arrowheads[Small], arrowtosource}}},
  ImageSize → Small, Boxed → False]
```

Out[∘]=



surface gradient

In[]:=
```
{openSCont, closedSCont} = Function[x, Total@MapThread[
      Block[{ptTri = #1, source = point, normal = #2, u2, u1, cross, pos},
        pos = First @@ Position[ptTri, source];
        Which[pos == 1,
         {u1, u2} = {ptTri[[2]], ptTri[[-1]]},
         pos == 2,
         {u2, u1} = {ptTri[[1]], ptTri[[-1]]},
         pos == 3,
         {u2, u1} = {ptTri[[2]], ptTri[[1]]}
        ];
        cross = Cross[normal, u2 - u1];
        1 / 2 cross
       ] &, x]] /@ {{opentriExample, normalsO}, {closedtriExample, normalsC}}
```

Out[]=
```
{{0.00809089, 0.03573710, 0.354497676}, {-0.218497033, 0.238211646, 1.335478151}}
```

In[]:=
```
Quiet[ϵco =.]; Quiet[ϵcc =.];
```

In[]:=
```
ϵco openSCont + ϵcc closedSCont
```

Out[]=
$\{-0.218497033\, \epsilon_{cc} + 0.00809089\, \epsilon_{co},$
$0.238211646\, \epsilon_{cc} + 0.03573710\, \epsilon_{co},\ 1.335478151\, \epsilon_{cc} + 0.354497676\, \epsilon_{co}\}$