SUPPLEMENTAL INFORMATION

S.1 Crack detection protocol in TGC

Step 1: Import .h5 file into ImageJ and save as an Image Sequence of .tiffs

Step 2: Apply the structural and modified median filters, and extract grains

tomoFilter.m

```
scan = '69'; % scan number
state = '1'; % loading step
sample = '02'; % sample number

% find the .tiffs and initialize other folders
imFolder = ['H:/CHESStransfer/dcdc-gc-' sample '/tomo_' scan '/tiffs/'];
baseName = ['s' state '_t' scan '_'];
%altName = ['s4_t179']; %for naming mistakes ie forgetting _
saveFolderStr = ['H:/CHESStransfer/dcdc-gc-' sample '/tomo_' ...
    scan '/filtered_step1/'];
saveFolderFin = ['H:/CHESStransfer/dcdc-gc-' sample '/tomo_' ...
    scan '/filtered_step2/'];
saveFolderGrain = ['H:/CHESStransfer/dcdc-gc-' sample '/tomo_' ...
    scan '/final_grains/'];

baseim = imread([imFolder baseName '0000.tif']); % test image for dimensions
[zDim, xDim] = size(baseim);
yDim = size(dir(imFolder),1) - 2; % directory always counts two extra

% import the corners for the desired sample
bigcorners = [556, 44; 57, 898; 840, 1344; 1340, 489]; % tomo 69, sam2
% move away from the edges to avoid affecting the background mean
% 40 pixels in at 30d angle for sample 02
smallcorners = bigcorners + [18, 24; 24, -18; -18, -24; -24, 18];

% calculate bounding box for ROI around crack
ma = (smallcorners(2,2) - smallcorners(3,2)) / ...
    (smallcorners(2,1) - smallcorners(3,1));
ba = smallcorners(2,2) - ma * smallcorners(2,1);
% hole is located in center of x face
xa = max(round(smallcorners(2,1) + ...
    0.4 * (smallcorners(3,1) - smallcorners(2,1))), 0);
za = max(round(ma * xa + ba), 0); % no values outside the image frame
xb = max(round(smallcorners(2,1) + ...
    0.6 * (smallcorners(3,1) - smallcorners(2,1))), 0);
zb = max(round(ma * xb + ba), 0);

mb = (smallcorners(4,2) - smallcorners(1,2)) / ...
    (smallcorners(4,1) - smallcorners(1,1));
bb = smallcorners(4,2) - mb*smallcorners(4,1);
md = (smallcorners(2,2) - smallcorners(1,2)) / ...
    (smallcorners(2,1) - smallcorners(1,1));
bd = za - md * xa;
```

```
xd = max(round((bd - bb) / (mb - md)), 0);
zd = max(round(md * xd + bd), 0);
xc = max(xd + xb - xa,0);
zc = max(round(mb * xc + bb),0);

roiCorners = [xa,za; xb,zb; xc,zc; xd,zd];

zindices = repmat(1:xDim, [zDim 1]);
xindices = repmat(1:zDim, [xDim 1])';
ROI = inpolygon(xindices, zindices, roiCorners(:,1), roiCorners(:,2));

% set the rectangular structuring element to approximately the size and
% shape of the vertical streaks
recelem = strel('rectangle',[25 3]);

% set the cubic structuring element to extend one pixel in every dir.
cubelem = strel('cube',3);

% structural element filtering
parfor iim = 0:yDim-1
    useim = imread([imFolder, baseName, sprintf('%04d',iim), '.tif']);
    adjim = uint16(useim*1.5e6);
    strim = imopen(adjim,recelem); % open with structural element
    midim = medfilt2(adjim - strim,[1 3]); % top hat

    % save for further filtering
    filtim = Tiff([saveFolderStr, baseName, ...
        sprintf('%04d',iim), '.tif'], 'w');
    tagstr = struct();
    tagstr.ImageLength = size(useim,1);
    tagstr.ImageWidth = size(useim,2);
    tagstr.Photometric = Tiff.Photometric.MinIsBlack;
    tagstr.BitsPerSample = 16;
    tagstr.SamplesPerPixel = 1;
    tagstr.SampleFormat = Tiff.SampleFormat.UInt;
    tagstr.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
    tagstr.Compression = Tiff.Compression.None;
    filtim.setTag(tagstr);
    filtim.write(midim);
    filtim.close();
end

parfor islice = 2:yDim-4 % don't use edges because of local stack size
    localStack = int16(zeros(zDim,xDim,3));
    sliceNum = islice + 1;
    % get a stack 5 slices deep
    for jlocal = 1:5
        localStack(:,:,jlocal) = imread([saveFolderStr, baseName, ...
            sprintf('%04d',sliceNum + jlocal - 3), '.tif']);
    end

    midim = localStack(:,:,3);
    grains = localStack>4000; % find grains
    laygrains = midim>4000; % find grains in the central layer
    gbw = bwconncomp(laygrains); % get clusters of points in grain binary
```

```
gns = bwpropfilt(gbw,'Area',[20 10000]); % remove spurious points
gfilt = cc2bw(gns);
% Save the processed image for the current slice
grainim = Tiff([saveFolderGrain, baseName, ...
    sprintf('%04d',sliceNum + jlocal - 3), '.tif'], 'w');
tagfin = struct();
tagfin.ImageLength = zDim;
tagfin.ImageWidth = xDim;
tagfin.Photometric = Tiff.Photometric.MinIsBlack;
tagfin.BitsPerSample = 8;
tagfin.SamplesPerPixel = 1;
tagfin.SampleFormat = Tiff.SampleFormat.UInt;
tagfin.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
tagfin.Compression = Tiff.Compression.None;
grainim.setTag(tagfin);
grainim.write(uint8(gfilt)*255);
grainim.close();

% create random noise around backgroud mean where grains are located so
% bright points aren't included in local background calculations
grainIntens = localStack(grains);
laygrainIntens = midim(laygrains);
numgrainPts = size(grainIntens,1);
rpl = 1000*rand(numgrainPts,1)+1600;
localStack(grains) = rpl;

% open and close with the cubic element to remove noisy points both
% dark and bright
newstack = 0.5*imclose(localStack,cubelem) + ...
    0.5*imopen(localStack,cubelem);
newmidim = newstack(:,:,3);
placeholder = newmidim;

% get median values for regions of different sizes around each point
localenvA = medfilt3(newstack,[3 3 3]);
useenvA = localenvA(:,:,3);
localenvB = medfilt3(newstack,[5 5 5]);
useenvB = localenvB(:,:,3);
localenvC = medfilt3(newstack,[7 7 5]);
useenvC = localenvC(:,:,3);
localenvE = medfilt3(newstack,[11 11 5]);
useenvE = localenvE(:,:,3);

% get the mean of each point in the stack at the location to reduce
% noise and strengthen persistent signals
deep3d = sum(newstack,3)/5;

% create conditions for finding the dark and bright crack sections
% based on analysis of values for representative regions
crackCheck = deep3d<1600 & useenvB<1600 & useenvC<1800 & ROI;
darkCheck = ~crackCheck & deep3d<1600 & ~laygrains & ROI;
artCheck = deep3d>2200 &  useenvB>2200 & useenvC>2000 & ~laygrains & ROI;
brightCheck = ~artCheck & deep3d>2200 & ~laygrains & ROI;
otherCheck = ~crackCheck & ~artCheck & ~laygrains & ...
    ~brightCheck & ~darkCheck;
```

```
    % strengthen signal for the strongest signals, use a small ROI median
    % filter on those somewhat strong, use a large ROI median filter elsewhere
    placeholder(crackCheck) = 0.75*deep3d(crackCheck);
    placeholder(darkCheck) = useenvA(darkCheck);
    placeholder(artCheck) = 1.2*deep3d(artCheck);
    placeholder(brightCheck) = useenvA(brightCheck);
    placeholder(otherCheck) = useenvE(otherCheck);

    % replace grains in image
    placeholder(laygrains) = laygrainIntens;

    % save filtered image
    portim = Tiff([saveFolderFin, baseName, ...
        sprintf('%04d',islice), '.tif'], 'w');
    tagfin = struct();
    tagfin.ImageLength = size(midim,1);
    tagfin.ImageWidth = size(midim,2);
    tagfin.Photometric = Tiff.Photometric.MinIsBlack;
    tagfin.BitsPerSample = 16;
    tagfin.SamplesPerPixel = 1;
    tagfin.SampleFormat = Tiff.SampleFormat.Int;
    tagfin.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
    tagfin.Compression = Tiff.Compression.None;
    portim.setTag(tagfin);
    portim.write(placeholder);
    portim.close();
end
```

Step 3: Use ImageJ to Enhance Contrast and normalize for the top and bottom 1% of pixels in the entire image sequence, and save

Step 4: Get connected components after thresholding and filter them by size and location to find those belonging to the crack

bwPropFilter.m

```
scan = '69'; % scan number
state = '1'; % state number
sample = '02'; % sample number
baseFolder = ['H:/CHESStransfer/dcdc-gc-' sample '/tomo_' scan ...
    '/contrast/s' state '_t' scan '_'];
saveFolder = ['H:/CHESStransfer/dcdc-gc-' sample '/tomo_' scan ...
    '/final_crack_bins/s' state '_t' scan '_'];
baseim = imread([baseFolder '0000.tif']); % sample image for dimensions
[zDim, xDim] = size(baseim);
yDim = size(dir(['H:/CHESStransfer/dcdc-gc-' sample '/tomo_' scan ...
    '/contrast/']),1) - 2;

% same procedure for ROI as in tomoFilter.m
bigcorners = [556, 44; 57, 898; 840, 1344; 1340, 489]; % tomo 69, sam2
smallcorners = bigcorners + [18, 24; 24, -18; -18, -24; -24, 18]; % sam2

zindices = repmat(1:xDim,[zDim 1]);
```

```
xindices = repmat(1:zDim,[xDim 1])';
bigROI = inpolygon(xindices, zindices, smallcorners(:,1), smallcorners(:,2));

ma = (smallcorners(2,2) - smallcorners(3,2)) / ...
    (smallcorners(2,1) - smallcorners(3,1));
ba = smallcorners(2,2) - ma * smallcorners(2,1);
xa = max(round(smallcorners(2,1) + ...
    0.45 * (smallcorners(3,1) - smallcorners(2,1))), 0);
za = max(round(ma * xa + ba), 0);
xb = max(round(smallcorners(2,1) + ...
    0.55 * (smallcorners(3,1) - smallcorners(2,1))), 0);
zb = max(round(ma * xb + ba), 0);

mb = (smallcorners(4,2) - smallcorners(1,2)) / ...
    (smallcorners(4,1) - smallcorners(1,1));
bb = smallcorners(4,2) - mb * smallcorners(4,1);
md = (smallcorners(2,2) - smallcorners(1,2)) / ...
    (smallcorners(2,1) - smallcorners(1,1));
bd = za - md * xa;
xd = max(round((bd - bb) / (mb - md)), 0);
zd = max(round(md * xd + bd), 0);
xc = max(xd + xb - xa,0);
zc = max(round(mb * xc + bb), 0);

roiCorners = [xa,za; xb,zb; xc,zc; xd,zd];

% get edge pixels for the crack ROI
smallROI = inpolygon(xindices, zindices, roiCorners(:,1),roiCorners(:,2));
longEdge = round(sqrt((xa-xd)^2 + (za-zd)^2));
shortEdge = abs(xa-xb);
xBegin = round(linspace(xa,xd,longEdge));
xFinal = round(linspace(xb,xc,longEdge));
xCen = round((xBegin+xFinal)/2);
zBegin = round(linspace(za,zd,longEdge));
zFinal = round(linspace(zb,zc,longEdge));
zCen = round((zBegin+zFinal)/2);

goodCrack = false(zDim,xDim,yDim);

parfor iyslice = 0:yDim-2
    testim = im2int16(imread(sprintf([baseFolder '%04d.tif'],iyslice)));
    crackbw = testim<-24000 & bigROI; % get points below threshold
    cc = bwconncomp(crackbw);
    cnc = bwpropfilt(cc, 'Circularity', [0 0.7]); % keep noncircular
    co = bwpropfilt(cnc,'Orientation',[-30 45]); % some deviation, esp near hole
    cbm = bwpropfilt(co, 'MinorAxisLength', [50 500]); %keep hole remnants
    csm = bwpropfilt(co, 'MinorAxisLength', [0 12]); % rest is lines
    cf = cc2bw(cbm) + cc2bw(csm);
    cfc = bwconncomp(cf);

    % good bright crack pixels are similar to the dark ones but as they
    % are deeper in the sample, will be straighter
    artbw = testim>-15000 & testim < -10000 & bigROI;
    ac = bwconncomp(artbw);
    anc = bwpropfilt(ac, 'Circularity', [0 0.3]); % more rigorous
```

```
ao = bwpropfilt(anc, 'Orientation', [30 60]);
asm = bwpropfilt(ao, 'MinorAxisLength', [0 12]);

% combine both bright and dark results and keep those with
% centroids in the ROI
combo = cc2bw(asm) + cc2bw(cfc);
combocc = bwconncomp(combo);
numGroups = length(combocc.PixelIdxList);
cents = regionprops(combocc, 'Centroid');

for igroup = 1:numGroups
    gcent = cents(igroup).Centroid;
    if ~inpolygon(gcent(2), gcent(1), roiCorners(:,1), roiCorners(:,2))
        combocc.PixelIdxList{igroup} = [];
        combo = cc2bw(combocc);
    end
end

% use the edge pixels to create line scans perpendicular to the crack
% and since there is only one crack, if multiple components intersect
% with the line scan, keep the one closest to the center of the ROI
for ix = 1:longEdge
    % get the values for the bounding box edges
    xProf = xBegin(ix):xFinal(ix);
    proflen = length(xProf);
    zProf = round(linspace(zBegin(ix), zFinal(ix), proflen));
    intersectdat = [];
    % find components in touching the line
    for iz = 1:proflen
        for ig = 1:numGroups
            locinIm = sub2ind(size(testim), xProf(iz), zProf(iz));
            if any(combocc.PixelIdxList{ig} == locinIm)
                intersectdat = [intersectdat; ig iz];
            end
        end
    end

    % if the line intersects multiple groups, find which they are, keep
    % the ones closest to the center, and delete the others so they
    % aren't intersected again
    if ~isempty(intersectdat) & size(unique(intersectdat(:,1)),1) > 1
        groupInter = unique(intersectdat(:,1));
        numInter = size(groupInter,1);
        ceninterdat = zeros(numInter,2);
        % get the mean for that line
        for jinter = 1:numInter
            ingroup = intersectdat(:,1)==groupInter(jinter);
            ceninterdat(jinter,:) = [groupInter(jinter)...
                mean(intersectdat(ingroup,2))];
        end
        dtocen = abs(shortEdge/2-ceninterdat(:,2));
        [~,closecen] = min(dtocen);
        grouptokeep = ceninterdat(:,1)==ceninterdat(closecen,1);
        badgroups = ceninterdat(~grouptokeep);
        % zero out the points in the non-chosen groups
```

```
            for ibad = 1:length(badgroups)
                if size(combocc.PixelIdxList{badgroups(ibad)},1) < 200
                    combocc.PixelIdxList{badgroups(ibad)} = [];
                    combo = cc2bw(combocc);
                end
            end
        elseif ~isempty(intersectdat)
            intergroup = intersectdat(1,1);
            numgroup = size(combocc.PixelIdxList{intergroup},1);
            zdtocen = abs(mean(intersectdat(:,2))-shortEdge/2);
            if zdtocen > 25 && numgroup < 200
                combocc.PixelIdxList{intergroup} = [];
                combo = cc2bw(combocc);
            end
        end
    end

    % if too few points total in the slize, zero it all out
    pcttrue = length(find(combo))/zDim/xDim;
    if pcttrue <= 1.5e-4
       combo = false(zDim,xDim);
    end

    % save final images
    portim = Tiff(sprintf([saveFolder '%04d_final_pts.tif'],iyslice), 'w');
    tag = struct();
    tag.ImageLength = zDim;
    tag.ImageWidth = xDim;
    tag.Photometric = Tiff.Photometric.MinIsBlack;
    tag.BitsPerSample = 8;
    tag.SamplesPerPixel = 1;
    tag.SampleFormat = Tiff.SampleFormat.UInt;
    tag.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
    tag.Compression = Tiff.Compression.None;
    portim.setTag(tag);
    portim.write(uint8(combo)*255);
    portim.close();
end
```

Step 5: Use ImageJ to Enhance Contrast and normalize for the top and bottom 1% of pixels in the entire image sequence, and save

Step 6: Use .h5 detector metadata to align find the x and z points that align for consecutive tomo scans, and fiducial markers (grain shapes) to find where the overlap in y is for consecutive scans

Step 7: Stack all scans into a single image sequence and keep points with a high incidence across multiple layers

`alignmentTomo.m`

```
groupNum = 1;
```

```
firstScan = 64;
numScans = 3;
scanStep = 5;
% each scan had a bright field and dark field before and after so the true
% scans are spaced 5 apart
scans = firstScan:scanStep:firstScan+scanStep*(numScans-1);
bigDir = 'H:/CHESStransfer/dcdc-gc-02/tomo_';
grainDir = '/final_grains/';
crackDir = '/final_crack_bins/';
saveDir = ['H:/CHESSresults/tomo_s' num2str(groupNum) '/'];
gsDir = 'aligned_grains/';
csDir = 'initial_alignment_crack/';
lims = zeros(3,2,numScans);

% s1, pixel limits from aligning known real locations [x; z; y (slice)]
% in y, keep the slices more centered in their respective scan instead of edges
lims(:,:,1) = [81, 1400; 1, 1320; 87, 700]; % 64
lims(:,:,2) = [44, 1363; 34, 1353; 65, 588]; % 69
lims(:,:,3) = [9, 1328; 32, 1351; 2, 608]; % 74

% finish alignment
totalNumIms = sum(lims(3,2,:)) - sum(lims(3,1,:)) + numScans;
finxDim = lims(2,2,1) - lims(2,1,1)+1;
finzDim = lims(1,2,1) - lims(1,1,1)+1;

orderIms = [lims(3,2,1):-1:lims(3,1,1) ...
    lims(3,2,2):-1:lims(3,1,2) ...
    lims(3,2,3):-1:lims(3,1,3)];
scanOrder = [ones(1, lims(3,2,1) - lims(3,1,1)+1) ...
    2*ones(1, lims(3,2,2) - lims(3,1,2)+1) ...
    3*ones(1, lims(3,2,3) - lims(3,1,3)+1)];

%set metadata for images
tag = struct();
tag.ImageLength = finzDim;
tag.ImageWidth = finxDim;
tag.Photometric = Tiff.Photometric.MinIsBlack;
tag.BitsPerSample = 8;
tag.SamplesPerPixel = 1;
tag.SampleFormat = Tiff.SampleFormat.UInt;
tag.PlanarConfiguration = Tiff.PlanarConfiguration.Chunky;
tag.Compression = Tiff.Compression.None;

% stack initial images
parfor iim = 1:totalNumIms
    iscan = scanOrder(iim);
    iscanNum = scans(iscan);
    islice = orderIms(iim);
    imlims = lims(:,:,iscan);

    % grains
    fullgim = imread([bigDir num2str(iscanNum) '\final_grains\s' ...
        num2str(groupNum) '_t' num2str(iscanNum) ...
        sprintf('_%04d.tif',gslice)]);
```

```
    cropgim = fullgim(imlims(1,1):imlims(1,2),imlims(2,1):imlims(2,2));

    grainim = Tiff(sprintf([saveDir gsDir 'grains_%04d.tif'],iim), 'w');
    grainim.setTag(tag);
    grainim.write(cropgim);
    grainim.close();

    % crack
    fullcim = imread([bigDir num2str(iscanNum) '\final_crack_bins\s' ...
        num2str(groupNum) '_t' num2str(iscanNum) ...
        sprintf('_%04d',islice) '_final_pts.tif']);
    cropcim = fullcim(imlims(1,1):imlims(1,2),imlims(2,1):imlims(2,2));

    crackim = Tiff(sprintf([saveDir csDir 'crack_%04d.tif'], iim), 'w');
    crackim.setTag(tag);
    crackim.write(cropcim);
    crackim.close();
end

finDir = 'clean_crack/';
%s1 hole locations in y, for different treatment
holeConTop = 644;
holeConBot = 704;

% create bins of 100 slices, to be used to filter the center 50 images in
% the bin
startBlockA = 1:50:holeConTop-100;
endBlockA = 100:50:holeConTop;
lastProcBlockA = endBlockA(end) - 25;

startBlockB = flip(totalNumIms-99:-50:holeConBot);
endBlockB = flip(totalNumIms:-50:holeConBot+100);
lastProcBlockB = startBlockB(1)+25;

startBins = [startBlockA,lastProcBlockA+1,startBlockB];
endBins = [endBlockA,lastProcBlockB-1,endBlockB];

% different bin containing images with the hole
cenBin = find(startBins==lastProcBlockA+1);

binSizes = endBins-startBins+1;
% 8 images out of 100 worked the best; larger than many small grains that might
% be counted as noise, but small enough that deflections in the crack away from
% straight can still be captured
binMinReach = 255*8*ones(size(startBins));
%crack wiggles more near crack so needs less intensity to keep
binMinReach(cenBin-1:cenBin+1) = 255*6*ones(1,3);

numBins = length(startBins);
parfor ibin = 1:numBins
    if ibin~=cenBin && ibin~=cenBin-1 && ibin~=cenBin+1
        sizehere = binSizes(ibin);
        startim = startBins(ibin);
        endim = endBins(ibin);
        locims = startim:endim;
```

```
        locstack = uint8(zeros(finzDim,finxDim,sizehere));
        for jim = 1:sizehere
            locstack(:,:,jim) = imread(sprintf([saveDir csDir ...
                'crack_%04d.tif'], locims(jim)));
        end
        sumim = sum(double(locstack),3);
        badlocs = find(sumim<binMinReach(ibin));
        figure,imagesc(sumim)
        for kcheck = 26:75
            useim = locstack(:,:,kcheck);
            useim(badlocs) = uint8(zeros(size(badlocs)));

            cleanim = Tiff(sprintf([saveDir finDir 'crack_%04d.tif'], ...
                locims(kcheck)), 'w');
            cleanim.setTag(tag);
            cleanim.write(useim);
            cleanim.close();
        end
    else
        %at center there is no crack, keep all points
        for kim = lastProcBlockA:lastProcBlockB-1
            origim = imread(sprintf([saveDir csDir 'crack_%04d.tif'],kim));
            bwim = Tiff(sprintf([saveDir finDir ...
                'crack_%04d.tif'], kim), 'w');
            bwim.setTag(tag);
            bwim.write(origim);
            bwim.close();
        end
    end
end
```

Step 8: Find the crack edges and crack front to determine its length

```
crackEdgeFinder.m

pxSize = 0.00148;
crackPts = cell(1,4);
cracklens = zeros(4,8);
numStates = 4; % number of total extension events

for state = 1:4

    % find the images
    crackFolder = (['H:/CHESSresults/tomo_s' num2str(state) ...
        '/clean_crack/']);
    numFiles = size(dir(crackFolder),1) - 2; % extra two in directoy

    firstYcen = -6.15; % position of the hole center at zero load, in mm

    maxpts = [];

    % image limits in all coordinates
    if state == 1
        xVals = -1.07670:pxSize:0.87542;
```

```
    zVals = -1.23358:pxSize:0.71854;
    yVals = -7.07008:pxSize:-4.48896;
    % the images tend to still have noise at the very edges, so make
    % sure that isn't included by stopping the program from including
    % those
    imlims = [1,1200];
    yCen = -6.077; % position of the hole center at this load, in mm
elseif state == 2
    xVals = -1.09002:pxSize:0.83;
    zVals = -1.24838:pxSize:0.70374;
    yVals = -7.88408:pxSize:-4.44012;
    imlims = [584,1916];
    yCen = -6.077;
elseif state == 3
    xVals = -1.04118:pxSize:0.91094;
    zVals = -1.24838:pxSize:0.70374;
    yVals = -8.25652:pxSize:-4.03704;
    imlims = [700,2400];
    yCen = -6.038;
elseif state == 4
    xVals = -1.02638:pxSize:0.91094;
    zVals = -1.23358:pxSize:0.71854;
    yVals = -8.57708:pxSize:-3.54656;
    imlims = [595,3155];
    yCen = -6.067;
end

holeMidBot = yCen + 0.043;
holeMidTop = yCen - 0.043; % where the cones intersect
holeBot = yCen + 0.215;
holeTop = yCen - 0.215; % where the crack first/last appears
yCenShift = yCen-firstYcen;

for ifile = imlims(1):imlims(2)
    % import the binary image
    crackim = imread(sprintf([crackFolder 'crack_%04d.tif'],ifile));
    bwp = find(crackim);
    % if the image isn't just noise, find the crack extrema for the 2d
    % slice and record all values and depths
    if length(bwp)>100
        % points on image with crack
        [imr,imc] = ind2sub([size(crackim,1), size(crackim,2)], bwp);
        crackr = xVals(imr);
        crackc = zVals(imc);
        imy = yVals(ifile);
        % get the min and max in x and corresponding z, and the min and
        % max in z and corresponding x, and calculate average endpoints
        ptA = mean([min(crackr) mean(crackc(crackr==min(crackr))); ...
            mean(crackr(crackc==max(crackc))) max(crackc)]);
        ptB = mean([max(crackr) mean(crackc(crackr==max(crackr))); ...
            mean(crackr(crackc==min(crackc))) min(crackc)]);
        maxpts = [maxpts;ptA imy ptB imy];
    end
end
numLays = size(maxpts,1);
```

```
% due to variation in where the extrema are located on each slice, the
    % actual edge is very jagged and sometimes disappears altogether, so ti
    % must be mitigated
    xA = smooth(maxpts(:,1),0.1);
    zA = smooth(maxpts(:,2),0.1);
    xB = smooth(maxpts(:,4),0.1);
    zB = smooth(maxpts(:,5),0.1);

    % find the points where even after smoothing, the crack edge becomes
    % very jumpy (where only a few points are found); search within the
    % last 50 points marked for simplicity
    jumpAbot = 50 - max([find(abs(diff(flip(zA(1:50))))>0.01,1) ...
        find(abs(diff(flip(xA(1:50))))>0.01,1)]);
    jumpAtop = min([find(abs(diff(zA(numLays-49:numLays)))>0.01,1), ...
        find(abs(diff(xA(numLays-49:numLays)))>0.01,1)]) + numLays - 50;

    jumpBbot = 50 - max([find(abs(diff(flip(zB(1:50))))>0.01,1) ...
        find(abs(diff(flip(xB(1:50))))>0.01,1)]);
    jumpBtop = min([find(abs(diff(zB(numLays-49:numLays)))>0.01,1), ...
        find(abs(diff(xB(numLays-49:numLays)))>0.01,1)]) + numLays - 50;

    % the crack in the original images is rotated, so rotate it into a
    % near-planar position for ease of calculation
    skewR = [cos(pi/6) sin(pi/6) 0;-sin(pi/6) cos(pi/6) 0; 0 0 1];
    skewptsA = (skewR*maxpts(:,1:3)')';
    skewptsB = (skewR*maxpts(:,4:6)')';
    newpts = [skewptsA(:,2:3),skewptsB(:,2:3)];

    % initialize
    newpts(1:jumpAbot,1:2) = nan(jumpAbot,2);
    newpts(jumpAtop:numLays,1:2) = nan(numLays-jumpAtop+1,2);
    newpts(1:jumpBbot,3:4) = nan(jumpBbot,2);
    newpts(jumpBtop:numLays,3:4) = nan(numLays-jumpBtop+1,2);

    % get the last 5 points before the identified jumpiness
    smoothZ = [smooth(newpts(:,1),0.1) newpts(:,2) ...
        smooth(newpts(:,3),0.1) newpts(:,4)];
    lastptsA = smoothZ(jumpAbot+1:jumpAbot+5,1:2);
    lastptsB = smoothZ(jumpBbot+1:jumpBbot+5,3:4);

    % make a parabola connecting the last points with a deeper
    % point identified before the jumpy points were suppressed
    pflast = polyfit([lastptsA(1,1), mean([zA;zB]), lastptsB(1,1)],...
        [lastptsA(1,2), mean(maxpts(1:min([jumpAbot,jumpBbot]), 3)), ...
        lastptsB(1,2)], 2);

    % get points along the prospective crack front and calculate the
    % expected depth along it
    testzbot = linspace(lastptsA(1), lastptsB(1), jumpAbot+jumpBbot);
    testybot = pflast(1) * testzbot.^2 + pflast(2) * testzbot + pflast(3);

    % connect all points into a single front
    smoothZ(1:jumpAbot,1:2) = [testzbot(jumpAbot:-1:1); ...
        testybot(jumpAbot:-1:1)]';
```

```
    smoothZ(1:jumpBbot,3:4) = [testzbot(jumpAbot+1:jumpAbot+jumpBbot); ...
        testybot(jumpAbot+1:jumpAbot+jumpBbot)]';

    % do the same with the crack on the other half of the sample
    numAtop = numLays-jumpAtop + 1;
    numBtop = numLays-jumpBtop + 1;
    firstptsA = smoothZ(jumpAtop-5:jumpAtop-1,1:2);
    firstptsB = smoothZ(jumpBtop-5:jumpBtop-1,3:4);
    pffirst = polyfit([firstptsA(1,1), mean([zA;zB]),firstptsB(1,1)],...
        [firstptsA(1,2), ...
        mean(maxpts(max([jumpAtop,jumpBtop]):numLays,3)), ...
        firstptsB(1,2)], 2);
    testztop = linspace(firstptsA(1),firstptsB(1),numAtop+numBtop);
    testytop = pffirst(1)*testztop.^2 + pffirst(2)*testztop + pffirst(3);
    smoothZ(jumpAtop:numLays,1:2) = [testztop(1:numAtop); ...
        testytop(1:numAtop)]';
    smoothZ(jumpBtop:numLays,3:4) = [testztop(numAtop+numBtop:-1:numAtop+1); ...
        testytop(numAtop+numBtop:-1:numAtop+1)]';

    % combine all points from top and bottom
    allPtsRd = [smoothZ(:,1:2);flip(smoothZ(:,3:4))]; %;smoothZ(1,1:2)];
    crackPts{state} = allPtsRd;

    % calculate the crack lengths and errors from the top and bottom holes
    botmax = max(allPtsRd(:,2));
    % crack error (longer): deepest point identified originally
    botErrmax = max(maxpts(:,3)) - botmax;
    % crack error (shorter): deepest point before jumpiness
    botErrmin = max(jumpAbot,jumpBbot)*pxSize;
    obotlen = botmax - holeBot;
    ibotlen = botmax - holeMidBot;
    topmax = min(allPtsRd(:,2));
    topErrmax = topmax - min(maxpts(:,3));
    topErrmin = min(numAtop,numBtop)*pxSize;
    otoplen = holeTop - topmax;
    itoplen = holeMidTop - topmax;

    cracklens(state,:) = [obotlen ibotlen botErrmax botErrmin ...
        itoplen otoplen topErrmax topErrmin];
end
```

S.2 Crack detection protocol in AlON

Step 1: Find the places in each consecutive tomography scan where overlap begins and average the overlapping scans for each y position due to noisiness on the edges

Step 2: Denoise the stacked images then apply a 3D non-local means filter with automatically estimated parameters to the whole stack, followed by a 2D non-local means filter on each individual slice to remove localized artifacts

Step 3: Classify points in the images as the crack so they can be registered with HEDM grains

fullTomoStitch.m

```
% import tiffs
state = 's1'; % tomography state number

baseFolder = ['K:\tomo_stitches\good\' state '_3dLMN_median_2dLMN\'];
totFiles = length(dir(baseFolder)) -2;
startNum = 10;
endNum = totFiles - 11;
startFileName = Tiff([baseFolder state num2str(startNum,'%04.0f') ...
    '.tif'], 'r');

pxSize = 1.172;

dims = size(read(startFileName));
dimsY = dims(1);
dimsX = dims(2);

% get an image from the end for basic parameters
blankIm = read(Tiff([baseFolder state num2str(totFiles-10,'%04.0f') ...
    '.tif'], 'r'));
% blur it out so no extra lines show up
blurIm = imgaussfilt(blankIm,2);
% use a Hough transform to find the sample edges
[H,theta,rho] = hough(edge(blurIm,'Canny'), 'RhoResolution', 1, ...
    'Theta', -90:0.05:89);
hPks = houghpeaks(H,10);
hLns = houghlines(edge(blurIm), theta, rho, hPks);
lineRh = [hLns.rho];
goodLns = unique(lineRh(1,:));
slopes = zeros(1,4);
intercs = zeros(1,4);
% sometimes a line is broken up but has the same angle, use only one
for iline = 1:length(goodLns)
    loc = find(lineRh==goodLns(iline));
    if length(loc) > 1
        lineLens = zeros(1,length(loc));
        for mark = 1:length(loc)
            y1 = hLns(loc(mark)).point1(1);
            x1 = hLns(loc(mark)).point1(2);
            y2 = hLns(loc(mark)).point2(1);
            x2 = hLns(loc(mark)).point2(2);
            lineLens(mark) = sqrt((y2 - y1)^2 + (x2 - x1)^2);
        end
        longest = find(lineLens == max(lineLens));
        goodNum = loc(longest);
    else
        goodNum = loc;
    end

    % store slope and intercept data
```

```
    lineData = hLns(goodNum);
    y1 = lineData.point1(1);
    x1 = lineData.point1(2);
    y2 = lineData.point2(1);
    x2 = lineData.point2(2);
    slopes(iline) = (y2 - y1) / (x2 - x1);
    intercs(iline) = y2 - slopes(iline) * x2;
end

% make sure no vertical lines
goodslopes = zeros(1,4);
goodintercs = zeros(1,4);
kline = 1;
for jlinecheck = 1:length(slopes)
    if abs(slopes(jlinecheck)) ~= Inf
        goodslopes(kline) = slopes(jlinecheck);
        goodintercs(kline) = intercs(jlinecheck);
        kline = kline+1;
    end
end
slopes = goodslopes;
intercs = goodintercs;

% get the corners by intersecting the lines
[sortSlopes,sortOrder] = sort(slopes);
sortIntercs = intercs(sortOrder);
cornerPts = zeros(2,4);
fillPt = 1;
for ipoint = 1:2
    usePoint = sortOrder(ipoint);
    for secondary = 3:4
        secondPoint = sortOrder(secondary);
        xPoint = (intercs(secondPoint) - intercs(usePoint)) / ...
            (slopes(usePoint) - slopes(secondPoint));
        % if points are outside the image dimensions, set them to within
        % the image
        if xPoint < 1
            cornerPts(1,fillPt) = 1;
        elseif xPoint > dimsX
            cornerPts(1,fillPt) = dimsX;
        else
            cornerPts(1,fillPt) = round(xPoint);
        end
        yPoint = slopes(usePoint) * xPoint + intercs(usePoint);
        if yPoint < 1
            cornerPts(2,fillPt) = 1;

        elseif yPoint > dimsY
            cornerPts(2,fillPt) = dimsY;
        else
            cornerPts(2,fillPt) = round(yPoint);
        end
        fillPt = fillPt+1;
    end
end
```

```
% use the corners to calculate final slopes
miny = min(cornerPts(2,:));
xatminy = cornerPts(1, cornerPts(2,:)==miny);
maxy = max(cornerPts(2,:));
xatmaxy = cornerPts(1, cornerPts(2,:)==maxy);
minx = min(cornerPts(1,:));
yatminx = cornerPts(2, cornerPts(1,:)==minx);
maxx = max(cornerPts(1,:));
yatmaxx = cornerPts(2, cornerPts(1,:)==maxx);

mA = (miny - yatminx) / (xatminy - minx);
bA = miny - mA*xatminy;
mB = (yatmaxx - miny) / (maxx - xatminy);
bB = yatmaxx - mB * maxx;
mC = (maxy - yatminx) / (xatmaxy - minx);
bC = maxy - mC * xatmaxy;
mD = (yatmaxx  -maxy) / (maxx - xatmaxy);
bD = yatmaxx - mD * maxx;

% find the edge points in x
xRange = minx:maxx;
xRangeA = minx:xatminy;
xRangeB = xatminy+1:maxx;
xRangeC = minx:xatmaxy;
xRangeD = xatmaxy+1:maxx;

% calculate the edge points in z
upperLimA = round(mA * xRangeA + bA);
upperLimB = round(mB * xRangeB + bB);
upperLim = [upperLimA upperLimB];
lowerLimC = round(mC * xRangeC + bC);
lowerLimD = round(mD  *xRangeD + bD);
lowerLim = [lowerLimC lowerLimD];

% set threshold limits
whiteMask = 1;
blackMask = 1.25;
strongBlackMask = 1.5;
mostBlackMask = 2;

% initialize storage
peaks = cell(1,totFiles);
chasms = cell(1,totFiles);
loners = cell(1,totFiles);
valleys = cell(1,totFiles);


%% initial point identification

parfor ilayer = startNum:endNum
    % set initial points in the layer to be blank
    imStack = zeros(dimsY,dimsX);
    whiteStackWgts = zeros(dimsY,dimsX);
    whiteStack = false(dimsY,dimsX);
```

```
blackStack = true(dimsY,dimsX);
strongBlackStack = blackStack;
mostBlackStack = blackStack;

% make a localized stack of 11 images for each center
for jline = 1:11
    imLay = jline-6;
    imStack(:,:,jline) = double(read(Tiff([baseFolder state ...
        num2str(ilayer - imLay,'%04.0f') '.tif'], 'r')));
end

im2Use = imStack(:,:,6); % center image

% get the overall stack properties
tomoMean = mean(imStack, 'all');
tomoStd = std(imStack, 0, 'all');

% for each value in x, find the corresponding z values marking the
% sample edges, then using the environment around a point (x,z) find if
% it matches any of the thresholds
for kx = minx:maxx
    xLoc = find(xRange==kx);
    zLims = [upperLim(xLoc), lowerLim(xLoc)];
    zStart = min(zLims);
    zEnd = max(zLims);

    % relevant region to be used for calculations can't extend past
    % image limits, so set appropriate limits
    if kx<21
        krangebot = 1;
        krangetop = kx + 20;
    elseif kx>dimsX-21
        krangebot = kx - 20;
        krangetop = dimsX;
    else
        krangebot = kx - 20;
        krangetop = kx + 20;
    end

    for jz = zStart:zEnd
        % limits in z
        if jz<21
            jrangebot = 1;
            jrangetop = jz + 20;
        elseif jz>dimsY-21
            jrangebot = jz - 20;
            jrangetop = dimsY;
        else
            jrangebot = jz - 20;
            jrangetop = jz + 20;
        end

        % find the mean and standard deviation in the region around
        % each point to modify the overall image values with
        localSubset = imStack(krangebot:krangetop, ...
```

```
            jrangebot:jrangetop, :);
        localMean = mean(localSubset, 'all');
        localStd = std(localSubset, 0, 'all');

        % let the mean be the average of image and local means
        useMean = 0.5 * localMean + 0.5 * tomoMean;
        % let the standard deviation rely more on the local value than
        % overall
        useStd = 0.75 * localStd + 0.25 * tomoStd;

        % the bright parts are an upper threshold, find if matching the
        % requirement
        whiteMin = useMean + whiteMask * useStd;
        if im2Use(kx,jz) > whiteMin
            whiteStackWgts(kx,jz) = (65535 - im2Use(kx,jz)) / ...
                (65535 - whiteMin);
            whiteStack(kx,jz) = true;
        end

        % apply the three different dark thresholds
        if imStack(kx,jz) < (useMean - blackMask * useStd)
            blackStack(kx,jz) = false;
        end
        if imStack(kx,jz) < (useMean - strongBlackMak * useStd)
            strongBlackStack(kx,jz) = false;
        end
        if imStack(kx,jz) < (useMean - mostBlackMask * useStd)
            mostBlackStack(kx,jz) = false;
        end
    end
end

% initialize the types of point being looked for
layPeaks = [];
layChasms = [];
layValleys = [];
layLoners = [];

% again look for points
for ax = minx:maxx

    xLoc = find(xRange==ax);
    zLims = [upperLim(xLoc), lowerLim(xLoc)];
    zStart = min(zLims);
    zEnd = max(zLims);

    for bz = zStart:zEnd

        % count a bright point as a peak if it has dark shadows on
        % either side
        if whiteStack(ax,bz) == 1 && ...
                any(~blackStack(ax, bz-10:bz-1)) && ...
                any(~blackStack(ax, bz+1:bz+10))
            layPeaks = [layPeaks;bz,ax];
        end
```

```
            % three different criteria being checked for dark points
            if blackStack(ax,bz) == 0
                leftCheck = 0;
                rightCheck = 0;
                leftBright = 0;
                rightBright = 0;
                backwardWhiteWeights = 10:-1:1;

                % find out if there's a brighter peak on either side of a
                % dark point within 10 pixels
                for range = 1:10
                    if whiteStackWgts(ax,bz-range) ~= 0
                        leftCheck = leftCheck + 1;
                        leftBright = leftBright + ...
                            whiteStackWgts(ax, bz-range) * ...
                            backwardWhiteWeights(range);
                    end
                    if whiteStackWgts(ax,bz+range) ~= 0
                        rightCheck = rightCheck + 1;
                        rightBright = rightBright + ...
                            whiteStackWgts(ax, bz+range) * ...
                            backwardWhiteWeights(range);
                    end
                end

                % if bright neighbors were found, weight by how wide they
                % are
                if leftCheck > 0
                    leftBright = leftBright / leftCheck;
                end
                if rightCheck > 0
                    rightBright = rightBright / rightCheck;
                end

                % if both peaks are there, they don't have to be that big
                % for the point to be recorded as a crack. One sufficiently
                % wide peak beside the dark point is also good. Then very
                % dark points are always recorded as crack points
                if leftCheck >= 4 && rightCheck >= 4
                    layValleys = [layValleys;bz,ax];
                elseif (leftCheck >= 8 || rightCheck >= 8) && ...
                        strongBlackStack(ax,bz) == 0
                    layChasms = [layChasms;bz,ax];
                else
                    if mostBlackStack(ax,bz) == 0
                        layLoners = [layLoners;bz,ax];
                    end
                end
            end
        end
    end
    peaks{ilayer} = layPeaks;
    valleys{ilayer} = layValleys;
    chasms{ilayer} = layChasms;
```

```
    loners{ilayer} = layLoners;
end

%% combine all the dark points and bright points from different y slices

allpts = [];
allpks = [];
for ilayer = startNum:endNum
    if ~isempty(valleys{ilayer})
        vecpts = [valleys{ilayer}, ilayer * ones(size(valleys{ilayer},1), 1)];
        allpts = [allpts; vecpts(:,1), vecpts(:,2), vecpts(:,3)];
    end
    if ~isempty(chasms{ilayer})
        vecpts = [chasms{ilayer}, ilayer * ones(size(chasms{ilayer},1), 1)];
        allpts = [allpts; vecpts(:,1), vecpts(:,2), vecpts(:,3)];
    end
    if ~isempty(loners{ilayer})
        vecpts = [loners{ilayer}, ilayer * ones(size(loners{ilayer},1), 1)];
        allpts = [allpts; vecpts(:,1), vecpts(:,2), vecpts(:,3)];
    end
    if ~isempty(peaks{ilayer})
        vecpts = [peaks{ilayer}, ilayer * ones(size(peaks{ilayer},1), 1)];
        allpks = [allpks; vecpts(:,1), vecpts(:,2), vecpts(:,3)];
    end
end

%% fill in gaps, reduce noise, and check for bright regions
% indicating a narrow crack

numpts = size(allpts,1);

% find all unique(x,z) points the crack was found on in any layer
[xypts,~,~] = unique(allpts(:,1:2), 'rows');
numUniqueXY = size(xypts, 1);
morepts = cell(1,numUniqueXY);

% some points recorded as peaks are just artifacts, but some represent the
% crack when it was very thin
parfor ipt = 1:numUniqueXY

    % create a line of all marked points for each (x,z) point
    parXY = xypts(ipt,:);
    linecrack = sort(allpts(allpts(:,1)==parXY(1) & ...
        allpts(:,2)==parXY(2), 3));
    whiteArt = sort(allpks(allpks(:,1)==parXY(1) & ...
        allpks(:,2)==parXY(2), 3));
    parNum = 0;
    parCrack = [];

    % crack needs to be sufficiently long not to be nosie
    if length(linecrack) > 5
        % the crack might move away and back to a point as the crack
        % changes orientation
        spacing = diff(linecrack);
        jumps = find(spacing>1);
```

```
% if there are not too many jumps, analyze
if length(jumps) < length(linecrack)/3
    % move jump by jump to test crack locations
    check = 1;
    while ~isempty(jumps)
        jumploc = jumps(check);

        % If the jump happens very early on, the first point is
        % likely spurious; if it happens very late, the last point
        % is also likely spurious. If the jump is small, there is
        % likely a blurry or narrow crack; fill it in. Otherwise,
        % split the crack into two for analysis of the first part
        % and send the rest back for the same set of tests
        if jumploc <= 2
            linecrack = linecrack(jumploc+1:length(linecrack));
        elseif jumploc >= length(spacing) - 2
            linecrack = linecrack(1:jumploc);
            break
        elseif spacing(jumploc) <= 10
            endpt = jumploc;
            startpt = jumploc+1;
            linecrack = [linecrack(1:endpt); ...
                (linecrack(endpt)+1:linecrack(startpt)-1)'; ...
                linecrack(startpt:length(linecrack))];
        else
            crackparta = linecrack(1:jumploc);
            if length(crackparta) > 5
                for ptentryA = 1:length(crackparta)
                     parCrack(parNum+ptentryA,:) = [parXY(1), ...
                         parXY(2), linecrack(ptentryA)];
                end
                parNum = parNum+length(linecrack);
            end
            crackpartb = linecrack(jumploc+1:length(linecrack));
            linecrack = crackpartb;
        end
        spacing = diff(linecrack);
        jumps = find(spacing>1);
    end

    % if the remaining length of crack after all spaces have been
    % accounted for is long enough, keep it
    if length(linecrack)>5
        for ptentryLeft = 1:length(linecrack)
           parCrack(parNum+ptentryLeft,:) = [parXY(1), ...
               parXY(2), linecrack(ptentryLeft)];
        end
        parNum = parNum+length(linecrack);
    end

% if there are a lot of jumps relative to the length of the
% identified points along the line, see if the part before the
% noise is long enough and keep, otherwise, discard
else
```

```
            linecrack = linecrack(1:jumps(1));
            if length(linecrack) > 5
                for ptentryStart = 1:length(linecrack)
                    parCrack(parNum+ptentryStart,:) = [parXY(1), ...
                        parXY(2), linecrack(ptentryStart)];
                end
                parNum = parNum+length(linecrack);
            end
        end

        % If there are identified bright points for that point, look at
        % those points directly ahead of the final portion of the crack. If
        % they don't begin too much later, add them to the crack
        if ~isempty(whiteArt)
            extraWt = whiteArt(whiteArt>max(linecrack));
            if ~isempty(extraWt) && (min(extraWt)-max(linecrack))<4
                possExt = find(whiteArt > endNum-6);
                for addpt = 1:length(possExt)
                    parCrack(parNum+addpt,:) = [parXY(1), parXY(2), ...
                        possExt(addpt)];
                    parNum = parNum+1;
                end
            end
        end
    end
    morepts{ipt} = parCrack;
end

%% unpack the cell

crackpts = [];
crackFrontPts = [];
for ipt = 1:numUniqueXY
    if ~isempty(morepts{ipt})
        vecpts = morepts{ipt};
        crackpts = [crackpts; vecpts(:,1), vecpts(:,2), vecpts(:,3)];

        % deepest point for a given (x,z)
        topPt = find(vecpts(:,3)==max(vecpts(:,3)));
        crackFrontPts = [crackFrontPts; vecpts(topPt,:)];
    end
end

writematrix(crackpts, ['sam1_' state '_final_crack_pts.csv'])
writematrix(crackFrontPts, ['sam1_' state '_final_crack_front_pts.csv'])

%% rotated version of the crack plane for depth analysis

ang = -19.6*pi/180; % rotation angle for given sample
R = [cos(ang) -sin(ang) 0;sin(ang) cos(ang) 0;0 0 1];
crackRotated = crackpts*R;
xLimCrack = crackRotated(crackRotated(:,1)>200 & ...
    crackRotated(:,1)<1000,:);
goodAreaCrack = round(xLimCrack(xLimCrack(:,2)>730 & ...
    xLimCrack(:,2)<1900,:));
```

```
yLines = unique(goodAreaCrack(:,2));
maxinLine = zeros(length(yLines),1);
for iline = 1:length(yLines)
    yLine = yLines(iline);
    yLineData = goodAreaCrack(goodAreaCrack(:,2)==yLine,:);
    maxinLine(iline) = max(yLineData(:,3));
end
writematrix([yLines,maxinLine],['sam1_' state '_crack_front_line_rot.csv'])
```

Step 4: Calculate the crack front depth

```
plotSam1Fronts.m (and plotSam4Fronts.m)

% read in the front points (not all states had tomography)
baseStr = 'sam1_s0_crack_front_line_rot.csv';

pxSize = 1.172;

sOneFront = readmatrix(replace(baseStr,'0','1'))*pxSize;
sTwoFront = readmatrix(replace(baseStr,'0','2'))*pxSize;
sSixFront = readmatrix(replace(baseStr,'0','6'))*pxSize;
sNineFront = readmatrix(replace(baseStr,'0','9'))*pxSize;
sTwelveFront = readmatrix(replace(baseStr,'0','12'))*pxSize;
sThirteenFront = readmatrix(replace(baseStr,'0','13'))*pxSize;

% get the points across the hole (shifted because rotation wasn't around 0),
% consistent across states, as x
frontLocs = sSixFront(:,1)-855.5;

% use the angle of the cylindrical hole sections to get its location at all
% points (x,y)
mA = -10.377;
b = 544.29;
mC = 9.2658;

holeYA = frontLocs(1:457);
holeXA = (holeYA-b)/mA;
holeYC = frontLocs(458:1171);
holeXC = (holeYC-b)/mC;
fullholeX = [holeXA;holeXC];

% for each front, find its stats
[minOne, maxOne, meanOne, ptsOne] = getFrontData(sOneFront, 53);
[minTwo, maxTwo, meanTwo, ptsTwo] = getFrontData(sTwoFront, 80);
[minSix, maxSix, meanSix, ptsSix] = getFrontData(sSixFront, 65);
[minNine, maxNine, meanNine, ptsNine] = getFrontData(sNineFront, 68);
[minTwelve,maxTwelve,meanTwelve,ptsTwelve] = getFrontData(sTwelveFront, 73);
[minThirteen,maxThirteen,meanThirteen, ...
    ptsThirteen] = getFrontData(sThirteenFront, 69);

function [minD,maxD,meanD,ptsOI] = getFrontData(front,offset)
% get stats about the crack front; offset tells how much the hole position
% changed between datasets

% smooth it over a small region so any spurious points that made it out
```

```
% of the last filtering (usually in the center where ring artifacts often
% made persistent shadows) go away, also for nicer plotting
newFront = smooth(front(:,2)-offset*1.172,15);

% calculate its length from the hole at each x position
minD = min(newFront(100:1171))-fullholeX(newFront==min(newFront));
maxD = max(newFront(100:1171))-fullholeX(newFront==max(newFront));
meanDif = abs(newFront-mean(newFront));
meanD = mean(newFront(100:1171))-fullholeX(meanDif==min(meanDif));

% report a few points along the front to compare apart from overall depth
ptsOI = newFront([109,315,374,534,723,910,1020]);

end

end
```

S.3 Tessellation creation protocol in AlON

voronoiNoGroundTruth.m

```
state = 's1'; % state name
sam = 'sam4'; % sample name
displace = 1000; % value in um of center of first scan
confLim = 0.8; % minimum confidence to keep in tessellation
allGrains = readmatrix(['K:/' sam '_final_data/ff/' sam '_' state ...
    '_ff_sam_ao_individual_grain_stats.csv']);
grains = allGrains(allGrains(:,6)>=0.8, :);

numGrains = size(grains,1);


sideLen = 200; % number of voxels in each of x, y, and z

ycom = -grains(:,3) + displace; % adjust y to match tomography orientation
xyz = [grains(:,4), ycom, grains(:,2)]; % adjust x and z to match tomo
Eul = grains(:,25:27);
EulOne = Eul(:,1);
EulTwo = Eul(:,2);
EulThree = Eul(:,3);
Stress = grains(:,16:21);
IDs = grains(:,1);
rads = grains(:,5);

totRadV = sum(4 * pi * (rads.^3) / 3);

lims = [min(xyz) - 50; max(xyz) + 50]; % set extrema
Xpts = linspace(lims(1,1), lims(2,1), sideLen); % create voxel nodes
Ypts = linspace(lims(1,2),l ims(2,2), sideLen);
Zpts = linspace(lims(1,3), lims(2,3), sideLen);
[X,Y,Z] = ndgrid(Xpts, Ypts, Zpts);


% initialize for speed
grainDists = zeros(sideLen, sideLen, sideLen, numGrains);

for indgrain = 1:numGrains
```

```
    graincom = xyz(indgrain,:);
    grainrad = rads(indgrain);
    % calculate the distances of grain centroids to nodes
    grainDblock = sqrt((graincom(1) - blockCOM(1))^2 + ...
        (graincom(2) - blockCOM(2))^2 + (graincom(3) - blockCOM(3))^2);
    % create the power-distance
    grainDweight = 0.25 * grainDblock / maxD + 1.25;
    % adjust the distance with the power-distance
    grainDists(:, :, :, indgrain) = sqrt((X - graincom(1)).^2 - ...
        (Y - graincom(2)).^2 + (Z - graincom(3)).^2) + ...
        grainDweight * grainrad;
end

% find the grain with the minimum distance to each node
[~, minDists] = min(grainDists, [], 4);


% assign grain IDs; can use Euler or stress to create other representations
tessMatch = reshape(IDs(minD
```