**S t r a i n M u s i c**

**2D/3D strain algorithm for ultrasound data**

|  |  |  |
| --- | --- | --- |
| **date** | **version** | **comments** |
| 01-sep-2016 | V1.0 | initial version |
| 22-mar-2016 |  | changes in template/kernel due to plane matching |
| 15-apr-2016 |  | downsampling added |
| 12-apr-2017 | V1.1 | scatterInterpolant only calculated when iteration or angle changed |
| 01-aug-2017 | V1.2 | Bug found in calcIndices (170801-01) |
| 22-nov-2018 | V1.3 | RigidMedianChunkFilter added |
| 26-nov-2018 | V1.4 | StrainChunkLSQSE added |
| 25-01-2021 | V1.5 |  |

Table of Contents

[1. Introduction 5](#_Toc453165535)

[2. Files 6](#_Toc453165536)

[a) data-files 6](#_Toc453165537)

[b) header-files 6](#_Toc453165538)

[c) ROI-files 7](#_Toc453165539)

[d) GRID-file 7](#_Toc453165540)

[e) PARAMeter-file 8](#_Toc453165541)

[f) RESult-file 9](#_Toc453165542)

[3. Outline of the program 10](#_Toc453165543)

[4. the PARAMeter structure 12](#_Toc453165544)

[5. The RESult structure 14](#_Toc453165545)

[6. Syntax user defined functions 16](#_Toc453165546)

[a) General fields in parameter struct 16](#_Toc453165547)

[b) Variable transport 16](#_Toc453165548)

[c) Blockmatching 16](#_Toc453165549)

[d) Cross-correlation 17](#_Toc453165550)

[e) Peak interpolation 18](#_Toc453165551)

[f) Filtering 18](#_Toc453165552)

[g) Post-processing 19](#_Toc453165553)

[h) functions that are already in the framework 19](#_Toc453165554)

[7. Downsampling 22](#_Toc453165555)

[8. Functions researchers can use in their own routines 23](#_Toc453165556)

[a) getOffsets 23](#_Toc453165557)

[b) calcIndices 23](#_Toc453165558)

[c) normxcorrn 23](#_Toc453165559)

[d) fromBaseWorkSpace 23](#_Toc453165560)

[e) toBaseWorkSpace 23](#_Toc453165561)

[f) indices2mat 24](#_Toc453165562)

[g) mat2indices 24](#_Toc453165563)

[h) getNumberWorkers 24](#_Toc453165564)

[i) envelopDownSample 24](#_Toc453165565)

[j) interpResult 24](#_Toc453165566)

[9. General rules 25](#_Toc453165567)

[10. examples 26](#_Toc453165568)

[a) calling the framework 26](#_Toc453165569)

[b) param structure 26](#_Toc453165570)

[c) downsampling 27](#_Toc453165571)

[11. References 30](#_Toc453165572)

# Introduction

Nowadays the hospital is getting more interested in determining the characteristics and functionality of tissues for diagnosis and monitoring of diseases. Ultrasound can be used to obtain this information; it’s easy to use, non-invasive and relatively cheap. At the Medical UltraSound Imaging Center (MUSIC) of the RadboudUMC a software-package <strain2dmain> was developed to estimate deformations in tissues from ultrasound recordings. From these deformations functional information for instance contractility of muscles and heart and stiffnes/elasticity of tissue can be derived. The current version (8.0) of the software was dedicated to 2D images (a 3D version was derived but was not tested and documented). Also, with this version of the software, the user is not able to incorporate their own developed algorithms, which is a disadvantage in a research group as MUSIC is.

Another small disadvantage is the amount of (redundant) parameters that had to be defined which are not clear and difficult to understand for new users and the time spent in some algorithms

Therefore, a new software package <strainMusic> was developed with the following in mind

* It should work on both 2D and 3D datasets
* It should act as a framework; the researchers should be able to use their own developed algorithms/functions for estimating displacement and/or strain. For standard estimation of strain and displacement functions are included in the framework
* The initial settings (parameters) should be easy to understand, well documented, and classified for each function
* Parallel processing should be possible to speed up the framework

The introduction of ultrafast ultrasound imaging opens new possibilities for displacement estimation. Therefore, with the development of the new software, this new functionality was added (user defined points to estimate displacement). Also, the used file format/structures are changed

This document contains a description of the framework and should be used as a reference for the users of the framework. The format of the files is described as well as the parameters used. This document also contains detailed instructions how to interface new custom-made functions by researchers to the framework

The new framework <StrainMusic> is mainly developed in Matlab with some functions written in C.

*September 2015*

Rik Hansen

Jan Menssen

# Files

The software uses the following files

* data files: 2 data files are used, containing the beamformed ultrasound data of the tissue in pre- and post-deformed state respectively. Each file contains data for one (compound) 2D or 3D ultrasound frame.
* header file: for all data-files in the same study (all frames), one header file is used that describes the data
* ROI-file (optional) A file that contains the region-of-interest (ROI) for which the displacement estimates should be calculated. If the file doesn’t exist, the framework assumes the whole frame should be used
* GRID-file (optional): this file contains the x-, y-, and z-coordinate of the beamformed US data and indices that specify the position in the beamformed grid for which displacement and/or strain should be calculated.
* PARAMeter file: this file contains settings for the framework to calculate strain and/or displacement.
* RESult files: these files are created by the package and contains the results from the strain/displacement calculations

Below the definition and syntax is given for all these files

## data-files

Ultrasound data storage is done for each (compound) frame is a separate file. The filename convention is

US\_<name data set>\_<framenr> .mat (US\_flowsims.00010.mat)

For <framenr> 5 digits should be used. A single file contains the variable

* USDATA{angle}(:,:,:) which is a cell-array containing the data for each beam of a (compound) ultrasound frame grouped per beamsteering angle. First dimension is axial, 2nd dimension lateral and if exists 3rd dimension elevational

To save memory, only one signaltype (RF) should be stored. In the case of signaltype = RF, envelope data are derived in the software. (Hilbert-transform). Also all variables are singles instead of doubles

## header-files

For each set of frames in a study a header file should exist. This file has the following name convention

USHEADER\_<name\_of\_data\_set>.mat (USHEADER\_flowsims.mat>

This file contains a variable USHEADER that consists of the following fields

* .c: speed of sound in m/s (double)
* .fs sampling frequency in Hz (double)
* .fc: central frequency of transducer in Hz (double)
* . nFrames: the number of frames of the original data set (double)
* . system: a string defining the system that is used to record the data. For simulation use the package name, for instance ‘Field II pro”
* .transducer : string that represents the name of the probe, e.g. ‘L12-5’
* .xmitangles : vector describing the steering angles that are used in transmition, angles are in degrees [20 0 -20]
* .acquisitionDimension : the number of dimensions used in the acquisition in string format, for instance ‘2D’ or ‘3D’
* pitch : double vector containing the pitch in lateral and elevational direction (for 3D). For 2D mode only a number is given, format [lateral elevational]
* .rcvFnum : the fnumber that is used for beamforming in receive (double)
* .signaltype : string representing the signal type. For instance ‘RF’ for RF data, ‘ENV’ for envelope data ‘DICOM’ for dicom data and so on

NOTE : the header field is far from complete and not all fields are used in the framework. Also not all fields can be defined for each transducer type and acquisition mode. All possible header description fields should be stored in a separate document.

## ROI-files

An ROI-file contains the region-of-interest for which the displacement estimates should be calculated. An ROI *can* be defined for each (compound) frame. The file name conventions is the same as for the data-file

ROI\_<name\_of\_dataset>\_<framenr>.mat (ROI\_flowsims\_00010.mat)

The dataset contains the following variables

* ROI{angle}(:,:,:) : cell-array of logicals. Same size/dimension as in the data file. True means the image point is positioned within the ROI and displacement will be estimated for that point, a false is outside ROI, so no displacements will be estimated

NOTE : the <strainMusic> framework is backwards compatible, so it’s not necessary to convert old ROI files. However when using the conversion tool, an ROI file using the new format is generated.

## GRID-file

This file contains a description of the USGRID, containing, the X, Y and Z coordinates (in m) of the ultrasound data samples. Also it holds the displacement estimation grid (DISPGRID), containing the coordinates of the data samples for which a displacement should be estimated. These coordinates are stored as indices into the file, (see Matlab functions <ind2sub> and <sub2ind> for a conversion between indices and position in the data-matrix)

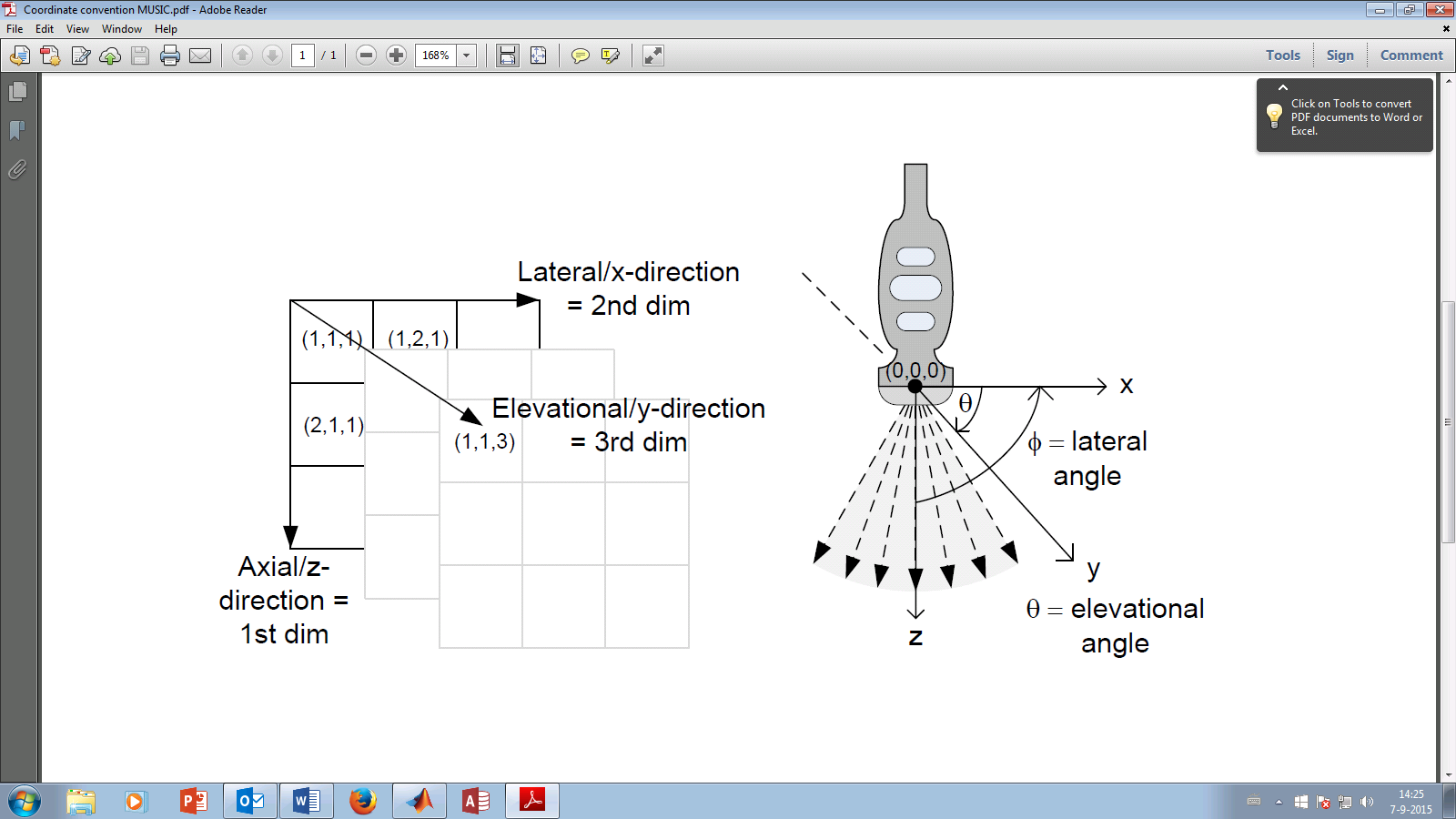
The file name convention is

GRID\_<user\_defined\_name>.mat (GRID\_smallarea.mat>

The file contains the following variables

* USGRID{angle}
  + .x : single row vector containing the x-coordinates of the US pixels (m)
  + .y : single row vector containing the y-coordinates of the US pixels (m)
  + .z : single row vector containing the z-coordinates of the US pixels (m)
  + .size : contains the size in x, y and z direction ([x y z])  
    Note: for 2D data, only define [x,z]
* DISPGRID : cell array of size (iter,angle) containing a field <indices>. This field is an unsigned int32 row-vector with indices that are used in displacement estimation. These indices can also be used in the USGRID.x/y/z vector to get the coordinates/

See the figure below



All vectors are row-vectors ([1 2 … n])

NOTE : if only a DISPGRID is defined for one iteration, the grid is used for all iterations

## PARAMeter-file

This file contains the parameters that are used to estimate displacement or strain. The name convention is

PARAM\_<user\_defined\_name>.mat (PARAM\_bestsettings.mat)

The file contains the following variables

* PARAM{angle} ; cell-array containing a structure

For a description of the structure see the paragraph regarding the PARAM structure

## RESult-file

This file type is generated by the StrainMusic software, The file name has the following syntax

RES\_<name\_of\_dataset>\_<preframenr>\_<postframenr>.mat

for instance ‘RES\_flowsims\_00001\_00002.mat’. The file contains a RES cell-array (number of angles) containing a structure with result for the displacement and strain estimations. For a detailed description see the chapter about this structure. Results of each iteration are stored in each field, for instance: RES{3}.ax{2} for the axial displacement obtained after the 2nd iteration for the 3rd compound angle

# Outline of the program

The <strain2Dmain> software uses a “coarse-to-fine” approach. In a number of iterations the displacement is estimated starting at a coarse scale obtaining rough estimates working towards a fine scale for estimates at higher accuracy. This approach is also used in the new framework <StrainMusic>

Before the iteration loop is started, the data and parameter file are loaded and if exists also the ROI and GRID file. Next an initial version of the <RES> structure is created. This part of the framework cannot be changed by the user.

The main part of the program consists of an iteration loop. In this loop, displacements are estimated, the obtained results can be filtered and on this filtered value, some “post-processing” could be done, for instance strain calculations. The displacement estimation is split into 3 separate tasks: 1) template/kernel selection (block matching), 2) estimation of displacement by using the peaks of a normalized cross correlation function and 3) a peak-interpolation step for sub-sample displacement estimation. After that the estimated displacements are written to the <RESult> structure. The post-processing part may consist of some steps for instance deriving strain from displacement. For all these steps, the researcher may use his own developed functions

When the loop is finished, some cleaning up is done and if desired the <RESult> structure is written to a file before the program ends. Researchers are not able to modify this part

Below the program is given in pseudo code

* read Data
* read PARAMeters
* read of create ROI
* read or create UltrasSound GRID and DISPlacement GRID
* initialise RESult structure
* for each iteration
  + create template/kernels from data (*ccf\_blockmatch)*
  + find displacement using peaks in normalized CCF (*ccf\_method*)
  + better peak finding at subsample level (*ccf\_peakinterp*)
  + filtering of estimated displacements <*xxfilter, xx={ax lat ele}>*
  + postprocessing (*postprocessing)*
* endfor
* clean up <RESult> structure if required
* save <RESult> structure

Within the iteration loop, it’s possible to use a user-defined function for each iteration separately. So, for the first iteration another peakfinding algorithm can be used than for the last iteration. Postprocessing is still even more flexible, each iteration step consists of a free number of post-processing steps (and functions)

The names in *italics* after the step refers to the fields that are used in the <PARAMeter> structure to set these functions and the belonging parameters of these functions.

Not visible in the pseudo code but each functionality is done for every compound angle. The subfunctions in <calcDisplacments> are done sequentially before the estimation is done for the next angle. Within the framework it is possible to use other functions for different angles. The post processing step is more is even more flexible, data of all angles can be used in functions for an angle seperatly

For researchers who use their own algorithms, it’s possible that data created in one step or function could be used in another step or function. In this version (StrainMusic V2.0) of the framework this is done by using the Matlab <base> workspace. Functions are developed to copy variables from the function to the <base> workspace and from the <base >workspace to the function. The researcher should use these functions; use of global data is not encouraged

In the <PARAMeter> structure the <.input> and <.output> fields are already defined for the above mentioned variables and although not necessary in this implementation of the framework, these fields will be used in a new release.

# the PARAMeter structure

This structure is used to set parameters for the framework. Regarding the huge number of redundant parameters used in <strain2Dmain>, this version of the software uses only those parameters necessary. The parameters are defined in functional groups to gives the user more insight into were the parameters is used for.

Because it’s possible to incorporate algorithms/functions that are developed by researchers, it must be possible to create new parameters that are needed by these algorithms

Parameters are stored in a structure. This structure is a cell-array of dimension {angle}. This means the parameter should be defined for every (compound) beamsteering angle that is used and, in most cases, also for each iteration

As well as for data, a conversion tool has been developed to convert parameters for the <strain2Dmain> software to parameters used for the <StrainMusic> framework

The parameters can be split into:

* general parameters
* parameters belonging to functionality (e.g. blockmatching, displacement filtering)

General parameters are:

* no\_iter : double, number of iterations used
* signaltype{no\_iter}: cell array, containing ‘raw’ or ‘env’ , for data that should be used in displacement estimation
* nCCF{no\_iter} : the number of template/kernels that should be handled at once. This parameter is added to avoid excessive memory usage in 3D calculation. It’s preferred to leave this parameter empty in 2D, then all template/kernels are created and handled in one run
* final\_iter\_only : Boolean, 1 indicates only results from the last iteration should be stored in the RES-file 0 means everything is stored. Default is 1
* filtered\_disps\_only : Boolean indicating of only filtered displacement should be stored in the <RES> structure, 1 means only filtered displacement 0 means also unfiltered data, default = 1

There are always 2 parameters defined for a functionality. These parameters are:

1. func : string containing the name of the function that is used (don’t forget preceeding “@” sign)
2. additional: structure that holds the (additional) parameter names for the function, e.g. <additional.window> and <additional.max\_disp> for rigid/freeshape blockmatching. Currently 2 fields are defined for data exchange (if no data exchange is needed, these fields should be omitted) and 1 field is used for parallel processing (if function is not parallel, field can be omitted)
   1. additional.inputs : cell array of strings containing variable names that should be copied from the base workspace and are used in the function (e.g {“myVar1” “myVar2”}
   2. additional.outputs : cell array of strings containing variable names that should be placed from the current function into the base workspace to use in another function (e.g. “myOutputVar1”, “myOutputVar2”)
   3. additional.ncores : the number of cores used for the algorithm. If empty, the maximum number of cores is used

So the following functionality parameters must be set

* kernel/templates blockmatch for CCF function
  + .ccf\_blockmatch(no\_iter).func
  + .ccf\_blockmatch(no\_iter).additional
* finding displacement estimates using normalized cross correlation
  + .ccf\_method(no\_iter).func
  + .ccf\_method(no\_iter).additional
* finding peaks on sub-sample level
  + .ccf\_peakinterp(no\_iter).func
  + .ccf\_peakinterp(no\_iter).additional
* displacement filtering
  + .axfilter.(no\_iter).func
  + .axfilter(no\_iter).additional
  + .latfilter(no\_iter).func
  + .latfilter.additional
  + .elefilter(no\_iter).func
  + .elefilter(no\_iter).additional
* postprocessing (including strain calculation)
  + postprocessing(nr\_step,no\_iter).func
  + postprocessing(nr\_step,no\_iter).addtional

The additional parameters for the standard function are given at the description of the function (see Chapter 6)

# The RESult structure

Results are stored in a structure. For each <angle>, the results are stored, so RES becomes a cell-array. Most fields differs for each iteration. These fields are also cell-array’s

Some fields are static, independent on angle and iteration. These fields are only stored in the first index of the cell-array RES

The following fields are in the structure

*Field relative to version software, independent of angle*

* .strain\_version : string specifying the version of the strain software that was used
* .os\_version : string containing the version of the operating system that was used
* .os\_system : string containing the operating system that was used
* .matlab\_version : string containing the matlab version used

*Fields relating the files that are used, angle independent*

* . prefile: name of the file that contains the pre-deformation data
* . postfile: name of the file that was used for the post-deformation data
* . roifile: name of the file that was used for the ROI data
* . gridfile: string specifying the displacement grid file that was used

*Field related to the files that are used AND angle dependent*

* .paramfile : string containing the name of the parameter file that was used

*General fields*

* .dispindx{no\_iter} : vector (unsigned int32) containing the indices of the points that are used for displacement estimation. Is the AND function between the indices that are in the GRID file and the ROI
* roi{no\_iter}.size : size of the ROI file (and the pre/post datafile). Field is the same as in the GRID file, internal use in <StrainMusic>

*Displacement fields*

* .ax{no\_iter} vector containing the axial displacement for each iteration. (<indices2mat> can be used to generate a matrix of data)
* .lat{no\_iter} : vector containing lateral displacement
* .ele{no\_iter} : vector containing elevational displacement
* .axf{iter}, filtered axial displacement, if no filter is used, this is a copy of the <ax> field
* .latf{iter}, filtered lateral displacement for each iteration, copy of <lat> if no filter is used
* .elef{iter} filtered elevational displacement for each iteration copy of <ele> if no filter is used.

*Optional fields, containing strain results*

* .sxx{no\_iter} : vector of doubles containing the lateral strains for each displacement pixel
* .sxy{no\_iter} : vector of doubles containing the lateral/elevational shear strain for each displacement pixel
* .sxz{no\_iter{ : vector of doubles containing the lateral/axial shear strain for each displacement pixel
* .syy{no\_iter} : vector of doubles containing the elevational strains for each displacement pixel (not in 2D0
* .syx{no\_iter} : vector of doubles containing the elevational/lateral shear strains for each displacement pixel
* .syz{no\_iter} : vector of doubles containing the elevational/axial shear strains for each displacement pixel
* .szz{no\_iter} : vector of doubles containing the axial strains for each displacement pixel
* .szx{no\_itter} : vector of doubles containing the axial/lateral shear strains for each displacement pixel
* .szy{no\_iter} : vector of doubles containing the axial/elevational shear strain for each displacement pixel

# Syntax user defined functions

This chapter contains information about functions that can be developed by the researcher to overrule the standard functions. For each iteration, another function can be used. These functions are called for each angle, this is not the responsibility of the researcher

## General fields in parameter struct

As described in chapter 4, for each function there are 2 fields in the parameter structure. These fields are :

* .func : function pointer to the used function
* .additional : structure that my contain additional parameters (field). The names of these parameter may be defined by the user. However 2 names are reserved for variable transport
  + .additional.input : cell\_array with names of variables that should be imported from the base workspace to the function
  + additional\_output. : cell array with names of variables names that should be exported from the function to the Matlab base workspace

## Variable transport

As mentioned in the previous paragraph it is possible to import and export data into a user defined function, for instance export a variable in the cross-correlation function and import it into the peak interpolation function. Two functions are written for this purpose. These functions should be included in the user defined functionality if variable transport is needed

* frombaseWorkSpace(vars)
* toBaseWorkSpace(vars);

These functions are also explained in the next chapter: Functions that can be used by the researcher.

## Blockmatching

From the pre- and post data this function finds all template/kernel combinations that are used in the cross-correlation function. This function is called for every angle separate.

Syntax of this function

[template,kernel,res] = func(iter,angle,res,pre,post,additional)

With the following input parameters:

* iter : iteration number
* angle : current angle
* res : results structure (for the selected angle)
* pre : double matrix with pre-deformation data (for the selected angle)
* post : double matrix with the post-deformation data (for the selected angle)
* param : parameter structure for the current angle
* additional : structure with additional parameters as defined by the user (templatesize, shift)

Output parameters

* template : structure containing 2 fields
  + size : contains the size of the templates, [ax lat ele] for 3D and [ax lat] for 2D
  + data : cell array of size <number of kernel/template combinations), with all templates
* kernel : structure containing 2 fields
  + size : contains the size of the templates, [ax lat ele] for 3D and [ax lat] for 2D
  + data : cell array of size <number of kernel/template combinations), with all kernels

## Cross-correlation

With input the template/kernel combinations, this function should return thee CCF-matrix and the peaks that are found. As in the blockmatch function, this function is called for every angle

[ccf,cmax,peaks] = func(template,kernel,additional)

Input parameters

* template : structure containing 2 fields
  + size : contains the size of the templates, [ax lat ele] for 3D and [ax lat] for 2D
  + data : cell array of size <number of kernel/template combinations), with all templates
* kernel : structure containing 2 fields
  + size : contains the size of the templates, [ax lat ele] for 3D and [ax lat] for 2D
  + data : cell array of size <number of kernel/template combinations), with all kernels
* additional : structure with additional parameters that can be defined by the user (and needed in the function)

Output parameters

* ccf : structure containing the cross-correlation matrices as returned by the used cross-correlation function. The structure contains 2 fields
  + size : contains the size of the ccf-matrix, [ax lat ele] for 3D and [ax lat] for 2D
  + data : cell array of size <number of ccf matrices)
* cmax : vector that contains the maximum value for each template/kernel combination
* peaks : structure containing the position the maximum is found. This position is stored in 3 vectors, the indices in the ccf-matrix
  + .ax(# templates/kernels) : position in axial direction
  + .lat(# templates/kernels) : position in lateral direction
  + .ele(# templates/kernels) : position in elevation direction

In 2D the ele-field is empty but MUST be defined to be compatible with the framework

## Peak interpolation

In this function, the peaks and ccf matrix found in the previous functions can be used for a better position estimation at sub-sample level. All positions are relative to the origin of the ccf-matrix (1,1,1)

[cmax,peaks] = func(ccf,peaks,additional)

With input parameters

* ccf : structure containing the cross-correlation matrices as returned by the used cross-correlation function. The structure contains 2 fields
  + size : contains the size of the ccf-matrix, [ax lat ele] for 3D and [ax lat] for 2D
  + data : cell array of size <number of ccf matrices)
* peaks : structure with <ax>, <lat> en <ele> field. Each field is a vector containing the displacement in that direction, sample based (the output of the cross-correlation function)
* additional : structure with additional parameters that are defined in the parameter structure

And output parameters

* cmax : vector with all maxima found (after a better peak finding). The size is the number of template/kernel combinations
* peaks : same as the input peaks structure, however now the position is at sub-sample level

## Filtering

Displacements found in the displacement-estimation can be filtered to remove outliers. Standard for each direction (axial, lateral and elevational) no filtering is done, but a tow median filters are added to the framework and can be used. However researcher can also use other filters by developing their own functions for axial, lateral and elevational direction separately. For each iteration, another function can be used. The syntax of this function

data = func(data,indices,grid,additional)

Input parameters

* data : vector with displacements as stored in the <res.ax>, <res.lat> or <res.ele>
* indices : indices for which displacement is estimated
* grid : US grid as defined in the grid file (needed for calculating filtering kernel)

And output parameter

* data : vector with filtered displacement that is stored in <res.axf>, <res.latf> or <res.elef>

## Post-processing

Filtered displacements can be post-processed. In many cases this post-processing is deriving the strain and shear strain from the displacements. As for the above mentioned functions, the researcher can develop his own post-processing functions. However, there are two differences, 1) it is possible to use more than one function in each iteration and 2) postprocessing can be done using information of all angles. The syntax of the function:

res = func(iter,angle,res,grid,additional)

With input parameters

* iter : iteration number
* angle : angle number
* res : result structure
* grid : ultrasound grid (see chapter 2 - GRID file)
* additional : structure with additional parameters

And output parameter

* res : modified result structure

## functions that are already in the framework

* rigidBlockMatch : standard routine that creates square kernel/templates from the pre- and postdata used for cross-correlation. 2D/3D implementation. Additional parameters are
  + window : size of template in samples (one\_sided, in samples), the format is [ax lat ele] in samples. Note: for 2D data, format is [ax lat].
  + max\_disp : maximum displacement (one sided, window + max\_disp is size kernel in one direction), format [ax lat ele] in samples. Note: for 2D data, format is [ax lat].
* freeShapeBlockMatch : freeshape routine that creates kernel/template from the pre and postdata used for cross correlation. Works also on 2D as well as in 3D
  + window : size of template in samples (one\_sided, in samples) [ax lat ele]
  + max\_disp : maximum displacement (one sided, window + max\_disp is size kernel), format [ax lat ele]
* normXcorrPiotr : calculates normalized cross-correlation. The toolbox function of Piotr Dollar are used, these can be used for 2D/3D. A parallel implementation is used. Additional parameters are :
  + ncores : specifies the number of cores to use. If empty all available workers are used. If <ncores> is larger than the available workers, all available workers are used
* normXcorrEatonCPU: calculates normalized cross-correlation. The function of Daniel Eaton is used (implemented in C by Jan Menssen), which is only available for 2D.
* parabolicFit : peakfinding at sub-sample level using parabolic interpolation in uncoupled directions. No additional parameters
* interpFast : peakfinding at sub-sample level using interpolation in a iterative way. Additional parameters are :
  + nIter : number of iterations used
  + psize : number of ccf-points in matrix around peak that are used (one-sided). Should be empty is whole matrix is used
  + method : interpolation method used : ‘linear’, ‘spline’, ‘cubic’ or ‘nearest’
  + ncores : the number of CPU cores that are used, if empty all cores are used
* RigidMedianFilter : filters the displacements using a median filter. The routine is realized using parallel computing Additional parameters are
  + kernel : the size of the kernel [ax lat ele] in m, NOT in samples, single-sided!!. For 2D data, format is [ax lat].
  + ncores : the number of wanted cores, if empty all available cores are used
  + angle : rotation angle (optional parameter)

|  |
| --- |
|  |
| (a)Data obtained at 0˚ steering angle, angle = 0, results in kernel parallel to RF-lines. (b) Data obtained at θ steering angle, angle = 0, results in kernel parallel to (x,z)-axis, not to RF-lines. (c) Data obtained at θ steering angle, angle = θ, results in kernel parallel to RF-lines |

* RigidMedianChunkFilter : filters the displacements using a median filter. The routine is realized using parallel computing using chunks to reduce to amount of memory copied to and from the workers. This makes it a faster version of <rigidMedianFilter>. In the future <RigidMedianFilter> will be replaced by this filter. Additional parameters are
  + kernel : the size of the kernel [ax lat ele] in m, NOT in samples
  + ncores : the number of wanted cores, if empty all available cores are used
  + angle : rotation angle (optional parameter)
  + chunksize : the size of the chunks (optimum between 5000-10000), if empty all indices are used
* strainLSQSE : calculates the strain using a LSQSE method in 2D as well as in 3D Additional parameter ;
  + kernel : the size of the kernel [ax lat] in m, NOT in samples
  + ncores : the number of wanted cores, if empty all available cores are used
* StrainChunkLSQSE; calculates the strain using a LSQSE method in 2D as well in 3D. In 2D this function is the same as <strainLSQSE>, in 3D it makes uses of blocks (chunks) of data, because it’s faster (the amount of data copied to and from the workers is smaller). Additional parameters are:
  + kernel : the size of the kernel [ax lat] in m, NOT in samples
  + ncores : the number of wanted cores, if empty all available cores are used
  + chunksize: the size of the blocks that are processed at onde, optimum between 5000-10000
* replaceDisp : postprocessing routine to replace displacement in the RES structure with displacement that are stored in a \*.RES file. Additional parameters are
  + file : name of the RES file with displacements

# Downsampling

In the current software, it is not possible to down-sample the signal and find displacement on the envelope of the down-samples signals. Some reasons are

* Displacement points could be everywhere
* It’s not necessary coordinates of data points are on a linear grid
* For 3D a lot of memory is needed
* In 2D the software is fast enough to calculate displacement on envelope of the raw data

However using a “trick” it is possible to use down-sampled envelope data for the first iteration and raw data for the next iteration steps. This trick consist of the following steps

1. Down-sample and create envelop data from the original data. Don’t forget to adapt also the GRID files and the ROI files. Create for the down-sampled data files described in this document (routine <envelopDownSample> does this job)
2. Create a parameter file with only 1 iteration. Use for the signaltype “raw” instead of “env” as expected
3. Run <strainMusic>, results are stored in the \*.RES file
4. Use this \*.RES file and the original and downsampled GRID files as input for the <interpResult> routine. <interpResult> interpolates the displacements found to displacement according the original GRID file (points for which displacements should be calculated)
5. Now use the original data, original ROI and original GRID.
6. Create a PARAM file with at least 2 iterations. For the first iteration, leave the ccf\_\*(1).func empty and replace the displacements for the first iteration that are normally found in the software with the interpolated results from step 4. So
   1. ccf\_blockmatch(1).func = [];
   2. ccf\_method(1).func = [];
   3. ccf\_peakinterp(1).func = [];
   4. postprocessing(1,1).func = @replaceDisp
   5. postprocessing(1,1).additional.file = “name of RES file”
7. This should do the job

In chapter 10 an example is given (c)

# Functions researchers can use in their own routines

Researchers who wants to develop their own routine can make use of a set of routines already in the framework in their own functions. These functions are

## getOffsets

This function is used in template/kernel matching and returns the (interpolated) displacements calculated in the previous iteration

[ax,lat,ele] = getOffsets(iter,angle,res,window)

If window is defined (in samples based on the size of the data), a full matrix for that window is returned, else only results for the requested indices are returned.

## calcIndices

This function returns the axial, lateral and elevational indices to obtain datapoints , as well as the calculated offsets for the given the displacement indices and a kernel size

[nrelem,axindx,latindx,eleindx,axoffset,latoffset,eleoffset,newindices]

= calcIndices(iter,angle,res,datasize,kernel,mode)

With mode the user is able to remove points that are outside the ROI

## normxcorrn

normalized cross-correlation function, from the toolbox of Piotr Dollar. Works for 2D as well as 3D data

ccf = normxcorrn(template,kernel,type)

and type depends on borders (‘valid’,’full’)

## fromBaseWorkSpace

variables are copied from the Matlab base workbase into the function that calls this routine. See chapter 6

fromBaseWorkSpace(vars)

with <vars> a cell-array of variable names

## toBaseWorkSpace

variables are copied from the function to the Matlab base workspace. See Chapter 6

toBaseWorkSpace(vars)

with <vars> a cell-array of variable names

## indices2mat

given the data in vector and indices (for instance <res.ax>) this function returns a matrix with the values at the position of the indices, Other positions in the matrix are set to NaN

mat = indices2mat(data,size,indices)

## mat2indices

Given a matrix, this function returns the values that are not NaN and the indices of these values

[data,indices,size] = mat2indices(mat)

## getNumberWorkers

returns the available number of workers is parallel processing is used. Starts the parallel pool if not already started. Always one worker/core is left for interactive computing

Then there are some functions, not used in the framework but helpful for down sampling

## envelopDownSample

this function uses the original data and GRID to create and envelop DATA file and a GRID file which belongs to the envelop data. Files with the postfix \_ENVELOP are generated

envelopDownSample(preDataFile,postDataFile,gridFile,roiFile,downsamplefactor)

with <downsamplefactor> a vector in format [ax lat ele] given the factor in each direction the original data is down-sampled

## interpResult

this routine interpolates the displacements found on the down-sampled data (and down-sampled points were displacements should be calculated) to displacements for the original displacement points

res = interpResult(ENV\_RESfile,RAW\_RESfile,ENV\_GRIDfile,RAW\_GRIDfile)

# General rules

* the name of user defined functions should be clear and the intention of the function must be in the name e.g. <parabolicPeakFinding>
* naming of variables should be done using the Hungarian convention. First part of variable start lowercase, next words starts uppercase e.g. <thisDummyVariable>
* if a function works only for 2D it should be postfixed with <\_2D>, if it works only for 3D the postfix <\_3D>. For functions that works for 2D as well as 3D no postfix is required. For instance <strainLSQSE\_3D> and <RigidMedianFilter>
* functions written by a researcher and that are intended to be included in the toolbox must be developed using the standard Matlab rules, headers, pre-allocation of data and so one. They should be “green”

# examples

Below two examples are given. The first examples shows how to call the framework, the 2nd how to create a parameter structure

## calling the framework

%% some specific settings

if ispc

dataFolder = 'u:\My Work in Progress\strainTest\strainMusic\';

addpath('u:\My Matlab\strainMusic');

end

if ismac

dataFolder = '/Users/jan/My Work in Progress/strainTest/strainMusic/';

addpath('/Users/jan/My Matlab/strainMusic');

end

PREFILE = [dataFolder 'US\_test3d\_1e-05ax\_disp\_00001.mat'];

POSTFILE = [dataFolder 'US\_test3d\_1e-05ax\_disp\_00002.mat'];

PARAMFILE = [dataFolder 'PARAM\_test3dDisp.mat'];

ROIFILE = [dataFolder 'ROI\_test3d\_1e-05ax\_disp\_00002.mat'];

GRIDFILE = [dataFolder 'GRID\_test3dDisp.mat'];

RESFILE = [dataFolder 'RES\_test3d\_1e-05ax\_00001\_00002.mat'];

%% and run the algorithm

res = strainMusic(PREFILE,POSTFILE,PARAMFILE,ROIFILE,GRIDFILE,RESFILE);

%% and remove some stuff

if ispc

rmpath('u:\My Matlab\strainMusic');

end

if ismac

rmpath('/Users/jan/My Matlab/strainMusic');

end

## param structure

% example to build a PARAM file for 3 compund angles

nrAngles = 3;

no\_iter = 2;

for iAngle = 1:nrAngles

% set the number of iterations and the signaltype. First iteration is

% enveloppe and second RF data (no downsampling)

PARAM{iAngle}.no\_iter = 2;

PARAM{iAngle}.signaltype = {'ENV' 'RAW'};

PARAM{iAngle}.downsampling = [];

% we use in the firt iteration the standard template/kernels as

% programmed in the function <rigidBlockMatch and in the second the

% freeshape mode (as descibed in the thesis of Richard Lopata). Both

% function has a <window> and <max\_disp> parameter

PARAM{iAngle}.ccf\_blockmatch(1).func = @rigidBlockMatch;

PARAM{iAngle}.ccf\_blockmatch(1).additional.window = [13 5];

PARAM{iAngle}.ccf\_blockmatch(1).additional.max\_disp = [220 1];

PARAM{iAngle}.ccf\_blockmatch(1).func = @freeshapeBlockMatch;

PARAM{iAngle}.ccf\_blockmatch(1).additional.window = [6 5];

PARAM{iAngle}.ccf\_blockmatch(1).additional.max\_disp = [110 1];

% for finding the maximum shift we use the normalized cross-correlation

% function as developed by Piotr Dollar. Use all available cores

for i=1:no\_iter

PARAM{iAngle}.ccf\_method(i).func = @normXcorrPiotr;

PARAM{iAngle}.ccf\_method{i).additional.ncores = [];

end

% a standard a parabolicFit function is used for peak interpolation in both

% iterations.

PARAM{iAngle}.ccf\_peakinterp(1).func = @parabolicFit;

PARAM{iAngle}.ccf\_peakinterp(1).additional = [];

PARAM{iAngle}.ccf\_peakinterp(2).func = @parabolicFit;

PARAM{iAngle}.ccf\_peakinterp(2).additional = [];

% we use a filter, implemented in the function <RigidMedian> in both

% axial and lateral direction. Only 3 cores are used. Kernel is given

% in mtr

for i=1:no\_iter

PARAM{iAngle}.axfilter(i).func = @rigidMedianFilter;

PARAM{iAngle}.axfilter(i).additional.kernel = [5.39e-05 2.078e-04];

PARAM{iAngle}.axfilter(i).ncores = 3;

PARAM{iAngle}.latfilter(i).func = @rigidMedianFilter;

PARAM{iAngle}.latfilter(i).additional.kernel = [5.39e-05 2.078e-04];

PARAM{iAngle}.latfilter(i).ncores = 3;

end

% strain is calculated in the last iteration by a LSQSE method,

% implemented in the function <strainLSQSE>

PARAM{iAngle}.postprocessing(1).func = [];

PARAM[iAngle}.postprocessing(1).additional = [];

PARAM{iAngle}.postprocessing(2).func = @strainLSQSE;

PARAM{iAngle}.postprocessing(2).additional.window = [2.7e-05 2.1e-04];

PARAM{iAngle}.postprocessing(2).additional.ncores = [];

% and after all we do a clean up

PARAM{iAngle}.final\_iter\_only = 1;

PARAN{iAngle}.filtered\_disps\_only = 1;

end

## downsampling

% Example to use first iteration on downsampled data

%

% Modifications

% 06-apr-2016 JM initial version

%% Original files

if ispc

dataFolder = 'u:\My Work in Progress\strainTest\strainMusic\';

addpath('u:\My Matlab\strainMusic');

end

if ismac

dataFolder = '/Users/jan/Dropbox/My Work in Progress/strainTest/strainMusic/';

addpath('/Users/jan/Dropbox/My Matlab/strainMusic');

end

PREFILE = [dataFolder 'US\_test3d\_1e-05ax\_disp\_00001.mat'];

POSTFILE = [dataFolder 'US\_test3d\_1e-05ax\_disp\_00002.mat'];

ROIFILE = [dataFolder 'ROI\_test3d\_1e-05ax\_disp\_00002.mat'];

RESFILE = [dataFolder 'RES\_envelop.mat'];

GRIDFILE = [dataFolder 'GRID\_test3dDisp.mat'];

PARAMFILE\_1 = [dataFolder 'PARAM\_envelop.mat'];

PARAMFILE\_2 = [dataFolder 'PARAM\_raw.mat'];

%% downsample (5 times axial, 2 times lateral and eleveational)

envelopDownSample(PREFILE,POSTFILE,GRIDFILE,ROIFILE,[5 2 2]);

%% create parameter files for first iteration

PARAM{1} = [];

PARAM{1}.no\_iter = 1;

PARAM{1}.signaltype{1} = 'RAW';

PARAM{1}.final\_iter\_only = 1;

PARAM{1}.filtered\_disps\_only = 1;

PARAM{1}.ccf\_blockmatch(1).func = @rigidBlockMatch;

PARAM{1}.ccf\_blockmatch(1).additional.window = [10 5 5];

PARAM{1}.ccf\_blockmatch(1).additional.max\_disp = [1 1 1];

PARAM{1}.ccf\_method(1).func = @normXcorrPiotr;

PARAM{1}.ccf\_method(1).additional.ncores = [];

PARAM{1}.ccf\_peakinterp(1).func = [];

PARAM{1}.axfilter(1).func = @rigidMedianFilter;

PARAM{1}.axfilter(1).additional.kernel = [5e-4 5e-4 5e-4];

PARAM{1}.axfilter(1).additional.ncores = [];

PARAM{1}.latfilter(1).func = @rigidMedianFilter;

PARAM{1}.latfilter(1).additional.kernel = [5e-4 5e-4 5e-4];

PARAM{1}.latfilter(1).additional.ncores = [];

PARAM{1}.elefilter(1).func = @rigidMedianFilter;

PARAM{1}.elefilter(1).additional.kernel = [5e-4 5e-4 5e-4];

PARAM{1}.elefilter(1).additional.ncores = [];

PARAM{1}.postprocessing(1).func = [];

save(PARAMFILE\_1,'PARAM');

%% perform the first the first iteration

PREFILE = [dataFolder 'US\_test3d\_1e-05ax\_disp\_env\_00001.mat'];

POSTFILE = [dataFolder 'US\_test3d\_1e-05ax\_disp\_env\_00002.mat'];

ROIFILE = [dataFolder 'ROI\_test3d\_1e-05ax\_disp\_env\_00002.mat'];

GRIDFILE = [dataFolder 'GRID\_test3dDisp\_env.mat'];

strainMusic(PREFILE,POSTFILE,PARAMFILE\_1,ROIFILE,GRIDFILE,RESFILE);

%% interpolate the result file

ENV\_GRIDFILE = [dataFolder 'GRID\_test3dDisp\_envelop.mat'];

ENV\_RESFILE = [dataFolder 'RES\_envelop.mat'];

RAW\_GRIDFILE = [dataFolder 'GRID\_test3dDisp.mat'];

RAW\_RESFILE = [dataFolder 'RES\_raw.mat'];

RAW\_ROIFILE = [dataFolder 'ROI\_test3d\_1e-05ax\_disp\_00002.mat'];

interpResult(ENV\_RESFILE,RAW\_RESFILE,ENV\_GRIDFILE,RAW\_GRIDFILE,RAW\_ROIFILE);

%% and do the other iterations

PARAM{1} = [];

PARAM{1}.no\_iter = 2;

PARAM{1}.signaltype = {'raw' 'raw'};

PARAM{1}.final\_iter\_only = 1;

PARAM{1}.filtered\_disps\_only = 1;

PARAM{1}.ccf\_blockmatch(1).func = [];

PARAM{1}.ccf\_blockmatch(2).func = @rigidBlockMatch;

PARAM{1}.ccf\_blockmatch(2).additional.window = [10 5 5];

PARAM{1}.ccf\_blockmatch(2).additional.max\_disp = [1 1 1];

PARAM{1}.ccf\_method(1).func = [];

PARAM{1}.ccf\_method(2).func = @normXcorrPiotr;

PARAM{1}.ccf\_method(2).additional.ncores = [];

PARAM{1}.ccf\_peakinterp(1).func = [];

PARAM{1}.ccf\_peakinterp(2).func = [];

PARAM{1}.axfilter(1).func = [];

PARAM{1}.latfilter(2).func = [];

PARAM{1}.elefilter(2).func = [];

PARAM{1}.axfilter(2).func = @rigidMedianFilter;

PARAM{1}.axfilter(2).additional.kernel = [5e-4 5e-4 5e-4];

PARAM{1}.axfilter(2).additional.ncores = [];

PARAM{1}.latfilter(2).func = @rigidMedianFilter;

PARAM{1}.latfilter(2).additional.kernel = [5e-4 5e-4 5e-4];

PARAM{1}.latfilter(2).additional.ncores = [];

PARAM{1}.elefilter(2).func = @rigidMedianFilter;

PARAM{1}.elefilter(2).additional.kernel = [5e-4 5e-4 5e-4];

PARAM{1}.elefilter(2).additional.ncores = [];

PARAM{1}.postprocessing(1).func = @replaceDisp;

PARAM{1}.postprocessing(1).additional.file = RAW\_RESFILE;

PARAM{1}.postprocessing(2).func = [];

save(PARAMFILE\_2,'PARAM');

%% and run the algorithm

PREFILE = [dataFolder 'US\_test3d\_1e-05ax\_disp\_00001.mat'];

POSTFILE = [dataFolder 'US\_test3d\_1e-05ax\_disp\_00002.mat'];

ROIFILE = [dataFolder 'ROI\_test3d\_1e-05ax\_disp\_00002.mat'];

GRIDFILE = [dataFolder 'GRID\_test3dDisp.mat'];

RESFILE = [dataFolder 'RES\_final.mat'];

res = strainMusic(PREFILE,POSTFILE,PARAMFILE\_2,ROIFILE,GRIDFILE,RESFILE);

%% done

# References

1. <strain2Dmain> : the first version of the strain software, developed by Richard Lopata, Rik Hansen & Tim Idzenga
2. 2D and 3D Ultrasound Strain Imaging, Methods and *in vivo* Applications. PhD thesis Richard Lopata, 2010
3. File storage and strain parameter definition\_final\_version : Rik Hansen
4. Piotr Computer Vision Matlab Toolbox : Toolbox develop by Piotr Dollar
   1. doc : <http://vision.ucsd.edu/~pdollar/toolbox/doc/index.html>.
   2. downloaded : <https://github.com/pdollar/toolbox>