Power Crust Algorithm

Helia Ghorbani, Ghazal Pouresfandiyar, Rezvan Sabahi

About Power Crust

Power Crust is an algorithm developed by Nina Amenta, Sunghee Choi, and Ravi Krishna Kolluri at the University of Texas at Austin. The algorithm takes in a point cloud as input, and outputs a surface mesh and the corresponding medial axis transform (MAT). A point cloud is a collections of points in space, typically in either 2D or 3D. These points are usually captured using a 3D scanner or similar technology, which, like all sensors that attempt to capture the real world, can often result in noisy data. Point clouds are typically assumed to be hallow in the inside, and the points represent points on the surface of the scanned object.

Algorithm Overview

At a high level, the algorithm works by computing a set of balls on the interior and the exterior of the point cloud. These balls are referred to as “polar balls”, named as such because they are located at a set of Voronoi vertices from the input points referred to as “poles”. Each polar ball is given a radius equal to the distance between it and any sample point on the point cloud’s surface. Because the balls were positioned at the Voronoi vertices, this results in the set of balls on the inside of the cloud, and those on the outside, reaching towards each other to meet up at the point cloud surface itself. The algorithm simply needs to run a labelling process to determine which poles belong on the inside of the mesh and which on the outside, and then it can reconstruct a dense surface from where the balls meet up.

After the polar balls have been appropriately labeled, the algorithm creates another Voronoi diagram, this time using the poles themselves as inputs. This time, it creates a special kind of Voronoi diagram called a power diagram. A power diagram is just a weighted version of the standard Voronoi diagram, where each input point is assigned a weight to describe its influence on the Voronoi (“power”) regions. The process uses the square of the polar ball’s radius as it’s weight.

The power diagram generated in the previous step is then used to generate the mesh and the MAT. The mesh is defined as the set of vertices and edges in the power diagram separating the interior poles from the outer poles. These edges and vertices can be represented as a polygonal mesh with the same dimensionality as the input point cloud. The medial axis is defined as the set of poles labeled as interior regions. To form the MAT, we just join the medial axis points together based on the connectivity of the power diagram. If we take the dual of the interior power diagram cells, it forms the MAT.

Note: Make sure to change this path in `powercrust.m` according to your path to run the project.

```
% Change it corresponding to your path
addpath('G:\4012\Graph\Power-Crust\power diagrams');
```

DisplayMedialAxis.m

```
function DisplayMedialAxis(MedialAxis, MAT)
% Takes in a medial axis and MAT, and displays it in a figure window
%
% inputs:
% MedialAxis - the collection of points representing the medial axis
% MAT - the medial axis transform. Represented by a cell array, where each
%       cell is a set of two points, representing an edge between medial axis
%       points

% Determine the dimensionality of the MedialAxis
[~, dim] = size(MedialAxis);

% Create a new figure window
figure;
hold on;

% Iterate over each edge in the MAT and plot it
for i = 1:length(MAT)
    edge = MAT{i};
    if (dim == 2)
        % If the MedialAxis is in 2D, plot the edge as a line
        plot(edge(:, 1), edge(:, 2), 'color', 'b');
    elseif (dim == 3)
        % If the MedialAxis is in 3D, plot the edge as a line in 3D space
        plot3(edge(:, 1), edge(:, 2), edge(:, 3), 'color', 'b');
    end
end

% Plot the MedialAxis points
if (dim == 2)
    % If the MedialAxis is in 2D, plot the points as red dots
    plot(MedialAxis(:, 1), MedialAxis(:, 2), 'Marker', '.', 'MarkerEdgeColor', 'r', 'MarkerSize', 5, 'LineStyle', 'none');
elseif (dim == 3)
    % If the MedialAxis is in 3D, plot the points as red dots in 3D space
    plot3(MedialAxis(:, 1), MedialAxis(:, 2), MedialAxis(:, 3), 'Marker', '.', 'MarkerEdgeColor', 'r', 'MarkerSize', 10, 'Line
Style', 'none');
end

% Set the title of the figure
title('Medial Axis Transform');

% Release the hold on the figure
hold off;

end
```

Explanation:

- The function `DisplayMedialAxis` takes two inputs: `MedialAxis` and `MAT`.
- `MedialAxis` is a collection of points representing the medial axis.
- `MAT` is the medial axis transform, represented by a cell array where each cell contains two points representing an edge between medial axis points.
- The function creates a new figure window to display the medial axis.
- It iterates over each edge in the `MAT` and plots it.
- If the `MedialAxis` is in 2D, the function plots the edges as lines and the points as red dots.
- If the `MedialAxis` is in 3D, the function plots the edges as lines in 3D space and the points as red dots in 3D space.
- The title of the figure is set as "Medial Axis Transform".
- The `hold off` command releases the hold on the figure, allowing for further modifications or plotting in the same figure window.

DisplayMesh.m

```
function DisplayMesh(meshEdges)
% Takes in a mesh and displays it in a figure window
%
% inputs:
% meshEdges - a cell array where each cell holds two points that represent
% an edge on the surface mesh

% Get the number of edges in the mesh
[numEdges, ~] = size(meshEdges);

% Create a new figure window
figure;
hold on;

% Iterate over each edge in the mesh and plot it
for i = 1:numEdges
    pts = meshEdges{i};
    [~, dim] = size(pts);
    if (dim == 2)
        % If the mesh is in 2D, plot the edge as a line
        plot(pts(:, 1), pts(:, 2), 'color', 'r');
    elseif (dim == 3)
        % If the mesh is in 3D, plot the edge as a line in 3D space
        plot3(pts(:, 1), pts(:, 2), pts(:, 3), 'color', 'r');
    end
end

% Set the title of the figure
title('Surface Mesh');
```

```
% Release the hold on the figure
hold off;

end
```

Explanation:

- The function `DisplayMesh` takes one input: `meshEdges`, which is a cell array where each cell holds two points representing an edge on the surface mesh.
- It extracts the number of edges in the mesh using the `size` function.
- The function creates a new figure window to display the mesh.
- It iterates over each edge in the `meshEdges` and plots it.
- If the mesh is in 2D, the function plots the edges as lines.
- If the mesh is in 3D, the function plots the edges as lines in 3D space.
- The title of the figure is set as "Surface Mesh".
- The `hold off` command releases the hold on the figure, allowing for further modifications or plotting in the same figure window.

FindBoundingPoints.m

```
function [boundsPoints] = FindBoundingPoints(points, multiplier)
% To avoid infinitely large Voronoi cells, we find the bounding points away
% from the regular point cloud. The number of bounding points returned will
% depend on the number of dimensions
%
% inputs:
% points - the point cloud for which we are looking for bounding points
% multiplier - the distance away from the point cloud to put the bounds at
%
% outputs:
% boundsPoints - a matrix of points, where each row is a point, and each
%                column represents a dimension

% Get the size of the point cloud
[~, cols] = size(points);

% Initialize the matrix to hold the bounding points
boundsPoints = zeros(2^cols, cols);

% Iterate over each dimension
for i = 1:cols
    % Calculate the maximum, minimum, and mean of the points in the current dimension
    maxPt = max(points(:, i));
    minPt = min(points(:, i));
    meanPt = mean(points(:, i));

    % Calculate the differences between the maximum and mean, and the minimum and mean
    diffMax = maxPt - meanPt;
    diffMin = minPt - meanPt;

    % Update the maximum and minimum points by adding the differences multiplied by the multiplier
    maxPt = maxPt + diffMax * multiplier;
    minPt = minPt + diffMin * multiplier;

    % Calculate the number of repeats based on the current dimension
    numRepeats = 2^(i-1);

    % Set the bounding points based on the updated maximum and minimum values
    j = 1;
    while j < length(boundsPoints)
        for k = 1:numRepeats
            boundsPoints(j, i) = maxPt;
            j = j + 1;
        end
        for k = 1:numRepeats
            boundsPoints(j, i) = minPt;
            j = j + 1;
        end
    end
end
```

```
end
```

Explanation:

- The function `FindBoundingPoints` takes two inputs: `points`, which is the point cloud for which we are looking for bounding points, and `multiplier`, which represents the distance away from the point cloud to place the bounding points.
- It extracts the number of columns in the `points` matrix using the `size` function.
- The function initializes a matrix `boundsPoints` to hold the bounding points. The size of the matrix is determined by `2^cols` rows and `cols` columns.
- It iterates over each dimension of the point cloud.
- For each dimension, it calculates the maximum, minimum, and mean values of the points in that dimension.
- It then calculates the differences between the maximum and mean, and the minimum and mean.
- The maximum and minimum points are updated by adding the differences multiplied by the `multiplier`.
- The number of repeats is calculated based on the current dimension.
- The function sets the bounding points in the `boundsPoints` matrix by alternating between the updated maximum and minimum values, based on the number of repeats.
- Finally, it returns the `boundsPoints` matrix containing the bounding points.

FindMedialAxis.m

```
function [MedialAxis, MAT] = FindMedialAxis(poles, labels, powerDiagram)
% This process finds the medial axis and medial axis transform of the points
%
% inputs:
% poles - the set of poles from the transform
% labels - a vector representing the label of each pole (1=inside, 0=outside)
% powerDiagram - the power diagram from the labels. It is a cell array,
%                 where each cell contains a list of all vertices for each pole's power region.
%
% outputs:
% MedialAxis - the set of points representing the medial axis of the point cloud
% MAT - the medial axis transform. Represents a set of edges between MedialAxis points

% The MedialAxis is just the set of poles labeled as being on the inside
MedialAxis = poles(labels, :);

% Find the neighboring Voronoi cells to determine the MAT
MAT = cell(length(MedialAxis), 1);
idx = 1;
for i = 1:length(powerDiagram)
    if labels(i)
        % If this pole is on the inside, look for neighbors also on the inside
        verts = powerDiagram{i};
        [rows, ~] = size(verts);
        for k = i + 1:length(powerDiagram)
            if labels(k)
                foundMatch = false;
                for j = 1:rows
                    thisVert = verts(j, :);
                    otherVerts = powerDiagram{k};
                    containsThisVert = ~isempty(find(otherVerts(:, 1) == thisVert(1) & otherVerts(:, 2) == thisVert(2), 1));
                    if containsThisVert
                        % We found a shared vertex between these regions. They are connected, stop looking
                        foundMatch = true;
                        break;
                    end
                end
                if foundMatch
                    % Since these regions are touching, add an edge between them in the MAT
                    newEdge = [poles(i, :); poles(k, :)];
                    MAT(idx) = {newEdge};
                    idx = idx + 1;
                end
            end
        end
    end
```

```
end

end
```

Explanation:

- The function `FindMedialAxis` takes three inputs: `poles`, which is the set of poles from the transform, `labels`, a vector representing the label of each pole (1=inside, 0=outside), and `powerDiagram`, which is the power diagram from the labels represented as a cell array.
- The function initializes an empty matrix `MedialAxis` to store the points representing the medial axis.
- It selects the poles that are labeled as being on the inside and assigns them to `MedialAxis`.
- The function also initializes an empty cell array `MAT` to store the medial axis transform, which represents a set of edges between the points in `MedialAxis`.
- It iterates over each power diagram cell.
- If the corresponding pole is labeled as being on the inside, it proceeds to look for neighboring regions also on the inside.
- For each pair of regions, it checks if they have a shared vertex by comparing their vertices.
- If a shared vertex is found, an edge is created between the two poles in `MedialAxis` and added to `MAT`.
- Finally, the function returns `MedialAxis` and `MAT`.

FindPoles.m

```
function [poleVerts, poleRadMat, sampleIdxForPole, oppositePoleIdx] = FindPoles(verts, cells, points)
% Finds the set of poles for the point cloud, given the Voronoi cells and vertices.
% The poles are a subset of the Voronoi vertices that represent the two most extreme points
% on either side of each Voronoi cell. There is typically one pole inside the mesh and one outside,
% but they will be labeled later.
%
% Inputs:
% verts - the Voronoi vertices of the point cloud
% cells - the Voronoi cells of the point cloud
% points - the original input points
%
% Outputs:
% poleVerts - a list of the vertices of the poles. Each row represents a distinct point, and the
%              columns represent the coordinate values
% poleRadMat - a list of the radius for each pole. Represents the distance from the pole
%               to its nearest point in the point cloud
% sampleIdxForPole - a list representing the original point associated with each pole,
%                     given as an index into the points list
% oppositePoleIdx - a list representing the opposite pole for each pole,
%                    given as an index into the poleVerts list

% Initialize arrays
polesMat = ones(length(cells) * 2, 1) * -1;
poleRadMat = ones(length(cells) * 2, 1) * -1;
sampleIdxForPole = ones(length(cells) * 2, 1) * -1;
oppositePoleIdx = ones(length(cells) * 2, 1) * -1;

% Look through each Voronoi cell, finding the extreme vertices and adding them to the pole list
j = 1;
for i = 1:length(cells)
    thisPoint = points(i, :);
    thisCell = cells{i};
    cellsVertices = verts(thisCell, :);

    % Find the furthest Voronoi vertex
    thisPointMat = repmat(thisPoint, length(cellsVertices), 1);
    distFromPoint = sum((cellsVertices - thisPointMat) .^ 2, 2);
    distFromPoint = sqrt(distFromPoint);
    [~, idx] = max(distFromPoint);
    firstPole = cellsVertices(idx, :);
    firstPoleIdx = thisCell(idx);
    firstPoleMat = repmat(firstPole, length(cellsVertices), 1);
    firstRad = distFromPoint(idx);

    % Find the second pole
    vectorToFurthestMat = firstPoleMat - thisPointMat;
    vectorToEach = cellsVertices - thisPointMat;
    negativeDot = dot(vectorToFurthestMat, vectorToEach, 2) < 0;
    onlyNegatives = distFromPoint(negativeDot, :);
    [secondRad, idx] = max(onlyNegatives);
    filteredVertices = thisCell(negativeDot);
    secondPoleIdx = filteredVertices(idx);

    addedFirst = false;

    if isempty(find(polesMat == firstPoleIdx, 1))
        % If the first pole wasn't created yet, add it to the list
        polesMat(j) = firstPoleIdx;
        poleRadMat(j) = firstRad;
        sampleIdxForPole(j) = i;
        secondOppIdx = j;
        firstj = j;
        j = j + 1;
        addedFirst = true;
    else
        secondOppIdx = find(polesMat == firstPoleIdx);
    end
    if isempty(find(polesMat == secondPoleIdx,

1))
        % If the second pole wasn't created yet, add it to the list
        polesMat(j) = secondPoleIdx;
        poleRadMat(j) = secondRad;
        sampleIdxForPole(j) = i;
        firstOppIdx = j;
        oppositePoleIdx(j) = secondOppIdx;
        j = j + 1;
    else
        firstOppIdx = find(polesMat == secondPoleIdx);
    end

    % If the first pole was added, set its opposite pole
    if addedFirst
        oppositePoleIdx(firstj) = firstOppIdx;
    end

end

% Remove leftover entries from the lists
polesMat(polesMat == -1) = [];
poleRadMat(poleRadMat == -1) = [];
sampleIdxForPole(sampleIdxForPole == -1) = [];
oppositePoleIdx(oppositePoleIdx == -1) = [];

% Create the list of pole points instead of just indices into the point list
poleVerts = verts(polesMat, :);

end
```

Explanation:

- The function `FindPoles` takes three inputs: `verts`, which is the Voronoi vertices of the point cloud, `cells`, the Voronoi cells of the point cloud, and `points`, the original input points.
- It initializes empty arrays `polesMat`, `poleRadMat`, `sampleIdxForPole`, and `oppositePoleIdx` to store information about the poles.
- The function then iterates over each Voronoi cell.
- For each cell, it finds the extreme vertices by calculating the distance from each vertex to the corresponding point.
- The furthest vertex is considered the first pole, and its index and radius are recorded.
- The second pole is found by considering the negative dot product between the vector from the first pole to the corresponding point and the vectors from the other vertices to the corresponding point. The vertex with the maximum negative dot product is selected as the second pole.
- If the first pole hasn't been added to the list yet, it is added along with its information. Otherwise, its index is retrieved.
- The same process is repeated for the second pole, and its index is recorded as the opposite pole index for the first pole.
- The process continues for all cells, adding poles and their information to the respective lists.
- After iterating over all cells, any remaining entries with the initial value of -1 in the lists are removed.
- Finally, the function creates the list of pole points (`poleVerts`) by indexing the vertices using the pole indices (`polesMat`).

Note: The code assumes that the variable `verts` contains the coordinates of all Voronoi vertices, `cells` is a cell array where each cell contains the indices of the vertices that form a Voronoi cell, and `points` is a matrix where each row represents a point in the original input.

FindSurfaceMesh.m

```
function [poleVerts, poleRadMat, sampleIdxForPole, oppositePoleIdx] = FindPoles(verts, cells, points)
% Finds the set of poles for the point cloud, given the Voronoi cells and vertices.
% The poles are a subset of the Voronoi vertices that represent the two most extreme points
% on either side of each Voronoi cell. There is typically one pole inside the mesh and one outside,
% but they will be labeled later.
%
% Inputs:
% verts - the Voronoi vertices of the point cloud
% cells - the Voronoi cells of the point cloud
% points - the original input points
%
% Outputs:
% poleVerts - a list of the vertices of the poles. Each row represents a distinct point, and the
%             columns represent the coordinate values
% poleRadMat - a list of the radius for each pole. Represents the distance from the pole
%              to its nearest point in the point cloud
% sampleIdxForPole - a list representing the original point associated with each pole,
%                    given as an index into the points list
% oppositePoleIdx - a list representing the opposite pole for each pole,
%                   given as an index into the poleVerts list

% Initialize arrays
polesMat = ones(length(cells) * 2, 1) * -1;
poleRadMat = ones(length(cells) * 2, 1) * -1;
sampleIdxForPole = ones(length(cells) * 2, 1) * -1;
oppositePoleIdx = ones(length(cells) * 2, 1) * -1;

% Look through each Voronoi cell, finding the extreme vertices and adding them to the pole list
j = 1;
for i = 1:length(cells)
    thisPoint = points(i, :);
    thisCell = cells{i};
    cellsVertices = verts(thisCell, :);

    % Find the furthest Voronoi vertex
    thisPointMat = repmat(thisPoint, length(cellsVertices), 1);
    distFromPoint = sum((cellsVertices - thisPointMat) .^ 2, 2);
    distFromPoint = sqrt(distFromPoint);
    [~, idx] = max(distFromPoint);
    firstPole = cellsVertices(idx, :);
    firstPoleIdx = thisCell(idx);
    firstPoleMat = repmat(firstPole, length(cellsVertices), 1);
    firstRad = distFromPoint(idx);

    % Find the second pole
    vectorToFurthestMat = firstPoleMat - thisPointMat;
    vectorToEach = cellsVertices - thisPointMat;
    negativeDot = dot(vectorToFurthestMat, vectorToEach, 2) < 0;
    onlyNegatives = distFromPoint(negativeDot, :);
    [secondRad, idx] = max(onlyNegatives);
    filteredVertices = thisCell(negativeDot);
    secondPoleIdx = filteredVertices(idx);

    addedFirst = false;

    if isempty(find(polesMat == firstPoleIdx, 1))
        % If the first pole wasn't created yet, add it to the list
        polesMat(j) = firstPoleIdx;
        poleRadMat(j) = firstRad;
        sampleIdxForPole(j) = i;
        secondOppIdx = j;
        firstj = j;
        j = j + 1;
        addedFirst = true;
    else
        secondOppIdx = find(polesMat == firstPoleIdx);
    end
    if isempty(find(polesMat == secondPoleIdx,

1))
        % If the second pole wasn't created yet, add it to the list
        polesMat(j) = secondPoleIdx;
        poleRadMat(j) = secondRad;
        sampleIdxForPole(j) = i;
        firstOppIdx = j;
        oppositePoleIdx(j) = secondOppIdx;
        j = j + 1;
    else
        firstOppIdx = find(polesMat == secondPoleIdx);
    end

    % If the first pole was added, set its opposite pole
    if addedFirst
        oppositePoleIdx(firstj) = firstOppIdx;
    end

end

% Remove leftover entries from the lists
polesMat(polesMat == -1) = [];
poleRadMat(poleRadMat == -1) = [];
sampleIdxForPole(sampleIdxForPole == -1) = [];
oppositePoleIdx(oppositePoleIdx == -1) = [];

% Create the list of pole points instead of just indices into the point list
poleVerts = verts(polesMat, :);

end
```

Explanation:

- The function `FindPoles` takes three inputs: `verts`, which is the Voronoi vertices of the point cloud, `cells`, the Voronoi cells of the point cloud, and `points`, the original input points.
- It initializes empty arrays `polesMat`, `poleRadMat`, `sampleIdxForPole`, and `oppositePoleIdx` to store information about the poles.
- The function then iterates over each Voronoi cell.
- For each cell, it finds the extreme vertices by calculating the distance from each vertex to the corresponding point.
- The furthest vertex is considered the first pole, and its index and radius are recorded.
- The second pole is found by considering the negative dot product between the vector from the first pole to the corresponding point and the vectors from the other vertices to the corresponding point. The vertex with the maximum negative dot product is selected as the second pole.
- If the first pole hasn't been added to the list yet, it is added along with its information. Otherwise, its index is retrieved.
- The same process is repeated for the second pole, and its index is recorded as the opposite pole index for the first pole.
- The process continues for all cells, adding poles and their information to the respective lists.
- After iterating over all cells, any remaining entries with the initial value of -1 in the lists are removed.
- Finally, the function creates the list of pole points (`poleVerts`) by indexing the vertices using the pole indices (`polesMat`).

Note: The code assumes that the variable `verts` contains the coordinates of all Voronoi vertices, `cells` is a cell array where each cell contains the indices of the vertices that form a Voronoi cell, and `points` is a matrix where each row represents a point in the original input.

FindSurfaceMesh.m

```
function [meshVerts, meshEdges] = FindSurfaceMesh(labels, edgeList, vertsForCells, polePts, poleRads, boundingBox)
  % generates the surface mesh for the model
  %
  % inputs:
  % labels - a vector representing the label of each pole (1=inside,0=outside)
  % edgeList - a list of power cells from the power diagram of the poles
  %           each cell contains a list of edges between vertices
  % vertsForCells - a list of the vertices associated with each cell of the power diagram
  % polePts -  a list of the vertices of the poles. Each row represents a distinct point, and the
  %           columns represent the coordinate values
  % poleRads - a list of the radius for each pole. Represents the distance from the pole
  %           to its nearest point in the point cloud
  % boundingBox - a list of points representing the box around the input
  %              point cloud
  %
  % outputs:
```

```
  % meshVerts - a list of all the vertices on the output surface mesh
  % meshEdges - a list of edges between the meshVerts, creating a solid polygonal mesh
  %             represented by a cell array, where each cell contains two vertices that can be joined with a line


  %% find the lists of all vertices touching the inside, and all vertices touching the outside
  insideVerts = [];
  outsideVerts = [];
  % iterate through each cell, adding all vertices to the inside and outside lists
  for i = 1:length(labels)
    verts = vertsForCells{i};
    [numVerts, ~] = size(verts);
    label = labels(i);
    if (label == 0)
      % add all vertices to the outside list
      [currentIdx, ~] = size(outsideVerts);
      outsideVerts(currentIdx+1:currentIdx+numVerts, :) = verts;
    else
      % add all vertices to the inside list
      [currentIdx, ~] = size(insideVerts);
      insideVerts(currentIdx+1:currentIdx+numVerts, :) = verts;
    end
  end

  %% find the list of vertices that are on both the inside and outside
  meshVerts = intersect(insideVerts, outsideVerts, 'rows');
  meshVerts = unique(meshVerts, 'rows');

  % toss any vertices that are outside of the bounding box
  inBox = IsInBoundingBox(meshVerts, boundingBox);
  meshVerts(~inBox, :) = [];

  %% find the set of edges between border vertices
  [~, dim] = size(meshVerts);
  meshEdges = cell(length(meshVerts), 1);
  edgeMidPts = zeros(length(meshVerts), dim);
  idx = 1;
  % iterate through every edge in the power diagram
  for i = 1:length(edgeList)
    points = edgeList{i};
    % find the vertices involved in the edge
    pt1 = points(1, :);
    pt2 = points(2, :);
    onBorder1 = ismember(pt1, meshVerts, 'rows');
    onBorder2 = ismember(pt2, meshVerts, 'rows');
    % if both vertices are on the border of the inside and the outside, keep the edge
    if(onBorder1 && onBorder2)
      midpt = (pt1 + pt2) ./ 2;
      meshEdges(idx, 1) = {points};
      edgeMidPts(idx, :) = midpt;
      idx = idx +

1;
    end
  end

  %% filter out the edges that intersect significantly with a polar ball
  intersectsBall = zeros(length(edgeMidPts), 1);
  significantVal = 0.02;

  for i = 1:length(edgeMidPts)
    % find the distance between the mid point of the edge and all polar balls
    midPt = edgeMidPts(i, :);
    midPtMat = repmat(midPt, length(polePts), 1);
    distanceMat = midPtMat - polePts;
    distanceMat = distanceMat .^ 2;
    distanceMat = sum(distanceMat, 2);
    distanceMat = sqrt(distanceMat);
    distanceMat = distanceMat - poleRads;
    % if they overlap significantly, mark it as overlapping
    if(min(distanceMat) < -significantVal)
      intersectsBall(i) = 1;
    end
  end
  % remove all overlapping values from the list of edges
  meshEdges = meshEdges(~intersectsBall);
end
```

Explanation:

- The `FindSurfaceMesh` function generates a surface mesh for a model based on the given inputs.

- It starts by initializing empty arrays for `insideVerts` and `outsideVerts`, which will store the vertices touching the inside and outside of the model, respectively.
- It iterates through each cell and adds all the vertices to either the `insideVerts` or `outsideVerts` list based on the corresponding label.
- After finding the lists of vertices touching the inside and outside, the function finds the vertices that are on both the inside and outside (`meshVerts`) by taking the intersection of `insideVerts` and `outsideVerts`. Duplicate vertices are removed using the `unique` function.
- The function checks if any vertices are outside of the bounding box using the `IsInBoundingBox` function. Vertices outside the box are discarded.
- Next, the function finds the set of edges between the border vertices. It iterates through each edge in the power diagram (`edgeList`). If both vertices of an edge are on the border of the inside and outside (`meshVerts`), the edge is kept, and the midpoint is calculated. The edge and midpoint information are stored in `meshEdges` and `edgeMidPts`, respectively.
- The function then filters out the edges that significantly intersect with a polar ball. It iterates through each midpoint in `edgeMidPts` and calculates the distance between the midpoint and all polar balls (`polePts`). If the distance minus the pole radius is below a significant value (`significantVal`), the edge is marked as overlapping. Finally, the overlapping edges are removed from `meshEdges`.
- In the end, the function returns `meshVerts` and the filtered `meshEdges`, representing the surface mesh of the model.

IsInBoundingBox.m

```
function [inBox] = IsInBoundingBox(pts, box)
% checks whether a set of points are within a bounding box
%
% inputs:
% pts - the points to check
% box - the box we are comparing the points to,
%       specified as a list of coordinate values
%
% outputs:
% inBox - a logical vector representing whether each point in
% pts is in the bounding box

  [~, cols] = size(box); % Get the dimensions of the bounding box
  inBox = true(length(pts), 1); % Initialize inBox with true values for each point

  for i=1:cols
    ptDim = pts(:,i); % Extract the current dimension values from pts
    maxVal = max(box(:,i)); % Find the maximum value for the current dimension in the bounding box
    minVal = min(box(:,i)); % Find the minimum value for the current dimension in the bounding box
    lessThanMax = ptDim < maxVal; % Check if each point dimension is less than the maximum value
    moreThanMin = ptDim > minVal; % Check if each point dimension is greater than the minimum value
    matchesBoth = and(lessThanMax, moreThanMin); % Check if a point satisfies both conditions
    inBox = and(inBox, matchesBoth); % Update inBox by performing element-wise logical AND
  end
end
```

Explanation:

- The `IsInBoundingBox` function checks whether a set of points (`pts`) are within a given bounding box (`box`).
- It starts by getting the dimensions of the bounding box using the `size` function, and stores the number of columns in `cols`.
- It initializes the `inBox` logical vector with `true` values for each point in `pts`.
- The function then iterates through each column of the bounding box (`box`).
- For each column, it extracts the corresponding dimension values from `pts` using indexing (`ptDim`).
- It finds the maximum and minimum values for the current dimension in the bounding box using the `max` and `min` functions, respectively (`maxVal` and `minVal`).
- Next, it checks if each point dimension in `pts` is less than the maximum value (`lessThanMax`) and greater than the minimum value (`moreThanMin`) using element-wise comparisons.
- It combines the two conditions using the `and` function to determine if a point matches both conditions (`matchesBoth`).

- Finally, the `inBox` logical vector is updated by performing element-wise logical AND (`and(inBox, matchesBoth)`) to keep track of whether each point satisfies all the dimension constraints of the bounding box.

The function returns the `inBox` logical vector, which represents whether each point in `pts` is within the bounding box.

LabelPoles.m

```
function [officialLabels] = LabelPoles(polePoints, inputPoints, weights, poleSampleIdx, poleOppositeIdx)
% Assigns a label to each pole. Each pole will be labeled as either inside (1) or outside (0)
%
% inputs:
% polePoints - the list of poles (extreme Voronoi vertices of the point cloud)
% inputPoints - the original input point cloud
% weights - a list of the weight (radius) for each pole
% poleSampleIdx - the original point in the point cloud that created each pole,
%                  represented as a list of indices into the inputPoints array
% poleOppositeIdx - the opposite pole associated with each pole in polePoints,
%                    represented as a list of indices into polePoints
%
% output:
% officialLabels - a vector representing the label of each pole (1=inside, 0=outside)

  % Initialize the final label array
  officialLabels = ones(length(polePoints), 1) * -1;

  % Initialize in and out arrays to keep track of the likelihood of a pole being on the inside or the outside
  out = zeros(length(polePoints), 1);
  in = zeros(length(polePoints), 1);

  % Mark all poles outside of the bounding box as outer poles
  boundingBox = FindBoundingPoints(inputPoints, 0);
  inBox = IsInBoundingBox(polePoints, boundingBox);
  out(~inBox) = 1;

  % Keep track of the indices of the remaining, unlabeled poles
  % This array will shrink as they are assigned labels, but can be used to map between indices
  remainingIndices = (1:length(polePoints))';

  % Generate the initial priority queue based on the in and out values of the poles
  priorityQueue = generatePriorityQueue(in, out);

  while (~isempty(remainingIndices))
    % Pop the next value off the priority queue
    nextIdx = priorityQueue(1, 2);
    inVal = in(nextIdx);
    outVal = out(nextIdx);
    overallIdx = remainingIndices(nextIdx);
    sampleIdx = poleSampleIdx(nextIdx);
    oppIdx = poleOppositeIdx(nextIdx);

    % Assign the label based on which value is most likely
    if (outVal >= inVal)
      officialLabels(overallIdx) = 0;
    else
      officialLabels(overallIdx) = 1;
    end

    % Remove the value from all lists
    in(nextIdx) = [];
    out(nextIdx) = [];
    remainingIndices(nextIdx) = [];
    poleSampleIdx(nextIdx) = [];
    poleOppositeIdx(nextIdx) = [];

    % If there are still points left, recalculate in and out values
    if (~isempty(remainingIndices))
      % Find which points overlap. They should likely be given the same label
      remainingPts = polePoints(remainingIndices, :);
      remainingPtRad = weights(remainingIndices);
      thisPt = polePoints(overallIdx, :);
      thisPtRad = weights(overallIdx);

      if (officialLabels(overallIdx) == 1)
        in = recalculateOverlapping(remainingPts, remainingPtRad, thisPt, thisPtRad, in);
      else
        out = recalculateOverlapping(remainingPts, remainingPtRad, thisPt, thisPtRad, out);
      end

      % Calculate the probability that the opposite pole is assigned the opposite label
      if (~isempty(find(remainingIndices == oppIdx, 1)))
        convertedIdx =

 find(remainingIndices == oppIdx);
        oppPt = polePoints(oppIdx, :);
        samplePt = inputPoints(sampleIdx, :);
        newValue = recalculateOppositePole(oppPt, samplePt, thisPt);

        if (officialLabels(overallIdx) == 1)
          out(convertedIdx) = max(newValue, out(convertedIdx));
        else
          in(convertedIdx) = max(newValue, out(convertedIdx));
        end
      end

      % Regenerate priority queue with new values
      priorityQueue = generatePriorityQueue(in, out);
    end
  end

  % Convert the official labels into a logical vector. They should all be 0 or 1
  officialLabels = logical(officialLabels);
end

% Recalculates the priority queue using the method described in the paper
function priorityQueue = generatePriorityQueue(in, out)
  [numPoles, ~] = size(in);

  % Find the priority queue values
  priority = abs(in - out) - 1;

  for i = 1:numPoles
    % If only one of in or out has a value != 0, use that value instead
    if ((in(i) == 0 && out(i) ~= 0) || (in(i) ~= 0 && out(i) == 0))
      priority(i, 1) = in(i) + out(i);
    end
  end

  [sorted, I] = sort(priority, 'descend');
  priorityQueue = [sorted, I];
end

% Calculates the probability that the opposite pole should be given the opposite label
% from the current pole. This process is based on the angle formed between the original sample
% and the two opposite poles
function [newValue] = recalculateOppositePole(oppositePt, samplePt, thisPt)
  v1 = oppositePt - samplePt;
  v1 = v1 / norm(v1);
  v2 = thisPt - samplePt;
  v2 = v2 / norm(v2);
  newValue = -cos(acos(dot(v1, v2)));
end

% Calculates new label values for poles that overlap with the current pole
% If they overlap by a large amount, they likely share the same label
function [sameLabel] = recalculateOverlapping(remainingPoints, remainingWeights, thisPt, thisRad, sameLabel)
  [numRemaining, ~] = size(remainingPoints);
  thisPtMat = repmat(thisPt, numRemaining, 1);
  thisPtRadMat = repmat(thisRad, numRemaining, 1);
  distanceMat = thisPtMat - remainingPoints;
  distanceMat = distanceMat .^ 2;
  distanceMat = sum(distanceMat, 2);
  distanceMat = sqrt(distanceMat);
  distanceMat = distanceMat - thisPtRadMat - remainingWeights;

  % Use a sigmoid function to map overlap rating between 0 and 1
  quarterRadius = thisRad * 0.25;
  overlapRating = sigmf(-distanceMat, [0.01, quarterRadius]);

  sameLabel = max(sameLabel, overlapRating);
end
```

Explanation:

- The `LabelPoles` function assigns a label (inside or outside) to each pole based on a set of criteria.
- It takes several inputs, including `polePoints` (the list of poles), `inputPoints` (the original point cloud), `weights` (a list of weights/radii for each pole), `poleSampleIdx` (the indices of the original points that created each pole), and `poleOppositeIdx` (the indices of

the opposite poles associated with each pole).

- The function initializes the `officialLabels` array with -1 values and creates `in` and `out` arrays to track the likelihood of each pole being on the inside or the outside.
- It marks poles outside of the bounding box as outer poles and updates the `out` array accordingly.
- The function enters a while loop that continues until all poles have been assigned labels.
- In each iteration of the loop, it selects the next pole based on the priority queue.
- The function assigns the label to the selected pole based on the likelihood values in the `in` and `out` arrays.
- The selected pole is removed from all lists, and if there are remaining poles, the `in` and `out` values are recalculated.
- The function also recalculates the probability that the opposite pole is assigned the opposite label from the current pole.
- The priority queue is regenerated with the updated values.
- Finally, the `officialLabels` are converted into a logical vector and returned as the output.

The `generatePriorityQueue` function creates a priority queue based on the `in` and `out` values of the poles. The priority is determined by the absolute difference between `in` and `out` values.

The `recalculateOppositePole` function calculates the probability that the opposite pole should be given the opposite label based on the angle formed between the original sample and the two opposite poles.

The `recalculateOverlapping` function calculates new label values for poles that overlap with the current pole. If the overlap is significant, they are likely to share the same label. The overlap rating is calculated using a sigmoid function.

Overall, the code iteratively assigns labels to poles based on various criteria and uses probability calculations to determine the label assignment.

PowerCrust.m

```
function [MeshVerts, MeshEdges, MedialAxis, MAT] = PowerCrust(points)
    % Power Crust algorithm implementation
    % Takes a 2D or 3D point cloud as input and returns a surface mesh and
    % the medial axis transform

    %% Setup
    % Add the required path
    addpath('G:\\4012\\Graph\\Power-Crust-MATLAB-master\\power diagrams');

    % Remove duplicate points
    points = unique(points, 'rows');
    points = double(points);

    % Check the dimensionality of the point cloud
    [~, dim] = size(points);
    if (dim ~= 2 && dim ~= 3)
        error('Point cloud must be 2D or 3D');
    end

    %% Step 1: Voronoi Diagram
    disp('Finding Voronoi Diagram');
    % Add boundary points to avoid infinite Voronoi cells
    boundPts = FindBoundingPoints(points, 5);
    points = [points ; boundPts];

    % Create a Voronoi diagram from the points
    [verts, cells] = voronoin(points);

    % Remove the last cells, which correspond to the bounding points
    numBounds = length(boundPts);
    cells = cells(1:length(cells) - numBounds, :);
    points = points(1:length(points) - numBounds, :);

    %% Step 2: Find Poles
    disp('Finding Poles');
    [poleVerts, poleRadMat, sampleIdxForPole, oppositePoleIdx] = FindPoles(verts, cells, points);

    %% Step 3: Compute Power Diagram of Poles
    disp('Finding Power Diagram');
    [PD, ~] = powerDiagramWrapper(poleVerts, poleRadMat .^ 2);

    %% Step 4: Label Poles
    disp('Labeling Poles');
    labels = LabelPoles(poleVerts, points, poleRadMat, sampleIdxForPole, oppositePoleIdx);

    %% Step 5: Generate Outputs
    disp('Generating Mesh/Medial Axis');
    boundingBox = FindBoundingPoints(points, 1);
    [MeshVerts, MeshEdges] = FindSurfaceMesh(labels, PD{dim}, PD{1}, poleVerts, poleRadMat, boundingBox);
    [MedialAxis, MAT] = FindMedialAxis(poleVerts, labels, PD{1});

    DisplayMesh(MeshEdges);
    DisplayMedialAxis(MedialAxis, MAT);
end
```

Explanation:

- The `PowerCrust` function implements the Power Crust algorithm.
- It takes a 2D or 3D point cloud as input (`points`) and returns the surface mesh (`MeshVerts`, `MeshEdges`) and the medial axis transform (`MedialAxis`, `MAT`).
- The code begins with setup, including adding the required path, removing duplicate points, and checking the dimensionality of the point cloud.
- Step 1 computes the Voronoi diagram of the points, including adding boundary points to avoid infinite Voronoi cells.
- Step 2 finds the poles of the Voronoi diagram.
- Step 3 computes the power diagram of the poles.
- Step 4 labels the poles based on proximity to the original points.
- Step 5 generates the output surface mesh and medial axis, using the labeled poles and power diagram.
- Finally, the code calls the `DisplayMesh` and `DisplayMedialAxis` functions to visualize the results.

Note: The code assumes the existence of various helper

functions (`FindBoundingPoints`, `FindPoles`, `powerDiagramWrapper`, `LabelPoles`, `FindSurfaceMesh`, `FindMedialAxis`, `DisplayMesh`, `DisplayMedialAxis`), which are not provided in the code snippet.

edgeAttPD.m

```
function EA = edgeAttPD(T, edges)
% function EA = edgeAttPD(T, edges)
%
% T: pieces of a triangulation
% edges: array of edges in the triangulation
%
% Each entry in the output cell EA corresponds to an edge from the input
% edges and contains all pieces of the triangulation attached to that edge.

[me, ne] = size(edges);
EA = cell(me,1);

for i=1:me
    EA{i} = find(sum(ismember(T, edges(i,:)), 2) == ne)';
end
```

- The function `edgeAttPD` takes two inputs: `T` which represents pieces of a triangulation and `edges` which is an array of edges in the triangulation.
- It initializes an empty cell array `EA` to store the attached pieces of the triangulation for each edge.
- The variable `me` stores the number of rows and `ne` stores the number of columns in the `edges` array.
- It then loops over each row of the `edges` array using the variable `i`.
- Inside the loop, it checks for each piece in the `T` triangulation if it contains the current edge. This is done using the `ismember` function to check if each element of `T` matches the current edge in `edges`.
- The `sum` function is used to count the number of matches (equal to `ne`) in each row of `T`.
- The logical condition `sum(...) == ne` returns a logical array indicating which rows of `T` have all the elements of the current edge in `edges`.

- The `find` function is used to find the indices of the rows where the condition is true.
- Finally, the resulting indices are stored in the corresponding cell of `EA`, indicating the pieces of the triangulation attached to the current edge.
- After the loop completes, the function returns the `EA` cell array containing the attached pieces for each edge.

Overall, the code computes the attached pieces of a triangulation for each edge and organizes the results in a cell array for further analysis or processing.

freeBouPD.m

```
function FB = freeBouPD(T, P)
% function FB = freeBouPD(T, P)
%
% T: triangulation
% P: pieces of the triangulation
%
% The output FB contains the pieces of P on the boundary of T.

[~, nT] = size(T);
[mP, ~] = size(P);

ii=1;
for i=1:mP
    if size(find(sum(ismember(T, P(i,:)),2)==nT-1),1)==1
        FB(ii,:) = P(i,:);
        ii = ii+1;
    end
end
```

- The function `freeBouPD` takes two inputs: `T`, which represents a triangulation, and `P`, which represents the pieces of the triangulation.
- It initializes an empty matrix `FB` to store the pieces of `P` on the boundary of `T`.
- The variables `nT` and `mP` store the number of columns in `T` and the number of rows in `P`, respectively.
- It initializes the variable `ii` to keep track of the row index in `FB`.
- It then loops over each row of `P` using the variable `i`.
- Inside the loop, it checks for each piece in `P` if it is on the boundary of `T`. This is done by comparing the sum of matches (equal to `nT-1`) between each row of `T` and the current piece in `P` using `ismember` and `sum`.
- The `find` function is used to find the indices where the condition is true (i.e., only one match on the boundary).
- The `size` function is used to determine the number of rows in the resulting indices. If it is equal to `1`, the piece is on the boundary.
- If the condition is true, the current piece is added to the `FB` matrix at the row index `ii` and `ii` is incremented.
- After the loop completes, the function returns the `FB` matrix containing the pieces of `P` on the boundary of `T`.

The code identifies and extracts the pieces of a triangulation that lie on the boundary, allowing further analysis or processing of those specific pieces.

liftPD.m

```
function LE = liftPD(E, wts)
% function LE = liftPD(E, wts)
%
% E: set of points
% wts: weights of the points in E
%
% The output array LE contains the points of E lifted one dimension higher.

[N, d] = size(E);
LE = zeros(N,d+1);

for i=1:N
```

```
    x = E(i,:);
    LE(i,:) = [x, x*x' - wts(i)];
end
```

- The function `liftPD` takes two inputs: `E`, which represents a set of points, and `wts`, which represents the weights of the points in `E`.
- It initializes the variables `N` and `d` to store the number of rows and the number of columns in `E`, respectively.
- It initializes an empty matrix `LE` with `N` rows and `d+1` columns to store the lifted points of `E` one dimension higher.
- The `for` loop iterates over each row of `E` using the variable `i`.
- Inside the loop, it assigns the current row of `E` to the variable `x`.
- It then assigns the lifted point to the corresponding row in `LE` using concatenation: `[x, x*x' - wts(i)]`.
  - The first `d` columns of the lifted point are the same as the original point `x`.
  - The last column is calculated by taking the dot product of `x` with itself (`x*x'`) and subtracting the corresponding weight `wts(i)`.
- After the loop completes, the function returns the `LE` matrix containing the points of `E` lifted one dimension higher.

The code performs a lifting operation on a set of points by adding an extra dimension to each point. The lifted points are obtained by appending a computed value to the original coordinates of each point. This operation can be useful in certain mathematical and computational geometry algorithms.

normalsPD.m

```
function ind = normalsPD(LE, C)
% function ind = normalsPD(E, C)
%
% LE: set of lifted points
% C: convex hull of LE
%
% The output ind contains indices of facets of the lower hull.

[m, n] = size(C);
center = mean(LE,1);

% Loop over each facet of the convex hull
for i=1:m
    % Calculate the null space of the matrix formed by subtracting the first row of LE from the remaining rows in C(i,:)
    v = null(bsxfun(@minus, LE(C(i,1),:), LE(C(i,2:end),:)))';
    [mm, ~] = size(v);
    if mm > 1
        % If the null space has more than one row, it may be degenerate. Set the i-th row of V as NaN.
        V(i,:) = NaN;
        % possibility of degenerate null vectors
    else
        % Otherwise, assign the null space vector to the i-th row of V.
        V(i,:) = v;
    end
    % Calculate the mean of the lifted points corresponding to the i-th row of C.
    mid(i,:) = mean(LE(C(i,:),:),1);
end

% Calculate the dot product between the vectors formed by subtracting center from each row of mid and the corresponding row of V.
dot = sum(bsxfun(@minus, center, mid).*V, 2);
outer = dot < 0;

% Negate the rows of V corresponding to the outer facets.
V(outer,:) = -1*V(outer,:);

% Assign true to the ind variable for the rows of V where the last column (V(:,n)) is greater than 0.
ind = V(:,n) > 0;
```

- The function `normalsPD` takes two inputs: `LE`, which represents a set of lifted points, and `C`, which represents the convex hull of `LE`.
- It initializes the variables `m` and `n` to store the number of rows and columns in `C`, respectively.
- It calculates the center of `LE` by taking the mean of all the lifted points and assigns it to the variable `center`.

- The function then enters a loop that iterates over each row of `C` using the variable `i`.
- Inside the loop, it calculates the null space of a matrix formed by subtracting the first row of `LE` from the remaining rows in the `i`th row of `C`. The resulting null space vector is assigned to the variable `v`.
- It checks the size of `v` to determine if there is a possibility of degenerate null vectors. If the number of rows in `v` is greater than 1, it assigns `NaN` to the `i`th row of the matrix `V`. Otherwise, it assigns `v` to the `i`th row of `V`.
- It calculates the mean of the lifted points corresponding to the `i`th row of `C` and assigns it to the `i`th row of the matrix `mid`.
- It calculates the dot product between the vectors formed by subtracting `center` from each row of `mid` and the corresponding row of `V`.
- It identifies the outer facets by checking if the dot product is less than 0 and assigns `true` to the `outer` variable for those facets.
- It negates the rows of `V` corresponding to the outer facets.
- Finally, it assigns `true` to the `ind` variable for the rows of `V` where the last column (`V(:,n)`) is greater than 0.

The code calculates the normals of the facets of the lower hull of a convex hull in higher-dimensional space. It computes the null space vectors of certain submatrices formed by the lifted points, checks for degenerate cases, and determines the orientation of the facets based on their dot product with the center of the lifted points. The resulting indices of the lower hull facets are returned in the `ind` variable.

piecesPD.m

Certainly! Here's the modified code with added comments and an explanation of the code's functionality:

```
function [P, total] = piecesPD(T)
% function [P, total] = piecesPD(T)
%
% Decomposes a triangulation into its constituent pieces of various dimensions.
%
% Inputs:
%   - T: Triangulation represented by an m-by-n matrix, where each row
%        corresponds to a simplex (e.g., triangle) in n-dimensional space.
%
% Outputs:
%   - P: Cell array where each entry P{i} contains the pieces of dimension i.
%   - total: Total number of pieces across all dimensions.
%

[m, n] = size(T);
P = cell(n,1);  % Initialize cell array to store pieces of different dimensions
total = 0;     % Initialize total count of pieces

% Extract vertices (dimension zero) from the triangulation
P{1} = unique(T);  % Unique vertices
total = total + size(P{1}, 1);  % Update total count

% Iterate over each dimension from 1 to n-1
for i = 2:n-1
    Q = [];  % Temporary storage for combinations of i vertices

    % Generate all combinations of i vertices from each row of T
    for j = 1:m
        Q = [Q; combnk(T(j,:), i)];
    end

    Q = sort(Q, 2);  % Sort combinations in ascending order along each row
    P{i} = unique(Q, 'rows');  % Unique combinations (pieces) of dimension i
    total = total + size(P{i}, 1);  % Update total count
    clear Q;  % Clear temporary storage
end

P{n} = T;  % Fully-dimensional facets (dimension n-1) are assigned as the last entry
total = total + size(P{n}, 1);  % Update total count

end
```

Explanation:

- The `piecesPD` function decomposes a given triangulation `T` into its constituent pieces of various dimensions.

- The function initializes the cell array `P` to store the pieces and sets the total count `total` to zero.
- The function starts by extracting the vertices (dimension zero) from the triangulation `T` using the `unique` function. These vertices are stored in `P{1}`.
- Next, the function iterates over each dimension from 1 to `n-1`, where `n` represents the dimensionality of the triangulation.
- For each dimension `i`, the function generates all combinations of `i` vertices from each row of `T` using the `combnk` function. The combinations are stored in the temporary variable `Q`.
- The combinations in `Q` are sorted in ascending order along each row using the `sort` function.
- Duplicate combinations are removed by applying the `unique` function to `Q` with the option `'rows'`. The resulting unique combinations (pieces) of dimension `i` are stored in `P{i}`.
- The count of unique combinations for dimension `i` is added to the `total` variable.
- After processing all dimensions from 1 to `n-1`, the fully-dimensional facets (dimension `n-1`) are assigned to `P{n}` as the last entry in the cell array.
- The count of fully-dimensional facets is added to the `total` variable.
- Finally, the function returns `P`, containing all the pieces of the triangulation, and the `total` number of pieces.

This decomposition of a triangulation into pieces of different dimensions can be useful in various applications, such as mesh processing, topological analysis, or geometric algorithms.

powercentersPD.m

```
function [PC, powers] = powercentersPD(T, E, wts)
% function [PC, powers] = powercentersPD(T, E, wts)
%
% Computes the power centers of triangles in a triangulation.
%
% Inputs:
%   - T: Triangulation represented by an m-by-n matrix, where each row
%        corresponds to a triangle and contains the indices of its vertices.
%   - E: Set of points in the triangulation, represented as an m-by-d matrix,
%        where d is the dimension of the points.
%   - wts: Weights of the points in E, represented as an m-by-1 vector.
%
% Outputs:
%   - PC: Matrix containing the power centers of the triangles. Each row
%         corresponds to a triangle and contains the coordinates of its power center.
%   - powers: Vector containing the power of each power center.

[m, n] = size(T);
PC = zeros(m, n-1);  % Initialize matrix to store power centers
powers = zeros(m, 1);  % Initialize vector to store powers

for i = 1:m
    triangle = E(T(i, :), :);  % Get the vertices of the current triangle
    weight = wts(T(i, :));  % Get the weights of the triangle's vertices

    firstPoint = triangle(1, :);
    firstPointMat = repmat(firstPoint, n-1, 1);
    otherTwo = triangle(2:n, :);
    diffFromFirst = 2 * (otherTwo - firstPointMat);

    pointWeightMat = repmat(weight(1), n-1, 1);
    otherWeights = weight(2:n);
    firstSquared = repmat(sum(firstPoint.^2), n-1, 1);
    otherSquared = sum(otherTwo.^2, 2);
    Bc = pointWeightMat - otherWeights - firstSquared + otherSquared;

    pc = diffFromFirst \\ Bc;  % Solve the linear system to find the power center

    PC(i, :) = pc';  % Store the power center in the output matrix
    powers(i, 1) = norm(pc - firstPoint')^2 - weight(1);  % Compute the power of the power center
end
```

Explanation:

- The `powercentersPD` function computes the power centers of triangles in a triangulation.

- The function takes as input the triangulation matrix `T`, the set of points `E`, and the weights `wts` associated with each point.
- The function initializes the output matrix `PC` to store the power centers and the output vector `powers` to store the corresponding powers.
- It then iterates over each triangle in the triangulation.
- For each triangle, the function extracts the triangle's vertices and weights.
- It performs the computation of the power center using a linear system of equations. The power center is the point equidistant from each vertex of the triangle, taking into account the weights of the vertices.
- The linear system is solved by constructing the necessary matrices and vectors and using the backslash operator `\\`.
- The resulting power center is stored in the `PC` matrix, and its power is computed and stored in the `powers` vector.
- Finally, after processing all triangles, the function returns the `PC` matrix containing the power centers and the `powers` vector containing the powers associated with each power center.

The power centers and their corresponding powers can be useful in various geometric computations and algorithms, such as centroidal Voronoi tessellation or Delaunay refinement.

powerdiagram.m

```
function [vxx, vy] = powerdiagram(varargin)
% Power diagram
% modified voronoi.m to use power centers instead of circumcenters

[cax, args, nargs] = axescheck(varargin{:});
error(nargchk(1, 5, nargs));

x = args{1};
y = args{2};
if ~isequal(size(x), size(y))
    error(message('MATLAB:voronoi:InputSizeMismatch'));
end
if ~ismatrix(x) || ~ismatrix(y)
    error(message('MATLAB:voronoi:HigherDimArray'));
end
x = x(:);
y = y(:);
tri = args{3};
ls = args{4};
wts = args{5};

if isempty(tri)
    return;
end

% Compute power centers of triangles
sE = [x, y];
[c, ~] = powercentersPD(tri, sE, wts);

% Create matrix T where i and j are endpoints of edge of triangle T(i,j)
n = numel(x);
t = repmat((1:size(tri, 1))', 1, 3);
T = sparse(tri, tri(:, [3 1 2]), t, n, n);

% i and j are endpoints of internal edge in triangle E(i,j)
E = (T & T') .* T;
% i and j are endpoints of external edge in triangle F(i,j)
F = xor(T, T') .* T;

% v and vv are triangles that share an edge
[~, ~, v] = find(triu(E));
[~, ~, vv] = find(triu(E'));

% Internal edges
vx = [c(v, 1) c(vv, 1)]';
vy = [c(v, 2) c(vv, 2)]';

%%% Compute lines-to-infinity
% i and j are endpoints of the edges of triangles in z
[i, j, z] = find(F);
% Counter-clockwise components of lines between endpoints
dx = x(j) - x(i);
dy = y(j) - y(i);
```

```
% Calculate scaling factor for length of line-to-infinity
% Distance across range of data
rx = max(x) - min(x);
ry = max(y) - min(y);
% Distance from vertex to center of data
cx = (max(x) + min(x)) / 2 - c(z, 1);
cy = (max(y) + min(y)) / 2 - c(z, 2);
% Sum of these two distances
nm = sqrt(rx .* rx + ry .* ry) + sqrt(cx .* cx + cy .* cy);
% Compute scaling factor
scale = nm ./ sqrt((dx .* dx + dy .* dy));

% Lines from voronoi vertex to "infinite" endpoint
% We know it's in the correct direction because components are counterclockwise (CCW)
ex = [c(z, 1) c(z, 1) - dy .* scale]';
ey = [c(z, 2) c(z, 2) + dx .* scale]';
% Combine with internal edges
vx = [vx ex];
vy = [vy ey];

if nargout < 2
    % Plot diagram
    if isempty(cax)
        % If no current axes, create one
        cax = gca;
    end
    if isempty(ls)
        % Default linespec
        ls = '-';
    end
    [l, c, mp, msg] = colstyle(ls); error(msg) % Extract from linespec


 if isempty(mp)
        % Default markers at points
        mp = '.';
    end
    if isempty(l)
        % Default linestyle
        l = get(ancestor(cax, 'figure'), 'DefaultAxesLineStyleOrder');
    end
    if isempty(c)
        % Default color
        co = get(ancestor(cax, 'figure'), 'DefaultAxesColorOrder');
        c = co(1, :);
    end
    % Plot points
    h1 = plot(x, y, 'marker', mp, 'color', c, 'linestyle', 'none', 'parent', cax);
    % Plot voronoi lines
    h2 = line(vx, vy, 'color', c, 'linestyle', l, 'parent', cax, 'yliminclude', 'off', 'xliminclude', 'off');
    if nargout == 1
        vxx = [h1; h2];
    end % Return handles
else
    vxx = vx; % Don't plot, just return vertices
end
```

Explanation:

- The function `powerdiagram` implements a modified version of the Voronoi diagram algorithm using power centers instead of circumcenters.
- The input arguments to the function are:
  - `x` and `y`: Coordinates of the input points.
  - `tri`: Triangulation of the points.
  - `ls`: Linespec for plotting the diagram (optional).
  - `wts`: Weights of the points (optional).
- The function starts by checking the input arguments and preparing the data.
- It computes the power centers of the triangles using the `powercentersPD` function.
- It creates a matrix `T` where each row represents an edge of a triangle in the triangulation.
- It identifies internal and external edges of the triangles.

- It computes the lines-to-infinity for the diagram.
- It combines the internal edges and lines-to-infinity to obtain the Voronoi edges.
- If the output is not requested, it plots the diagram using the specified linespec.
- Finally, it returns the Voronoi vertices if requested.

Note: The code includes some commented lines for plotting the diagram, which can be uncommented if visualizing the diagram is desired.

pwrDiagramPD.m

```
function PD = pwrDiagramPD(T, PC)
% function PD = pwrDiagramPD(T, PC)
%
% T: triangulation
% PC: power centers of triangles in T
%
% The output cell PD contains pieces of the power diagram, indexed by
% dimension. PD{mP} contains pieces of dimension zero (power centers that
% correspond to fully-dimensional pieces of the triangulation T) and PD{1}
% contains fully-dimensional regions of the power diagram (corresponding to
% vertices of the triangulation).

% Compute pieces of the triangulation
P = piecesPD(T);
mP = size(P, 1);
PD = cell(mP, 1);

% Iterate over each dimension of the power diagram
for i = 1:mP
    % Compute edge attributes for the current dimension
    EA = edgeAttPD(T, P{i});

    % Map the edge attributes to power centers
    for j = 1:size(EA, 1)
        EA{j} = PC(EA{j}, :);
    end

    % Store the mapped edge attributes in the power diagram cell array
    PD{i} = EA;
end
```

Explanation:

- The function `pwrDiagramPD` computes the power diagram based on the input triangulation `T` and the power centers `PC` of the triangles.
- The output `PD` is a cell array that contains pieces of the power diagram, indexed by dimension.
- The outer loop iterates over each dimension of the power diagram.
- Inside the loop, the function computes the edge attributes for the current dimension using the `edgeAttPD` function.
- The edge attributes are then mapped to the corresponding power centers by indexing the `PC` array with the edge attributes.
- The mapped edge attributes are stored in the power diagram cell array `PD`.
- Finally, the function returns the computed power diagram.

Note: The code assumes the existence of the `piecesPD` and `edgeAttPD` functions, which are not provided in the code snippet.

Runner2D.m

```
% Load the teapot point cloud
ptCloud = pcread('teapot.ply');
points = double(ptCloud.Location);

% Flatten the point cloud into 2D
ptCloudA = points(:, 1:2:3);
ptCloudB = points(:, 2:3);
ptCloudC = points(:, 1:2);
```

```
% Choose one of the flattenings as the point cloud
points = ptCloudA;

% Remove interior points using the boundary function
k = boundary(points(:, 1), points(:, 2), 1);
points = points(k, :);

% Plot the original point cloud
figure;
plot(points(:, 1), points(:, 2), 'Marker', '.', 'MarkerEdgeColor', 'r', 'MarkerSize', 10, 'LineStyle', 'none');
title('Original input');

% Run Power Crust algorithm
PowerCrust(points);
```

Explanation:

- The code loads a teapot point cloud from a PLY file and assigns its 3D coordinates to the variable `points`.
- Three different flattenings (`ptCloudA` , `ptCloudB` , `ptCloudC`) of the point cloud are provided, but the code selects `ptCloudA` as the point cloud to be used.
- The code applies the `boundary` function to remove interior points from the point cloud, retaining only the boundary points.
- It then plots the original point cloud.
- Finally, it calls the `PowerCrust` function, passing in the `points` to compute the surface mesh and medial axis transform.

Note: Make sure to have the `teapot.ply` file in the same directory as the script or provide the correct path to the file. Additionally, ensure that the `PowerCrust` function and its helper functions are accessible or provided in the same script.

Runner3D.m

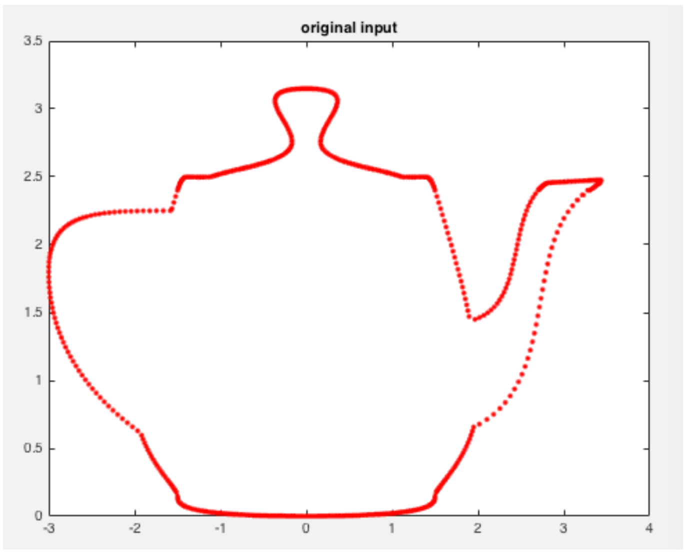
```
% Load the teapot point cloud
ptCloud = pcread('teapot.ply');

% Downsample the point cloud
ptCloud = pcdownsample(ptCloud, 'gridAverage', 0.25);

% Extract the point coordinates
points = ptCloud.Location;

% Plot the original point cloud
figure;
pcshow(ptCloud);
title('Original input');

% Run Power Crust algorithm
PowerCrust(points);
```

Explanation:

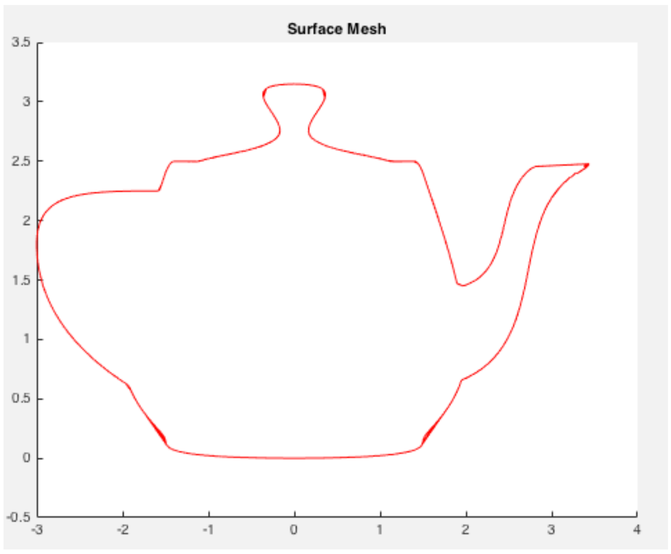
- The code loads a teapot point cloud from a PLY file using the `pcread` function.
- It then applies downsampling to the point cloud using `pcdownsample` , with the 'gridAverage' method and a voxel size of 0.25. This reduces the density of the point cloud.
- The point coordinates are extracted from the point cloud using `ptCloud.Location` and stored in the variable `points`.
- The original point cloud is plotted using `pcshow`.
- Finally, it calls the `PowerCrust` function, passing in the `points` to compute the surface mesh and medial axis transform.

Note: Make sure to have the `teapot.ply` file in the same directory as the script or provide the correct path to the file. Additionally, ensure that the `PowerCrust` function and its helper functions are accessible or provided in the same script.

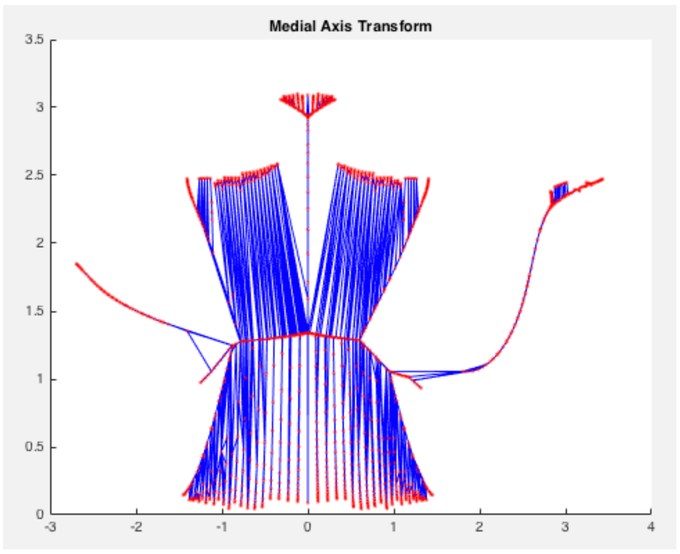
Results:
Input (2D):



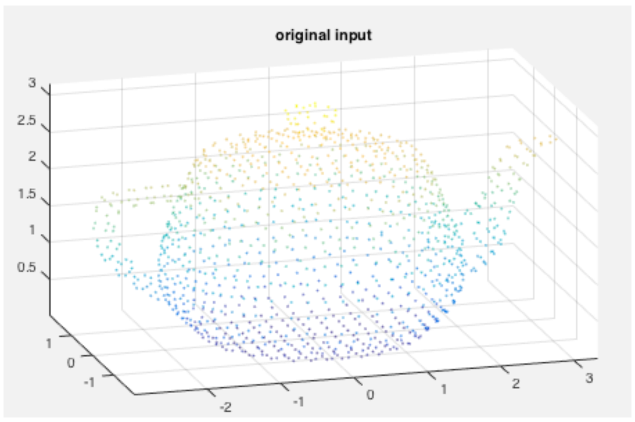
Mesh (2D):



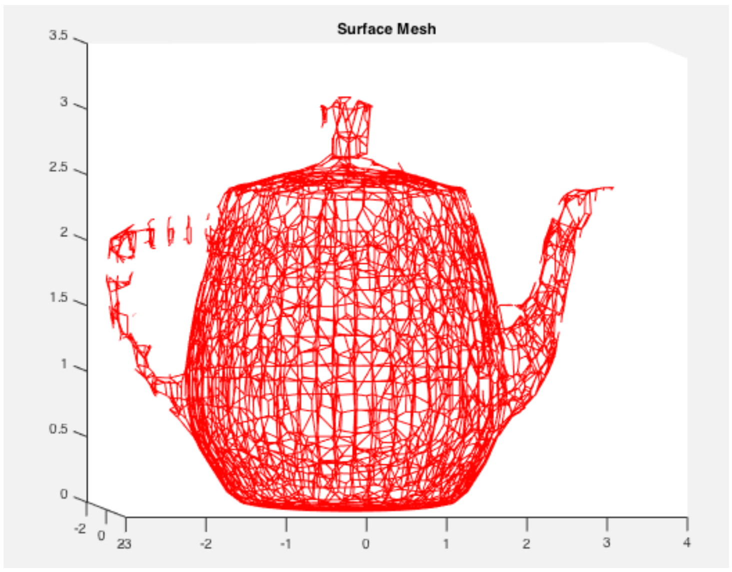
Output MAT (2D):



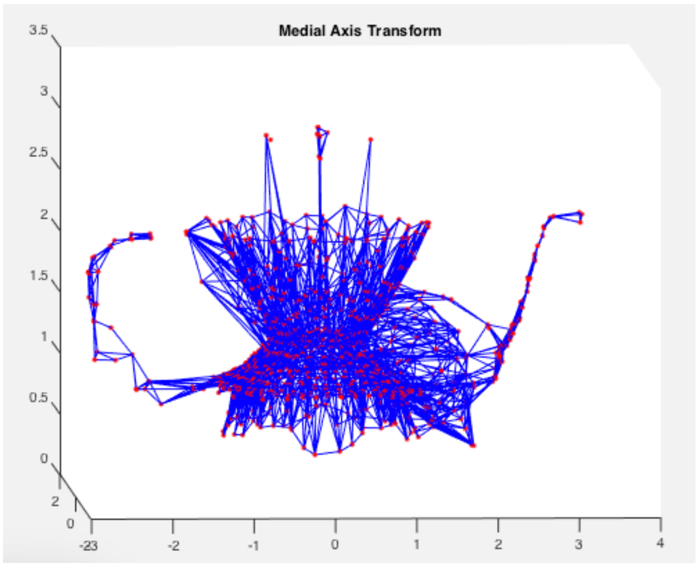
Input (3D):



Mesh (3D):



Output MAT (3D):



Reference:

- https://github.com/daniel-sanche/Power-Crust-MATLAB
- https://github.com/kubkon/powercrust
- Amenta, N., Choi, S. and Kolluri, R.K., 2001, May. The power crust. In *Proceedings of the sixth ACM symposium on Solid modeling and applications* (pp. 249-266).