Module - computing Surface ∇

In[1]:=
```
DumpGet["C:\\Users\\aliha\\Desktop\\wolfram-vertex-3D\\PREVIOUS
    CODE - slow heuns\\meshGen_noise.mx"];
```

In[2]:=
```
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[10]:=
```
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 < 0., y_ /; y ≤ ylim1, z_} :→ SetPrecision[{x + xlim2, y + ylim2, z}, 10],
     {x_ /; x < 0., y_ /; ylim1 < y < ylim2, z_} :→ SetPrecision[{x + xlim2, y, z}, 10],
     {x_ /; x < 0., y_ /; y > ylim2, z_} :→ SetPrecision[{x + xlim2, y - ylim2, z}, 10],
     {x_ /; 0. < 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 < 0, y_ /; y ≤ ylim1, _} :→ SetPrecision[{xlim2, ylim2, 0}, 10],
     {x_ /; x < 0, y_ /; ylim1 < y < ylim2, _} :→ SetPrecision[{xlim2, 0, 0}, 10],
     {x_ /; x < 0, y_ /; y > ylim2, _} :→ SetPrecision[{xlim2, -ylim2, 0}, 10],
     {x_ /; 0 < 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[11]:=
```
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[14]:=

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

Out[14]= CompiledFunction[Argument count: 1 Argument types: {{_Real, 2}}]

In[15]:=

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

In[17]:=

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

In[20]:=
```
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[22]:=
```
𝒟 = Rectangle[{First@xLim, First@yLim}, {Last@xLim, Last@yLim}];
```

In[23]:=
```
getLocalTopology[ptsToIndAssoc_, indToPtsAssoc_, vertexToCell_,
    cellVertexGrouping_, wrappedMat_, faceListCoords_][vertices_] :=
  Module[{localtopology = <||>, wrappedcellList = {}, vertcellconns,
    localcellunion, vertInBounds, v, wrappedcellpos, vertcs = vertices,
    transVector, wrappedcellCoords, wrappedcells, vertOutofBounds,
    shiftedPt, transvecList = {}, $faceListCoords = Values@faceListCoords,
    vertexQ},
   vertexQ = MatchQ[vertices, {__?NumberQ}];
   If[vertexQ,
    vertcellconns =
     AssociationThread[{#}, {vertexToCell[ptsToIndAssoc[#]]}] &@vertices;
    vertcs = {vertices};
    localcellunion = Flatten[Values@vertcellconns],
    (* this will yield vertex → cell indices connected in the local mesh *)
    vertcellconns =
     AssociationThread[#, Lookup[vertexToCell, Lookup[ptsToIndAssoc, #]]] &@vertices;
    localcellunion = Union@Flatten[Values@vertcellconns];
   ];
   (* condition to be an internal
    edge: both vertices should have 3 or more neighbours *)
   (*Print["All topology known"];*)
   (* the cells in the local mesh define the entire network topology →
    no wrapping required *)
   (* else cells need to be wrapped because other cells are
     connected to the vertices → periodic boundary conditions *)
   With[{vert = #},
      If[((𝒟~RegionMember~Most[vert]) &&
          !(vert[[1]] == xLim[[2]] || vert[[2]] == yLim[[2]])),
        (* the vertex has less than 3 neighbouring cells but
```

```
    the vertex is within bounds *)
   (*Print["vertex inside bounds with fewer than 3 cells"];*)
   v = vertInBounds = vert;
   (* find cell indices that are attached to the vertex in wrappedMat *)
   wrappedcellpos = DeleteDuplicatesBy[
     Cases[Position[wrappedMat, x_ /; SameQ[x, v], {3}],
      {Key[p : Except[Alternatives @@
             Join[localcellunion, Flatten@wrappedcellList]]], y__} :> {p, y}],
     First];
   (*wrappedcellpos = wrappedcellpos/.
      {Alternatives@@Flatten[wrappedcellList],__} :> Sequence[];*)
   (* if a wrapped cell has not been considered earlier (i.e. is new)
    then we translate it to the position of the vertex *)
   If[wrappedcellpos ≠ {},
    If[vertexQ,
     transVector = SetPrecision[(v - Extract[$faceListCoords,
             Replace[#, {p_, q__} :> {Key[p], q}, {1}]]) & /@ wrappedcellpos, 10],
     (*the main function is enquiring an edge and not a vertex*)
     transVector =
      SetPrecision[(v - Extract[$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];
    (*local topology here only has wrapped cell *)
   ],
   (*Print["vertex out of bounds"];*)
   (* else vertex is out of bounds *)
   vertOutofBounds = vert;
   (* translate the vertex back into mesh *)
   transVector = vertOutofBounds /. transformRules;
   shiftedPt = SetPrecision[vertOutofBounds + transVector, 10];
   (* find which cells the vertex is a part of in the wrapped matrix *)
   wrappedcells = Complement[
     Union@Cases[Position[wrappedMat, x_ /; SameQ[x, shiftedPt], {3}],
        x_Key :> Sequence @@ x, {2}] /. Alternatives @@ localcellunion → Sequence[],
     Flatten@wrappedcellList];
   (*forming local topology now that we know the wrapped cells *)
   If[wrappedcells ≠ {},
    AppendTo[wrappedcellList, Flatten@wrappedcells];
    wrappedcellCoords = AssociationThread[wrappedcells,
      Map[Lookup[indToPtsAssoc, #] &, cellVertexGrouping[#] & /@ wrappedcells, {2}]
     ];
    With[{opt = (vertOutofBounds /. periodicRules)},
     Block[{pos, vertref, transvec},
       Do[
```

```
              With[{cellcoords = wrappedcellCoords[cell]},
               pos = FirstPosition[cellcoords /. periodicRules, opt];
               vertref = Extract[cellcoords, pos];
               transvec = SetPrecision[vertOutofBounds - vertref, 10];
               AppendTo[transvecList, transvec];
               AppendTo[localtopology, cell →
                 Map[SetPrecision[# + transvec, 10] &, cellcoords, {2}]];
              ], {cell, wrappedcells}]
            ];
          ];
         ];
        ];
      ] & /@ vertcs;
    If[localcellunion ≠ {},
     AppendTo[localtopology,
      Thread[localcellunion →
        Map[Lookup[indToPtsAssoc, #] &, cellVertexGrouping /@ localcellunion, {2}]]
     ]
    ];
    transvecList = Which[
      MatchQ[transvecList, {{{__?NumberQ}}}], First[transvecList],
      MatchQ[transvecList, {{__?NumberQ} ..}], transvecList,
      True, transvecList //. {x___, {p : {__?NumberQ} ..}, y___} ⧴ {x, p, y}
     ];
    {localtopology, Flatten@wrappedcellList, transvecList}
   ];
```

Launch Kernels

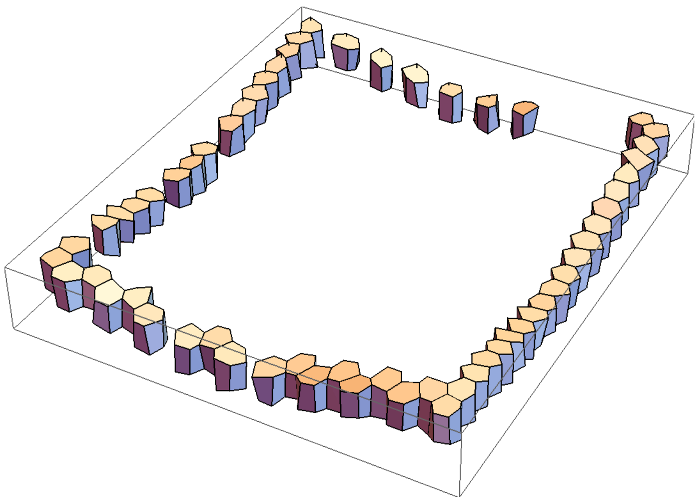
In[24]:=
```
LaunchKernels[]
```

Out[24]=
```
{KernelObject[ Name: local KernelID: 1 ], KernelObject[ Name: local KernelID: 2 ],
 KernelObject[ Name: local KernelID: 3 ], KernelObject[ Name: local KernelID: 4 ]}
```

prerequisite run

In[25]:=
```
Graphics3D[Polygon /@ (faceListCoords /@ boundaryCells)]
```

Out[25]=



In[26]:=
```
(*missing boundary cells need to be found *)
```

In[27]:=
```
bcells = KeyTake[faceListCoords, boundaryCells];
```

In[28]:=
```
Length@boundaryCells
```

Out[28]= 60

In[29]:=
```
keyLs = Union@(Flatten@Lookup[vertexToCell,
        Lookup[ptsToIndAssoc,
         With[{ylim1 = yLim[[1]], ylim2 = yLim[[2]], xlim2 = xLim[[2]]},
           DeleteDuplicates@Cases[bcells,
              {x_ /; x ≥ xlim2, __} | {x_ /; x < 0, __} |
               {_, y_ /; y > ylim2, _} | {_, y_ /; y ≤ ylim1, _}, {3}]
          ] /. periodicRules
        ]
       ] ~Join~ boundaryCells);
```

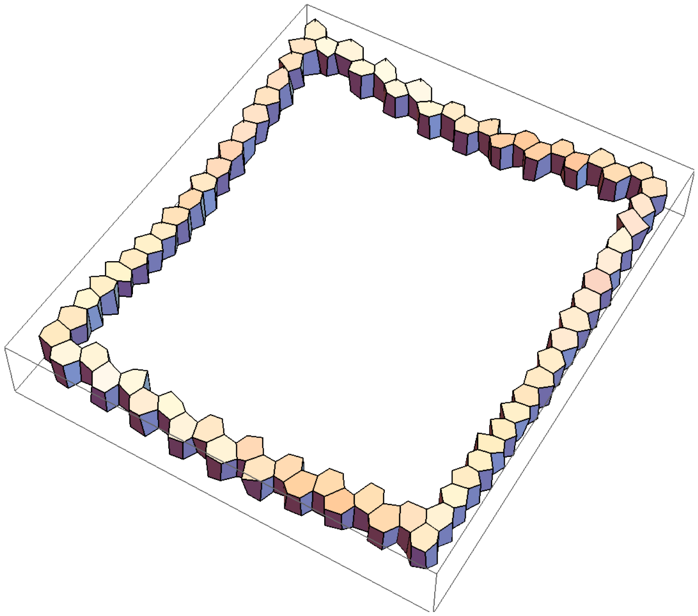
In[30]:=
```
Length[keyLs] - Length[boundaryCells]
```

Out[30]= 16

In[31]:=
```
border = faceListCoords /@ keyLs;
```

In[32]:=
```
Graphics3D[{Polygon /@ border}, ImageSize → Medium]
```

Out[32]=



In[33]:=
```
wrappedMatC = KeyTake[wrappedMat, keyLs];
```

In[34]:=
```
vertKeys = Keys@indToPtsAssoc;
```

In[35]:=
```
(
  topo = <|
     # → (getLocalTopology[ptsToIndAssoc, indToPtsAssoc, vertexToCell, cellVertexGrouping,
            wrappedMatC, faceListCoords][indToPtsAssoc[#]] // First) & /@ vertKeys
     |>;
 ) // AbsoluteTiming
```

Out[35]= {4.34903, Null}

finding triangles connected to a vertex

In[36]:=
```
(trimesh = Map[triangulateToMesh, topo, {2}]); // AbsoluteTiming
```

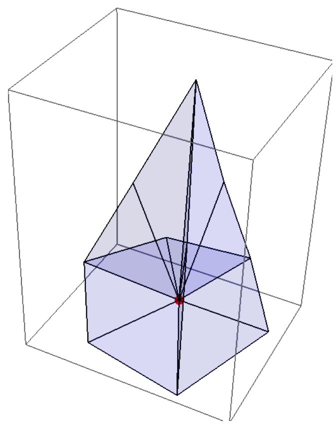
Out[36]= {2.09038, Null}

In[37]:=
```
examplevertToTri =
   GroupBy[Flatten[Values@trimesh[#], 2], MemberQ[indToPtsAssoc[#]]][True] &[
    1]; // AbsoluteTiming
```

Out[37]= {0.0003001, Null}

In[38]:=
```
(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[38]=

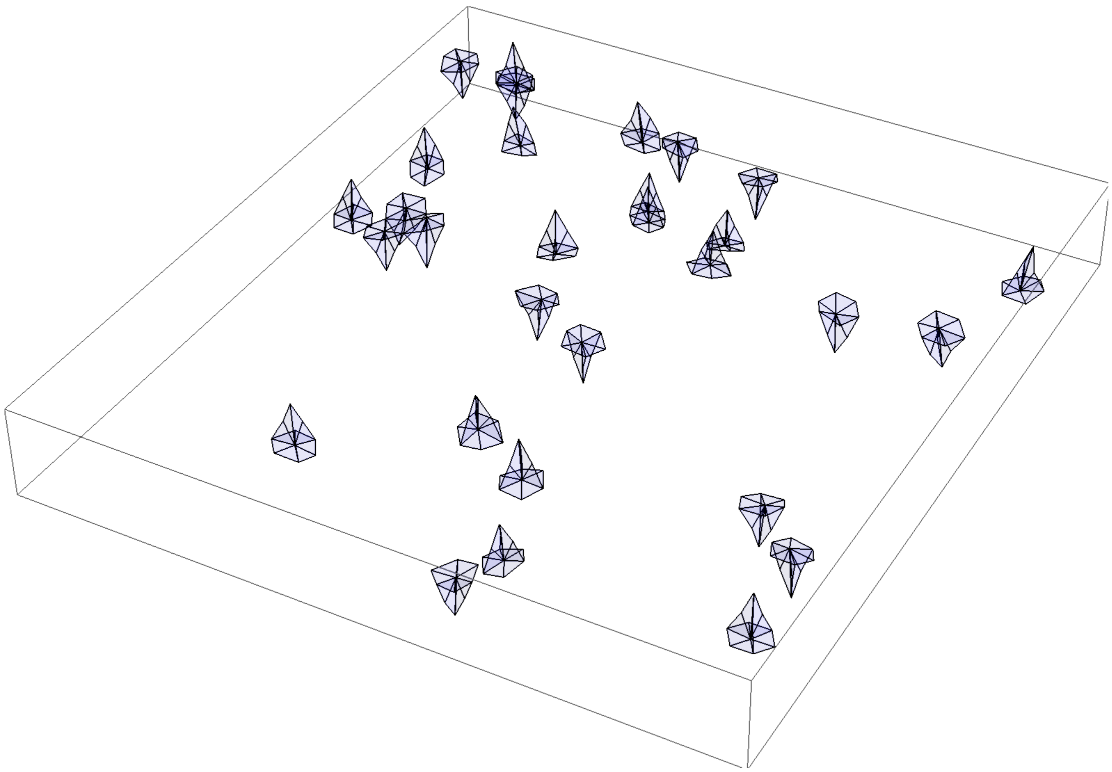


In[39]:=
```
(associatedtri = With[{ItoPA = indToPtsAssoc, tmesh = trimesh},
     AssociationThread[vertKeys, Function[vert, <|GroupBy[
             Flatten[#, 1], MemberQ[ItoPA[vert]]
            ][True] & /@ tmesh[vert]|>] /@ vertKeys]
    ];
 ) // AbsoluteTiming
```

Out[39]= {0.590441, Null}

In[73]:=
```
SeedRandom[3];
Graphics3D[{Opacity[0.1], Blue, Triangle /@
   Flatten[Values@Values@RandomSample[associatedtri, 30], 2]}, ImageSize → Large]
```

Out[74]=



In[42]:=
```
(centTri = <|# → meanTri[Values[associatedtri@#]] & /@ Keys@indToPtsAssoc|>;) //
 AbsoluteTiming
```

Out[42]= {0.397898, Null}

In[43]:=
```
centTri = SetPrecision[#, 10] & /@ centTri;
```

In[44]:=
```
(normals = Map[SetPrecision[#, 8] &, triNormal@Values@# & /@ associatedtri]); //
 AbsoluteTiming
```

Out[44]= {0.564806, Null}

In[45]:=
```
(normNormals = Map[Normalize, normals, {3}];) // AbsoluteTiming
```

Out[45]= {0.123284, Null}

In[46]:=
```
(triangulatedmesh = triangulateToMesh /@ faceListCoords); // AbsoluteTiming
(polyhedra = Polyhedron@* (Flatten[#, 1] &) /@ triangulatedmesh;) // AbsoluteTiming
```

Out[46]= {0.173156, Null}

Out[47]= {0.002182, Null}

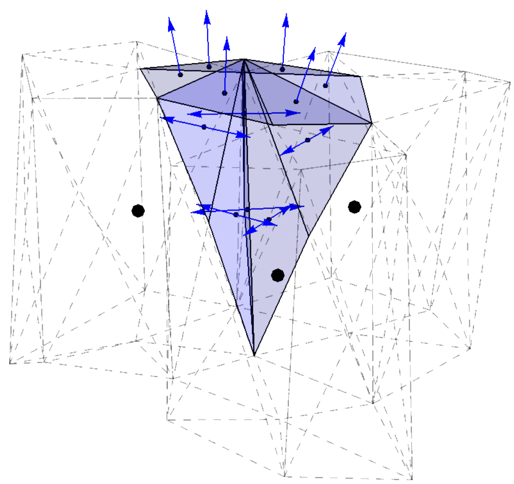
In[48]:=
```
(polyhedcent = RegionCentroid /@ polyhedra); // AbsoluteTiming
```

Out[48]= {4.41992, Null}

```
In[49]:= (
          topoF = <|
             # → (getLocalTopology[ptsToIndAssoc, indToPtsAssoc, vertexToCell, cellVertexGrouping,
                  wrappedMatC, faceListCoords][indToPtsAssoc[#]]) & /@ vertKeys
             |>;
         ) // AbsoluteTiming

Out[49]= {4.16146, Null}

In[50]:= (keyslocaltopoF = Keys@*First /@ topoF); // AbsoluteTiming

Out[50]= {0.004565, Null}

In[51]:= (shiftVecAssoc = Association /@ Map[Apply[Rule],
              Thread /@ Select[(#[[2 ;; 3]]) & /@ topoF, # ≠ {{}, {}} &], {2}]); // AbsoluteTiming

Out[51]= {0.0051678, Null}

In[52]:= (cellcentroids = cellCentroids[polyhedcent, keyslocaltopoF, shiftVecAssoc]);

In[53]:= (signednormals = AssociationThread[Keys@indToPtsAssoc,
             Map[
              MapThread[
                #2 Sign@MapThread[Function[{x, y}, (y - #1).x], {#2, #3}] &,
                {cellcentroids[#], normNormals[#], centTri[#]}] &, Keys@indToPtsAssoc]
            ]
          ); // AbsoluteTiming

Out[53]= {0.21506, Null}
```

In[54]:=
```
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]
 ][5]
```

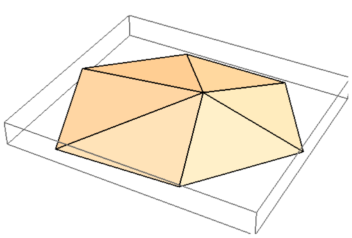
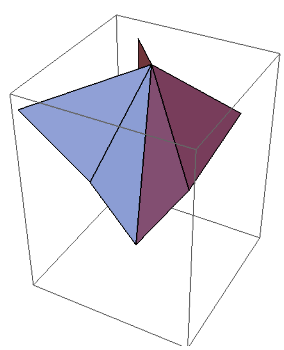
Out[54]=



make sets of open/closed triangles

In[55]:=
```
opencloseTri = Flatten[Values@#, 1] & /@ associatedtri;
```

In[56]:=
```
Graphics3D /@ Map[Triangle,
  GroupBy[GatherBy[opencloseTri[1], Intersection], Length, Flatten[#, 1] &], {2}]
```

Out[56]= ⟨| 1 →  , 2 →  |⟩

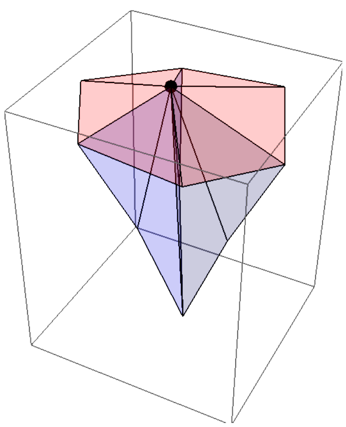
In[57]:=
```
triDistAssoc = Block[{trianglemembers},
   Map[
     (trianglemembers = #;
       GroupBy[GatherBy[trianglemembers, Intersection], Length, Flatten[#, 1] &]) &,
     opencloseTri]
  ];
```

In[58]:=
```
pointind = 5;
```

In[59]:=
```
{opentriExample, closedtriExample} =
  {triDistAssoc[pointind][1], triDistAssoc[pointind][2]};
```

In[60]:=
```
Graphics3D[{{Opacity[0.2], Red,
    Map[Triangle][opentriExample], Blue, Map[Triangle][closedtriExample]},
  {Black, PointSize[0.04], Point@indToPtsAssoc[pointind]}}, ImageSize → Small]
```

Out[60]=



associate normals with triangles

In[61]:=
```
vertTriNormalpairings = <|
    # → <|Thread[Flatten[Values@associatedtri[#], 1] → 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[62]:=
```
normalsO = Lookup[vertTriNormalpairings[pointind], opentriExample];
```

In[63]:=
```
normalsC = Lookup[vertTriNormalpairings[pointind], closedtriExample];
```

In[64]:=
```
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[66]:=
```
point = indToPtsAssoc[pointind];
```

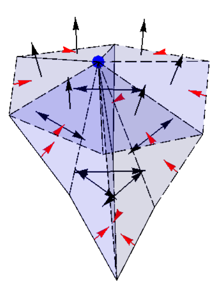
In[67]:=
```
{crossprod, midpt} =
  Flatten[#, 1] & /@ Transpose[#, {2, 1}] &@ (Function[x, Transpose@MapThread[
        Block[{ptTri = #1, source = point, normal = #2, target, facept, cross},
          If[First @@ Position[ptTri, source] == 1,
           {target, facept} = {ptTri[[2]], ptTri[[-1]]};
           cross = Cross[normal, facept - target],
           {target, facept} = {ptTri[[1]], ptTri[[-1]]};
           cross = Cross[normal, target - facept]
          ];
          {0.5 cross, (target + facept) / 2}
         ] &, x]] /@ {{opentriExample, normalsO}, {closedtriExample, normalsC}});
```

In[68]:=
```
centLsPartition = TakeDrop[centLs, Length@opentriExample];
```

In[69]:=
```
arrowtosource = Flatten@Map[
    Module[{cent = #[[1]], vec = #[[2]]},
      Arrow[{cent, cent + 0.3 vec}]
     ] &, Thread[{midpt, crossprod}]];
```

In[70]:=
```
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[70]=



surface gradient

In[71]:=
```
{openSCont, closedSCont} = Function[x, Total@MapThread[
     Block[{ptTri = #1, source = point, normal = #2, target, facept, cross},
       If[First @@ Position[ptTri, source] == 1,
        {target, facept} = {ptTri[[2]], ptTri[[-1]]};
        cross = Cross[normal, facept - target],
        {target, facept} = {ptTri[[1]], ptTri[[-1]]};
        cross = Cross[normal, target - facept]
       ];
       1/2 cross
      ] &, x]] /@ {{opentriExample, normalsO}, {closedtriExample, normalsC}}
```

Out[71]= {{0.00809089, 0.03573710, 0.35449768}, {-0.2184970, 0.2382116, 1.3354782}}

In[72]:= ϵ_{co} openSCont + ϵ_{cc} closedSCont

Out[72]= $\{-0.2184970\, \epsilon_{cc} + 0.00809089\, \epsilon_{co},\ 0.2382116\, \epsilon_{cc} + 0.03573710\, \epsilon_{co},\ 1.3354782\, \epsilon_{cc} + 0.35449768\, \epsilon_{co}\}$