```
%% IMAGE BASED INERTIAL TEST (IBIT): Smoothing Sweep Monte-Carlo↙
Simulator
% Author: Lloyd Fletcher
% PhotoDyn Group, University of Southampton
% http://photodyn.org/
% Date Created: 30th Aug. 2017
% Date Edited: 8th Aug. 2019 - v1.0r
%
% Performs a parametric sweep of smoothing kernels to determine optimal
% processing parameters for the IBII test. After running this sweep the
% data can be displayed using the 'AnalyseSweep' program.
%
% The following papers describe the IBII method:
%[1] L. Fletcher, J. Van-Blitterswyk, F. Pierron, A Novel Image-Based
%   Inertial Impact Test (IBII) for the Transverse Properties of
%   Composites at High Strain Rates, J. Dynamic Behavior Mater. (2019).
%   doi:10.1007/s40870-019-00186-y.
%[2] L. Fletcher, F. Pierron, An image-based inertial impact (IBII) test
%   for tungsten carbide cermets, J. Dynamic Behavior Mater. 4 (2018)
%   481-504. doi:10.1007/s40870-018-0172-4.
%[3] J. Van Blitterswyk, L. Fletcher, F. Pierron, Image-based inertial
%   impact test for composite interlaminar tensile properties, J.
%   Dynamic Behavior Mater. 4 (2018) 543-572. doi:10.1007/s40870-018-↙
0175-1.
%
% This work is licensed under the Creative Commons Attribution-
% NonCommercial-ShareAlike 4.0 International License. To view a copy of
% this license, visit http://creativecommons.org/licenses/by-nc-sa/4.0/↙
or
% send a letter to Creative Commons, PO Box 1866, Mountain View, CA↙
94042,
% USA.
%
% If there are any issues with this code please contact:
% Lloyd Fletcher: l.c.fletcher@soton.ac.uk / lloydcolinfletcher@gmail.↙
com

clc
clear all
close all
```

```
fprintf('---------------------------------------------------------------↙
\n')
fprintf('IMAGE DEFORMATION SMOOTHING SWEEP SIMULATOR - v1.0r\n')
fprintf('---------------------------------------------------------------↙
\n')

%% INITIALISE: Add path for processing functions
% Add the path for the grid method code and other useful functions:
funcPath = [pwd,'\Functions\'];

% If the default path is not found we should find it
if exist(funcPath,'file') ~= 7
    hWarn = warndlg('Folder for global processing functions not↙
found','Function folder not found');
    waitfor(hWarn);
    funcPath = uigetdir(pwd,'Locate Global Processing Function Folder');
end
addpath(funcPath);
addpath([funcPath,'GridMethodToolbox\']);

%% LOAD DATA FILES - .tiff images
% NOTE: the deformed grid image files can be generated from FE data↙
using
% the 'GridImageDeformation' program. Samples images are provided in the
% same directory as this code file.

fprintf('Loading reference image from the selected test data folder.\n')
hardCodePath = true;
if ~hardCodePath
    [imageFile,imagePath] = uigetfile({'*.*','All Files'},'Select the↙
first image in the sequence');
else
    %imagePath = [pwd,'\SmoothingSweepData_Isotropic\'];
    imagePath = [pwd,'\SmoothingSweepData_ReducedOrtho\'];
    imageFile = 'DefGridImage_001.tiff';
end

%% INITIALISE: User Defined Simulation Sweep Options
fprintf('Initialising sweep variables and assigning kernels to workers.↙
\n')
```

```
%
-----------------------------------------------------------------------
--
% Simulation Options
% NOTE: Test flag is used to test the code by running only a few
iterations
% for few cases.
%///////////////////////
simOpts.testFlag = false;
simOpts.numWorkers = 18;
%///////////////////////
if simOpts.testFlag
    simOpts.numSimsPerConfig = 3;
    simOpts.numWorkers = 2;
else
    simOpts.numSimsPerConfig = 30;
end

% Scale factors for comparing frame rates and pixel array sizes with
% similar smoothing kernels
temporalScaleFactor = 1;
spatialScaleFactor = 1;
% Flag for suppressing console output from IBII processing functions
printToCons = false;

%
-----------------------------------------------------------------------
--
% Smoothing Sweep Parameters
sKernels = [11,21,31,41,51,61];
tKernels = [5,11,15,21,25,31];
%sKernels = [11,15,21,25,31,35,41,45,51,55];
%tKernels = [5,7,9,11,13,15,17,19,21,25,31];
tOrder = 3;
if simOpts.testFlag
    sKernels = [41];
    tKernels = [11];
end

%
-----------------------------------------------------------------------
```

```
--
% Image Noise Parameters
imageNoiseSweep.addNoise = true;
imageNoiseSweep.pcNoise = 0.4; % Normally 0.35-0.5% for HPVX
imageNoiseSweep.bits = 16;
imageNoiseSweep.convToUInt16 = true;

%% INITIALISE: Image Mask and Kernel Worker Assignment

%↙
-------------------------------------------------------------------------↙
--
% Specimen Location for Masking Grid Images
locPath = imagePath;
locFile = 'specimenLocation.mat';
if exist([locPath,locFile],'file') == 2
    load([locPath,locFile]);
else
    waitfor(warndlg({'Specimen location file not found.','Using default↙
specimen location.'},'Warning!'))
    specimenLoc.bottomLeft = [14,5];
    specimenLoc.topRight = [398,246];
end

%↙
-------------------------------------------------------------------------↙
--
% Assign smoothing kernels to each worker

% Scale based on frame rate or number of pixels
sKernels = round((sKernels*spatialScaleFactor-1)/2)*2+1;
tKernels = round((tKernels*temporalScaleFactor-1)/2)*2+1;
% Add the 'no smoothing' case
sKernels = [0,sKernels];
tKernels = [0,tKernels];

% Create a cell array of all different combinations of smoothing
ii = 1;
for isf = 1:length(sKernels)
    for itf = 1:length(tKernels)
        sweepKernels{ii}.spatialKernel = sKernels(isf);
```

```
        sweepKernels{ii}.temporalKernel = tKernels(itf);
        sweepKernels{ii}.temporalOrder = tOrder;
        ii = ii+1;
    end
end

% Work out how many kernel combinations are going to be assigned to each
% worker and distribute them evenly.
totalKernels = length(sweepKernels);
intKernsPerWorker = floor(totalKernels/simOpts.numWorkers);
kernelsPerWorker(1:simOpts.numWorkers) = intKernsPerWorker;
remainingKerns = mod(totalKernels,simOpts.numWorkers);

if remainingKerns > 0
    for ii = 1:remainingKerns
        kernelsPerWorker(ii) = kernelsPerWorker(ii)+1;
    end
end

kernelsIndsPerWorker = [];
check = cumsum(kernelsPerWorker);
check = [0,check];
for ii = 1:(length(check)-1)
    kernelsIndsPerWorker((check(ii)+1):check(ii+1)) = ii;
end
maxKernelsPerWorker = max(kernelsPerWorker);

%% RUN THE MONTE CARLO SIMULATOR
% INITIALISE - Load Processing Parameter Data Structures
fprintf('Loading the processing parameters file.\n')
% Initialise material/disp variable to avoid matlab conflict
material = [];
disp = [];
pos = [];
pulse = [];

initPath = imagePath;
initFile = 'processingParameters.mat';
if exist([initPath,initFile],'file') ~= 2
    hWarn = warndlg('Processing parameter file does not↙
exist.','Processing parameters not found');
```

```
    waitfor(hWarn);
    [initFile,initPath,~] = uigetfile('*.mat','Locate processing↙
parameter file');
end
% Load the processing parameters from file
load([initPath,initFile])

%↙
-----------------------------------------------------------------------↙
--
% Overwrite the image noise structure to be controlled by master program
imageNoise = imageNoiseSweep;

%↙
-----------------------------------------------------------------------↙
--
% VFM options for parametric sweep
% Allows the impact edge cut to be updated for each iteration based on↙
the
% spatial smoothing kernel size. Otherwise it is fixed at the grid pitch
% size.
VFOpts.updateImpactEdgeCut = false;

%% MAIN SIMULATION LOOP
fprintf('Starting the parallel pool with %i workers.\n',simOpts.↙
numWorkers)
delete(gcp('nocreate'));
% Create parallel pool and set 'time out' for 8 hours
parpool('local',simOpts.numWorkers,'IdleTimeout',8*60);

spmd (simOpts.numWorkers)

% Find which worker this is and assign it some kernels to sweep
for ww = 1:simOpts.numWorkers
    if ww == labindex
       workerKernels = sweepKernels(kernelsIndsPerWorker == labindex);
    end
end

iter = 1;
totalIters = length(workerKernels)*simOpts.numSimsPerConfig;
```

```
% Pre-allocation for speed
simPulsePeakStressVec = zeros(1,simOpts.numSimsPerConfig);
simQxxAvgVecSG = zeros(1,simOpts.numSimsPerConfig);
simQxxAvgVecVFMan = zeros(1,simOpts.numSimsPerConfig);
simQxxAvgVecVFOpt = zeros(1,simOpts.numSimsPerConfig);
simQxyAvgVecVFMan = zeros(1,simOpts.numSimsPerConfig);
simQxyAvgVecVFOpt = zeros(1,simOpts.numSimsPerConfig);
simQxxRangeMinVecSG = zeros(1,simOpts.numSimsPerConfig);
simQxxRangeMaxVecSG = zeros(1,simOpts.numSimsPerConfig);
simQxxRangeSDVecSG = zeros(1,simOpts.numSimsPerConfig);
simQxxRangeMinVecVFMan = zeros(1,simOpts.numSimsPerConfig);
simQxxRangeMaxVecVFMan = zeros(1,simOpts.numSimsPerConfig);
simQxxRangeSDVecVFMan = zeros(1,simOpts.numSimsPerConfig);
simQxxRangeMinVecVFOpt = zeros(1,simOpts.numSimsPerConfig);
simQxxRangeMaxVecVFOpt = zeros(1,simOpts.numSimsPerConfig);
simQxxRangeSDVecVFOpt = zeros(1,simOpts.numSimsPerConfig);

% MAIN simulation loop
for kernNum = 1:length(workerKernels)
    for simNum = 1:simOpts.numSimsPerConfig
        tic
        fprintf('\n')
        fprintf↙
('---------------------------------------------------------------\n')
        fprintf('MAIN SIM LOOP - ITERATION %i of %i\n',iter,totalIters)
        fprintf↙
('---------------------------------------------------------------\n')
        fprintf('Worker number %i \n',labindex)
        fprintf('Monte Carlo Simulation %i of %i\n',simNum,simOpts.↙
numSimsPerConfig)
        if iter > 1
            fprintf('\n')
            fprintf('Previous iteration took: %.2f seconds\n',iterTime↙
(iter-1))
            fprintf('Average iteration time is: %.2f seconds\n',mean↙
(iterTime))
            fprintf('Total time elapsed for this worker: %.3f hours\n',↙
sum(iterTime)/(60*60))
            fprintf('\n')
            fprintf('Projected total time for this worker: %.2f↙
```

```
hours\n',...
                (mean(iterTime)*totalIters)/(60*60))
            fprintf('Projected time to completion for this worker: %.3f↙
hours\n',...
                (mean(iterTime)*(totalIters-iter+1))/(60*60))
            fprintf('Projected total simulation time is: %.2f↙
hours\n',...
                (mean(iterTime)*maxKernelsPerWorker*simOpts.↙
numSimsPerConfig)/(60*60))
            fprintf('\n')
        end

        %↙
---------------------------------------------------------------------------↙
--
        % Update the smoothing options for this pass
        % If the smoothing kernal size is zero, don't smooth
        if workerKernels{kernNum}.spatialKernel == 0
            smoothingOpts.spatialSmooth = false;
        else
            smoothingOpts.spatialSmooth = true;
        end

        if workerKernels{kernNum}.temporalKernel == 0
            smoothingOpts.WATempSmooth = false;
            smoothingOpts.FFTempSmooth = false;
        else
            smoothingOpts.WATempSmooth = true;
            smoothingOpts.FFTempSmooth = true;
        end

        % Store the smoothing kernal parameters in the options structure
        smoothingOpts.spatialKernal = [workerKernels{kernNum}.↙
spatialKernel,workerKernels{kernNum}.spatialKernel];
        smoothingOpts.WATemporalKernal = [workerKernels{kernNum}.↙
temporalKernel,workerKernels{kernNum}.temporalOrder];
        smoothingOpts.FFTemporalKernal = [workerKernels{kernNum}.↙
temporalKernel,workerKernels{kernNum}.temporalOrder];

        % Update the extrapolation window
        extrapOpts.strainPx = grid.pxPerPeriod+floor(workerKernels↙
```

```
{kernNum}.spatialKernel/2);
        extrapOpts.FFStrainPx = grid.pxPerPeriod+floor(workerKernels↙
{kernNum}.spatialKernel/2);

        % If we are taking out the impact edge from the VF calcs update
        % this variable here
        if VFOpts.updateImpactEdgeCut && globalOpts.smoothingOn
            VFOpts.cutEdgePx = grid.pxPerPeriod+floor(smoothingOpts.↙
spatialKernal(1)/2)+1;
        else
            VFOpts.cutEdgePx = grid.pxPerPeriod+1;
        end

        fprintf('Spatial smoothing kernal: [%i,%i]\n',smoothingOpts.↙
spatialKernal(1),smoothingOpts.spatialKernal(2))
        fprintf('Temporal smoothing kernal: [%i,%i]\n',smoothingOpts.↙
WATemporalKernal(1),smoothingOpts.WATemporalKernal(2))
        fprintf↙
('---------------------------------------------------------------\n')

        %↙
--------------------------------------------------------------------------↙
--
        % GRID METHOD PROCESSING
        %↙
--------------------------------------------------------------------------↙
--
        % Add noise and process images with the grid method
        % Process the image sequence with the grid method toolbox
        [grid,pos,disp] = func_gridMethodProcessingImageDef(imagePath,↙
imageFile,...
            grid,gridMethodOpts,specimenLoc,imageNoise);

        %↙
--------------------------------------------------------------------------↙
--
        % Update Geometry and Number of Frames Based on Displacement↙
Matrix Size
        [specimen,grid] = func_updateSpecGeom(specimen,grid,disp);

        % Currently the rotations are unused so remove them to save RAM
```

```
        disp = rmfield(disp,'rot');

        %↙
------------------------------------------------------------------------↙
--
        % KINEMATIC FIELD CALCULATION
        %↙
------------------------------------------------------------------------↙
--
        % Load the Reference Image and Determine Where the Free Edge is
        [freeEdge,specimen,disp] = func_getFreeEdge(globalOpts.↙
hardCodeFreeEdge,...
            imagePath,imageFile,specimen,disp,printToCons);

        %↙
------------------------------------------------------------------------↙
--
        % Smooth and Calculate Strain
        [~,strain,~] = func_smoothCalcStrain(globalOpts,pos,time,...
            grid,disp,smoothingOpts,extrapOpts,printToCons);

        %↙
------------------------------------------------------------------------↙
--
        % Smooth and Calculate Acceleration
        [~,~,accel] = func_smoothCalcAccel(pos,time,grid,disp,↙
smoothingOpts,...
            extrapOpts,diffOpts,printToCons);

        %↙
------------------------------------------------------------------------↙
--
        % VFM: MANUAL
        %↙
------------------------------------------------------------------------↙
--
        if strcmp('isotropic',globalOpts.matModel)
            % Create the virtual fields
            VFs = func_VFDynInitManIsoLinElas(VFOpts,pos,accel,strain);
            % Use the virtual fields for stiffness identification
            identStiffVFMan = func_VFDynManIsoLinElas(VFOpts,pos,time,↙
```

```
material,...
                VFs,strain,accel);

            % Calculate the median to work out the 'average' identified↙
value
            identStiffVFMan.QxxAvgOverT = nanmean(identStiffVFMan.QxxVsT↙
(VFOpts.avgQVsTRange));
            identStiffVFMan.QxyAvgOverT = nanmean(identStiffVFMan.QxyVsT↙
(VFOpts.avgQVsTRange));
            identStiffVFMan.ExxAvgOverT = nanmean(identStiffVFMan.ExxVsT↙
(VFOpts.avgQVsTRange));
            identStiffVFMan.NuxyAvgOverT = nanmean(identStiffVFMan.↙
NuxyVsT(VFOpts.avgQVsTRange));

        elseif strcmp('orthotropicReduced',globalOpts.matModel)
            % Process kinematic data with manual virtual fields
            identStiffVFMan.ExxVsT = func_VFDynManReducedLinElas(VFOpts,↙
pos,material,accel,strain);

            % Calculate an average stiffness over the specified time↙
range
            identStiffVFMan.ExxAvgOverT = nanmean(identStiffVFMan.ExxVsT↙
(VFOpts.avgQVsTRange));

            % For this case Qxx approx Exx
            identStiffVFMan.QxxVsT = identStiffVFMan.ExxVsT;
            identStiffVFMan.QxxAvgOverT = identStiffVFMan.ExxAvgOverT;

        elseif strcmp('orthotropic',globalOpts.matModel)
            identStiffVFMan = nan;
        elseif strcmp('orthotropicAngle',globalOpts.matModel)
            identStiffVFMan = nan;
        else
            identStiffVFMan = nan;
        end

        %↙
------------------------------------------------------------------------↙
--
        % VFM: PIECE-WISE SPECIAL OPTIMISED
        %↙
```

```
----------------------------------------------------------------------------↙
--
        if globalOpts.processOptVF
            if strcmp('isotropic',globalOpts.matModel)
                % Use isotropic virtual fields to get Qxx and Qxy)
                [identStiffVFOpt,VFOptDiag] =↙
func_VFDynPWSpecOptIsoLinElas(VFOpts,...
                    pos,specimen,material,accel,strain);

                % Calculate the median to work out the 'average'↙
identified value
                identStiffVFOpt.QxxAvgOverT = nanmean(identStiffVFOpt.↙
QxxVsT(VFOpts.avgQVsTRange));
                identStiffVFOpt.QxyAvgOverT = nanmean(identStiffVFOpt.↙
QxyVsT(VFOpts.avgQVsTRange));
                identStiffVFOpt.ExxAvgOverT = nanmean(identStiffVFOpt.↙
ExxVsT(VFOpts.avgQVsTRange));
                identStiffVFOpt.NuxyAvgOverT = nanmean(identStiffVFOpt.↙
NuxyVsT(VFOpts.avgQVsTRange));

            elseif strcmp('orthotropicReduced',globalOpts.matModel)
                % Use reduced optimised virtual fields to obtain Qxx
                [identStiffVFOpt,VFOptDiag] =↙
func_VFDynPWSpecOptReducedLinElas(VFOpts,...
                    pos,strain,accel,specimen, material);

                % Calculate an average stiffness over the specified time↙
range
                identStiffVFOpt.ExxAvgOverT = nanmean(identStiffVFOpt.↙
ExxVsT(VFOpts.avgQVsTRange));

                % For this case Qxx is approx Exx
                identStiffVFOpt.QxxVsT = identStiffVFOpt.ExxVsT;
                identStiffVFOpt.QxxAvgOverT = identStiffVFOpt.↙
ExxAvgOverT;

            else
                identStiffVFOpt = nan;
            end
        else
            identStiffVFOpt = nan;
```

```
        end

        %↙
------------------------------------------------------------------------↙
--
        % STANDARD STRESS GAUGE: CALCULATION
        %↙
------------------------------------------------------------------------↙
--
        % Standard Stress Gauge equation calculation
        [stress.xAvg,~] = func_stressGaugeProcess(material,time,pos,↙
accel.xAvg);

         % Create the 'pulse' struct to store the loading data
        pulse.vec = squeeze(stress.xAvg(end,:));
        [pulse.peakStress,pulse.peakInd] = max(abs(pulse.vec));
        pulse.peakTime = time.vec(pulse.peakInd);

        %↙
------------------------------------------------------------------------↙
--
        % STANDARD STRESS GAUGE: find stiffness(es) by fitting the↙
stress-strain curves
        %↙
------------------------------------------------------------------------↙
--
        if strcmp('orthotropicReduced',globalOpts.matModel)
            % Fit the stress strain curves to obtain Qxx/Exx
            % identStiffSG = func_identQxxFromStressStrainCurve↙
(stressGaugeOpts,stress,strain);
            [identStiffSG.QxxVsL,identStiffSG.QxxLinFitCoeffs]...
                = func_identStiffLinFitStressStrainCurve↙
(stressGaugeOpts,stress.xAvg,strain.xAvg);

            % Set the range over which the average stiffness is↙
identified
            stressGaugeOpts.avgQVsLRange = round(stressGaugeOpts.↙
avgQVsLRangePc(1)*length(pos.x))...
                :round(stressGaugeOpts.avgQVsLRangePc(2)*length(pos.x));
            % Calculate Identified Averages
            identStiffSG.QxxAvgOverL = nanmean(identStiffSG.QxxVsL↙
```

```
(stressGaugeOpts.avgQVsLRange));
            identStiffSG.ExxVsL = identStiffSG.QxxVsL;
            identStiffSG.ExxAvgOverL = nanmean(identStiffSG.ExxVsL↙
(stressGaugeOpts.avgQVsLRange));

        else % Default to isotropic material model
            % Calculate the axial strain including the poisson effect
            if strcmp('VFOpt',stressGaugeOpts.strainCalcNuxy)
                identStiffSG.strainCalcNuxy = identStiffVFOpt.↙
NuxyAvgOverT;
            elseif strcmp('VFMan',stressGaugeOpts.strainCalcNuxy)
                identStiffSG.strainCalcNuxy = identStiffVFMan.↙
NuxyAvgOverT;
            else % Assume the QS value
                identStiffSG.strainCalcNuxy = material.nuxy;
            end
            strain.xnyAvg = strain.xAvg + identStiffSG.↙
strainCalcNuxy*strain.yAvg;

            % Fit the stress strain curves to obtain Qxx
            [identStiffSG.QxxVsL,identStiffSG.QxxLinFitCoeffs] = ...
                func_identStiffLinFitStressStrainCurve(stressGaugeOpts,↙
stress.xAvg,strain.xnyAvg);

            % Set the range over which the average stiffness is↙
identified
            stressGaugeOpts.avgQVsLRange = round(stressGaugeOpts.↙
avgQVsLRangePc(1)*length(pos.x))...
                :round(stressGaugeOpts.avgQVsLRangePc(2)*length(pos.x));
            % Calculate Identified Averages
            identStiffSG.QxxAvgOverL = nanmean(identStiffSG.QxxVsL↙
(stressGaugeOpts.avgQVsLRange));
            identStiffSG.ExxVsL = identStiffSG.QxxVsL.*(1-identStiffSG.↙
strainCalcNuxy^2);
            identStiffSG.ExxAvgOverL = nanmean(identStiffSG.ExxVsL↙
(stressGaugeOpts.avgQVsLRange));
        end

        %↙
///////////////////////////////////////////////////////////////////////////↙
/
```

```
        % Store all required data for this simulation
        %↙
///////////////////////////////////////////////////////////////////////↙
/

        % Smoothing options
        sweepSmoothingOpts{kernNum}{simNum} = smoothingOpts;
        % Pulse Data
        pulseStructSweep{kernNum}{simNum} = pulse;
        % Identified stiffness data
        identStiffVFManStructSweep{kernNum}{simNum} = identStiffVFMan;
        identStiffVFOptStructSweep{kernNum}{simNum} = identStiffVFOpt;
        identStiffSGStructSweep{kernNum}{simNum} = identStiffSG;

        % Get the peak stress of the pulse
        simPulsePeakStressVec(simNum) = pulse.peakStress;
        % Get the average identified stiffness values
        simQxxAvgVecSG(simNum) = identStiffSG.QxxAvgOverL;
        simQxxAvgVecVFMan(simNum) = identStiffVFMan.QxxAvgOverT;
        simQxxAvgVecVFOpt(simNum) = identStiffVFOpt.QxxAvgOverT;

        if strcmp('isotropic',globalOpts.matModel)
            simQxyAvgVecVFMan(simNum) = identStiffVFMan.QxyAvgOverT;
            simQxyAvgVecVFOpt(simNum) = identStiffVFOpt.QxyAvgOverT;
        end
        %↙
///////////////////////////////////////////////////////////////////////↙
/

        % Store the time that it took to complete this iteration
        iterTime(iter) = toc;
        iter = iter+1;
    end
    %↙
///////////////////////////////////////////////////////////////////////↙
/
    % Calculate averages/SD from Monte Carlo Loop
    %↙
///////////////////////////////////////////////////////////////////////↙
/
```

```
    % Pulse peak stress value
    kernelPulseAvg.peakStress(kernNum) = mean(simPulsePeakStressVec);

    % Qxx systematic Error - mean over kernels
    kernelQxxAvg.SG(kernNum) = mean(simQxxAvgVecSG);
    kernelQxxAvg.VFMan(kernNum) = mean(simQxxAvgVecVFMan);
    kernelQxxAvg.VFOpt(kernNum) = mean(simQxxAvgVecVFOpt);
    % Qxx Random Error - SD over kernels
    kernelQxxSD.SG(kernNum) = std(simQxxAvgVecSG);
    kernelQxxSD.VFMan(kernNum) = std(simQxxAvgVecVFMan);
    kernelQxxSD.VFOpt(kernNum) = std(simQxxAvgVecVFOpt);
    % Qxx Vector of Identified Values
    kernelQxxVec.SGSweep{kernNum} = simQxxAvgVecSG;
    kernelQxxVec.VFManSweep{kernNum} = simQxxAvgVecVFMan;
    kernelQxxVec.VFOptSweep{kernNum} = simQxxAvgVecVFOpt;

    if strcmp('isotropic',globalOpts.matModel)
        % Qxy systematic Error - mean over kernels
        kernelQxyAvg.VFMan(kernNum) = mean(simQxyAvgVecVFMan);
        kernelQxyAvg.VFOpt(kernNum) = mean(simQxyAvgVecVFOpt);
        % Qxy Random Error - SD over kernels
        kernelQxySD.VFMan(kernNum) = std(simQxyAvgVecVFMan);
        kernelQxySD.VFOpt(kernNum) = std(simQxyAvgVecVFOpt);
        % Qxy Vector of Identified Values
        kernelQxyVec.VFManSweep{kernNum} = simQxyAvgVecVFMan;
        kernelQxyVec.VFOptSweep{kernNum} = simQxyAvgVecVFOpt;
    end
    %↙
///////////////////////////////////////////////////////////////////////////↙
/
end
end

%% POST-PROCESSING: Save the data to file
fprintf('\n')
fprintf('--------------------------------------------------------------↙
\n')
fprintf('PARAMETRIC SWEEP SIMULATION LOOP COMPLETE.\n')
fprintf('--------------------------------------------------------------↙
\n')
```

```
% Unpack the composite data structures to save them to file
fprintf('Unpacking worker composite data structures for saving.\n')
for ww = 1:simOpts.numWorkers
    % Store the smoothing kernels from each worker
    saveData(ww).sweepSmoothingOpts = sweepSmoothingOpts(ww);
    saveData(ww).workerKernels = workerKernels(ww);

    % Store all the data structures with the required information
    saveData(ww).pulseStructSweep = pulseStructSweep(ww);
    saveData(ww).identStiffSGStructSweep = identStiffSGStructSweep(ww);
    saveData(ww).identStiffVFManStructSweep = identStiffVFManStructSweep↙
(ww);
    saveData(ww).identStiffVFOptStructSweep = identStiffVFOptStructSweep↙
(ww);

    % Store all variables of interest that are averaged over the given
    % number of iterations
    saveData(ww).kernelPulseAvg = kernelPulseAvg(ww);
    saveData(ww).kernelQxxAvg = kernelQxxAvg(ww);
    saveData(ww).kernelQxxSD = kernelQxxSD(ww);
    saveData(ww).kernelQxxVec = kernelQxxVec(ww);
    if strcmp('isotropic',globalOpts.matModel)
        saveData(ww).kernelQxyAvg = kernelQxyAvg(ww);
        saveData(ww).kernelQxySD = kernelQxySD(ww);
        saveData(ww).kernelQxyVec = kernelQxyVec(ww);
    end

    % Iteration and simulation time data
    saveData(ww).iterTime = iterTime(ww);
    workerIterTimeVec(ww) = iterTime(ww);
    workerTotalTimeVec(ww) = sum(workerIterTimeVec{:,ww});
end

[fastestWorkerTime,fastestWorker] = min(workerTotalTimeVec);
fastestWorkerNumKerns = kernelsPerWorker(fastestWorker);
[slowestWorkerTime,slowestWorker] = max(workerTotalTimeVec);
slowestWorkerNumKerns = kernelsPerWorker(slowestWorker);

% Save the data to file
fprintf('Saving data to file...\n')
savePath = imagePath;
```

```
saveFile = 'ParametricImageDefSweep_AllData.mat';
save([savePath,saveFile],'saveData')
fprintf('Data saved.\n')

% Print some info about how long the simulation took
fprintf('\n')
fprintf('Total simulation time: %.2f hours\n',slowestWorkerTime/(60*60))
fprintf('Fastest worker: %i, at %0.2f hours assigned %i kernels\n',...
    fastestWorker,fastestWorkerTime/(60*60),fastestWorkerNumKerns)
fprintf('Slowest worker: %i, at %0.2f hours assigned %i kernels\n',...
    slowestWorker,slowestWorkerTime/(60*60),slowestWorkerNumKerns)
fprintf('\n')
```