Manual for LIDAR post-processing in MATLAB

Editor
Markus Schmidt, schmmark@ethz.ch

Date and version
May 16, 2013, version 1.0

Laboratory for Energy Conversion
ETH Zürich

Contents

1 Wake Analysis

This is the section for wake analysis. The wake analysis consists of two sections. The first deals with time averaging. In this case, 2 consequent LIDAR measurements are taken and a time averaged property field of all measurements is interpolated. These properties are velocity, vorticity and wind shear,
The second section, phase averaging, computes the phase of the blades in order to interpolate the same properties as in the time averaging method.

1.1 Time Averaging

The method of measurement is based on 2 separate LIDAR measurements in a 2D-plane at different location. Since the LIDAR is only able to measure radial velocities, this 2 measurements are necessary to interpolate a 2D flow field.
The algorithm presented here takes first both measurements as single ones and interpolates the radial velocities. In a second step, these interpolated fields are used to undergo a vectorial addition and achieve a 2D velocity flow field.
The data is filtered by the following constrains: All measurements below a range of 50 metres are deleted due to the unability of the LIDAR to provide reasonable results. In addition, the threshold of the Doppler-intensity (as a meausure of signal-to-noise ratio) is set to 1.01. The file

1.1.1 `time_average2D`: I/O Definition

1. `[vel_comps] = time_average2D(gridprops, range)`
2. `[vel_comps] = time_average2D(gridprops, range,...`
   `save_figure)`
3. `[vel_comps] = time_average2D(gridprops, range,...`
   `save_figure, lidar_sync, ldm_sync)`
4. `[vel_comps] = time_average2D(gridprops, range,...`
   `save_figure, lidar_sync, ldm_sync, vlimit)`

The numerical input `grid_props` has to be of the format
`[x_min, x_max, y_min, y_max, stepwidth]`. The x-coordinate is defined according to measurement 1. $x = 0$ is the location of LIDAR of measurement 1. The positive x-direction describes the situation when azimuth and elevation are both of 0 degrees. The y direction is the height above the LIDAR. `range` is the distance of 2 data points on the beams in metres. This 2 parametres are compulsory.
If figures are to be computed and saved, `save_figure` has to be set `true`. Note

that the selected folder will be saved in `rootfigures.txt` in the script folder. Deleting of this file will lead to a GUI to select a new location.
Furthermore, the parametres `lidar_sync` and `ldm_sync` can be used to syncronize the measurements of WindRover and LDM timewise. In this case, attention has to be given to the correct timeframes for cropping the measurements. The parametre `vlimit` limits the axis of the colorbars to the given value in positive and negative direction.
During the first run, the function demands locations of the data. These locations are stored in the file `lastpath.txt` in the script folder and will be used in future runs of the function. Deletion of `lastpath.txt` resets the locations.
Note that the meshgrid and interpolated values are according to the function `meshgrid()` in MATLAB.
The output of `time_average2D` is in every case the struct `vel_comps` which has the following structure:

- `x_meshgrid`: x-coordinates in meshgrid format
- `y_meshgrid`: y-coordinates in meshgrid format
- `u_lin`: horizontal wind component (linear)
- `v_lin`: vertical wind component (linear)
- `u_near`: horizontal wind component (nearest neighbour)
- `v_near`: vertical wind component (nearest neighbour)
- `u_nat`: horizontal wind component (natural neighbour)
- `v_nat`: vertical wind component (natural neighbour)

1.1.2 `vorticity2D`: I/O Definition

1. `[omega] = vorticity2D()`
2. `[omega] = vorticity2D(x_mesh, y_mesh, u, v, save_figure,...`
   `omega_limit)`
3. `[omega] = vorticity2D(vel_comps, save_figure,...`
   `omega_limit)`

The function `vorticity2D` has 3 different input methods. The first is without any input variables. A GUI is started to select and import the relevant data.
The second input method is applied for velocity field created by code of Mohsen

Zendehbad. The input parametres `x_mesh` and `y_mesh` do not comply with the requirements in `meshgrid`. Hence this option was necessary.
The third option takes the struct `vel_comps` created by function `time_average2d` as an input.
In addition, the parametre `save_figure` has the same function as in `time_average2D`. Note that the selected folder will be saved in `rootfigures.txt` in the script folder. Deleting of this file will lead to a GUI to select a new location.
During the first run, the function demands locations of the data. These locations are stored in the file `lastpath.txt` in the script folder and will be used in future runs of the function. Deletion of `lastpath.txt` resets the locations.
The function of `omega_limit` is to limit the colorbar axis to the value given. The calculation is done with a centered finite difference method of 2^{nd}(2c) and 4^{th}(4c) order, respectively.
The output for option 1 and 2 is

- `omega`: vorticity field, 2c, according to input grid

The output for option 3 is the struct `omega` with data of vorticity:

- `x_meshgrid`: x-coordinates in meshgrid format
- `y_meshgrid`: y-coordinates in meshgrid format
- `omega_4c_lin`: vorticity field, 4c, linear interpolation
- `omega_4c_near`: vorticity field, 4c, nearest neighbour
- `omega_4c_nat`: vorticity field, 4c, natural neighbour

1.1.3 `shear2D`: I/O Definition

1. `[shear_x, shear_y] = shear2D()`
2. `[shear_x, shear_y] = shear2D(x_mesh, y_mesh, u, v,...`
   `save_figure, shear_limit)`
3. `[shear_x, shear_y] = shear2D(vel_comps, save_figure,...`
   `shear_limit)`

The input parametres are equal to `vorticity2D` and the same rules apply. The output `shear_x` and `shear_y` are defined as

- `shear_x`: Wind shear in x-direction
- `shear_y`: Wind shear in y-direction

1.2 Phase Averaging

Since the regarded turbines all have 3 blades, the first assumption is a repeated wake independend of the position of one blade but their overall configuration. This leads to a period of 120 degrees regarding the position of the blades.
To find the the phase for each beam of the LIDAR measurements, the following algorithm is applied. The LDM data is used to compute the phase. First, the number of revolutions (nor) is calculated with the given trigger points. It is assumed that the nor is constant between 2 consequent datapoints. Second, the trigger point is defined as phase = 0. The phase of one LIDAR beam is then computed by multiplying the nor with the time difference between the last LDM trigger and the time of the beam. With a radians multiplication one achieves the phase in radians.
The phase is then divided into discrete intervals of number `Nphase`. The same internal algorithm are applied to these sets of data then it is done once in `time_average2D`.

1.2.1 `phase_average2D`: I/O Definition

- `[phase_velocity] = phase_average2D(gridprops, range,... Nphase, lidar_sync, ldm_sync)`
- `[phase_velocity] = phase_average2D(gridprops, range,... Nphase, lidar_sync, ldm_sync, save_figure)`
- `[phase_velocity] = phase_average2D(gridprops, range,... Nphase, lidar_sync, ldm_sync, save_figure, limit)`

All description of input parametres of `time_average2D` apply as well for `phase_average2D`. In addition, the parametre `Nphase` defines the number of discrete phase ranges. The output struct `phase_velocity` consists of

- `phase_range`: range of phases in rad
- `x_meshgrid`: x-coordinates in meshgrid format
- `y_meshgrid`: y-coordinates in meshgrid format
- `u_lin`: horizontal wind component (linear)
- `v_lin`: vertical wind component (linear)
- `velocity_2D_lin`: absolute value of wind component (linear)
- `u_near`: horizontal wind component (nearest neighbour)

- `v_near`: vertical wind component (nearest neighbour)
- `velocity_2D_near`: absolute value of wind component (nearest neighbour)
- `u_nat`: horizontal wind component (natural neighbour)
- `v_nat`: vertical wind component (natural neighbour)
- `velocity_2D_nat`: absolute value of wind component (natural neighbour)

1.2.2 `phase_vorticity2D`: I/O Definition

1. `[phase_vorticity] = phase_vorticity2D(phase_velocity)`
2. `[phase_vorticity] = phase_vorticity2D(phase_velocity,...`
   `save_figure)`
3. `[phase_vorticity] = phase_vorticity2D(phase_velocity,...`
   `save_figure, limit)`

The input parametre `phase_velocity` is the output of `phase_average2D`. The parametres `save_figure` and `limit` are used in the same manner as in the previous functions. The output struct `phase_vorticity` is defined as

- `phase_range`: range of phases in rad
- `x_meshgrid`: x-coordinates in meshgrid format
- `y_meshgrid`: y-coordinates in meshgrid format
- `vorticity_2D_lin`: absolute value of wind component (linear)
- `vorticity_2D_near`: absolute value of wind component (nearest neighbour)
- `vorticity_2D_nat`: absolute value of wind component (natural neighbour)

1.2.3 `phase_shear2D`: I/O Definition

1. `[phase_shear] = phase_shear2D(phase_velocity)`
2. `[phase_shear] = phase_shear2D(phase_velocity,...`
   `save_figure)`
3. `[phase_shear] = phase_shear2D(phase_velocity,...`
   `save_figure, shear_limit)`

The input parametre `phase_velocity` is the output of `phase_average2D`. The parametres `save_figure` and `shear_limit` are used in the same manner as in the previous functions. The output struct `phase_shear` is defined as

- `phase_range`: range of phases in rad
- `x_meshgrid`: x-coordinates in meshgrid format
- `y_meshgrid`: y-coordinates in meshgrid format
- `shear_2D_lin`: shear field, 4c, linear interpolation
- `shear_2D_lin_x`: shear field, 4c, linear interpolation, x-derivative
- `shear_2D_lin_y`: shear field, 4c, linear interpolation, y-derivative
- `shear_2D_near`: shear field, 4c, nearest neighbour
- `shear_2D_near_x`: shear field, 4c, nearest neighbour, x-derivative
- `shear_2D_near_y`: shear field, 4c, nearest neighbour, y-derivative
- `shear_2D_nat`: shear field, 4c, natural neighbour
- `shear_2D_nat_x`: shear field, 4c, natural neighbour, x-derivative
- `shear_2D_nat_y`: shear field, 4c, natural neighbour, y-derivative neighbour)

2 Drive and Measure

During the measurement campaign called "Drive and Measure" the WindRover is moving while scanning its environment. The goal is to create a radar like velocity or gradient map on an existing satellite image.

2.1 `drivescan2D`: I/O Definition

1. `[ ] = drivescan2D( start_time, end_time, lidar_home, log_home,... range, map_file, long, lat,scale, color_code)`

2. `[ ] = drivescan2D( start_time, end_time, lidar_home, log_home,... range, map_file, long, lat,scale, color_code, usegradient, limit)`

The parametres `start_time` and `end_time` define the start and end of the measurement in format `[yyyy mm dd hh minmin ss]`. The parametres `lidar_home` and `log_home` are strings containt the folder with LIDAR (main year folder) or GPS files, respectively. The `range` is the distance between 2 datapoints on one laser beam and `map_file` contains the adress of the satellite image. `long` and `lat` contain the GPS coordinates of the reference point on this map, while `scale` is the scale in metres on the map. The parametre `color_code` is s string containing the adress of the color code `.txt.` file.
As an option the parametres `use_gradient` and `limit` can be used. If `usegradient` is `true`, instead of the velocity field the gradient will be computed. The effect of parametre `limit` depends on `usegradient`. For the velocity the axis is set to `caxsis([-limit, limit])` and for gradient to `caxsis([0, limit])`.
Furthermore, during the first run the function asks for a location to save the figures and images. The chosen location is saved in `root_superposer` for following runs of the function. There is no output in MATLAB of the function, but the results are saved as `.fig` and `.png`.

3 Source Code

3.1 time_average2D

```
function [ vel_comps ] = time_average2D(gridprops, range,
    save_figure,...
     lidar_sync,ldm_sync, vlimit)
%% function time_average2D
% [ vel_comps ] = time_average2D(gridprops, range)
% [ vel_comps ] = time_average2D(gridprops, range,
    save_figure)
% [ vel_comps ] = time_average2D(gridprops, range,
    save_figure,...
%       lidar_sync,ldm_sync)
% [ vel_comps ] = time_average2D(gridprops, range,
    save_figure,...
%       lidar_sync,ldm_sync, vlimit)
%
% DESCRIPTION
% The function applies a time averaging to the data
    selected during the
% algorithm. Based on the imported data, the user has to
    choose to time
% intervals which will be declared as measurement one and
    two. As output it
% provides a struct with velocity components dependend on
    their
% interpolation method. In addition, plots will be saved to
     subfolder
% /figures if save_figure is TRUE.
%
% INPUT
%- gridprops: array with properties of final grid: [xmin,
    xmax, ymin, ymax, stepsize]
%- range: radial distance between 2 measurements. Is used
    to resolve the
% range gate of the LIDAR system
%- save_figure: boolean. If TRUE, figures will be saved.
    Default is FALSE
%- lidar_sync: sync time in s to add to the data in LIDAR
    and location
```

```
% - ldm_sync: sync time in s to add to the data in LDM
% - vlimit: absolute value of maximal wind speed to be
    shown. Other values
% will be cropped and set by NaN in the final plots. Zero
    by default
%
% OUTPUT
% - vel_comps: struct with data of the velocity components
%    x_meshgrid: x-coordinates in meshgrid format
%    y_meshgrid: y-coordinates in meshgrid format
%    u_lin: horizontal wind component (linear)
%    v_lin: vertical wind component (linear)
%    u_near: horizontal wind component (nearest neighbour)
%    v_near: vertical wind component (nearest neighbour)
%    u_nat: horizontal wind component (natural neighbour)
%    v_nat: vertical wind component (natural neighbour)
%
% Code by: Markus Schmidt
%
% $Revision: 2.0$ $Date: 2013/05/13 $
%
% This code is licensed under a Creative Commons
    Attribution-ShareAlike
% 3.0 Unported License
% ( http://creativecommons.org/licenses/by-sa/3.0/deed.
    en_GB )

% Global variables
known_times = true;      % Times of measurements are known

% Input Check
if nargin > 6 || nargin < 2
    error('Incorrect number of input arguments')
end

if numel(gridprops) ~= 5
    error('Dimension of gridprops is wrong.')
end

if ~exist('save_figure','var')
```

```
    save_figure = false;
end

if ~exist('lidar_sync','var')
    lidar_sync = 0;
end

if ~exist('ldm_sync','var')
    ldm_sync = 0;
end

%% Import of data
test_series = import_datav2(range,lidar_sync,ldm_sync);

%% Select measurements
% Manually defined start and endpoint for testing the
   algorithm. UI is
% written below.
if known_times
    start1 = [19, 00, 00];  %6.61e4, but missing GPS data
    end1 = [19, 33, 20];    %6.68e4
    start2 = [19, 38, 20];  %6.71e4
    end2 = [20, 26, 40];    %7e4
    [ measure_1, measure_2 ] = ts_selector( test_series,
       start1, end1,...
        start2, end2);
else
    [ measure_1, measure_2 ] = ts_selector(test_series);
end

%% Computation of 2D velocity field
if exist('vlimit','var')
    vel_comps = velocity2D( measure_1, measure_2, gridprops
        , save_figure, vlimit );
else
    vel_comps = velocity2D( measure_1, measure_2, gridprops
        , save_figure);
end
end
```

3.2 vorticity2D

```
function [ omega ] = vorticity2D( varargin )
%% function vorticity2D
% a) function [ omega ] = vorticity2D()
% b) function [ omega ] = vorticity2D(x_mesh, y_mesh, u, v,
     save_figure, omega_limit )
% c) function [ omega ] = vorticity2D( vel_comps,
   save_figure, omega_limit )
%
% DESCRIPTION
% The function has 3 possible input combinations. Option a)
     is without an
% input argument. An assistant is started to find .txt-
   files with suitable
% data according to the requirements of 'txt_files' from
   option b).
%
% Option b) provides an char array with the locations of
   the .txt-files. In
% addition, parametres for save_figure and omega_limit can
   be provided.
%
% The option c) takes an existing 2D velocity field, which
   has an uniform
% grid and interpolated values (with linear, nearest
   neighbour and natural
% neighbour) and computes the vorticity.
% The numerical derivation is solved with the finite
   difference method,
% first with a 2nd order centered (2c) and second with a 4
   th order centered (4c)
% algorithm. Figures can be saved with setting 'save_figure
   ' to true and a
% filtering of the final data via omega_limit.
%
% INPUT b)
% - x_mesh:            x-values in grid format
% - y_mesh:            y-values in grid format
% - u:                 horizontal velocity according to grid
% - v:                 vertical velocity according to grid
```

```
% - save_figure:    boolean. If TRUE, figures will be saved
   . Default is FALSE
% - omega_limit:    absolute value of maximal vorticity to
   be shown. Other values
% will be cropped and set by NaN in the final plots. If not
     set, no
% filtering will be applied.
%
% INPUT c)
% - vel_comps:      struct with data of the velocity
   components
%   x_meshgrid:     x-coordinates in meshgrid format
%   y_meshgrid:     y-coordinates in meshgrid format
%   u_lin:          horizontal wind component (linear)
%   v_lin:          vertical wind component (linear)
%   u_near:         horizontal wind component (nearest
   neighbour)
%   v_near:         vertical wind component (nearest
   neighbour)
%   u_nat:          horizontal wind component (natural
   neighbour)
%   v_nat:          vertical wind component (natural
   neighbour)
% - save_figure:    boolean. If TRUE, figures will be saved
   . Default is FALSE
% - omega_limit:    absolute value of maximal vorticity to
   be shown. Other values
% will be cropped and set by NaN in the final plots. If not
     set, no
% filtering will be applied.
%
% OUTPUT
%   for a) and b) only
%   omega:  vorticity field, 2c, according to input grid
%
%   for c) only
% - omega: struct with data of vorticity
%   x_meshgrid:     x-coordinates in meshgrid format
%   y_meshgrid:     y-coordinates in meshgrid format
```

```
%    omega_4c_lin:    vorticity field, 4c, linear
    interpolation
%    omega_4c_near:   vorticity field, 4c, nearest neighbour
%    omega_4c_nat:    vorticity field, 4c, natural neighbour
%
% Code by: Markus Schmidt
%
% $Revision: 1.1$ $Date: 2013/05/15 $
%
% This code is licensed under a Creative Commons
    Attribution-ShareAlike
% 3.0 Unported License
% ( http://creativecommons.org/licenses/by-sa/3.0/deed.
    en_GB )

% Global settings
close all                               % Close all open
    figures

%% Input check
if nargin>6
    error('Number of input arguments wrong.')
end

% Check wether input is present. If not, start import
    function
if nargin ==0
    [ x_mesh, y_mesh, u, v ] = velcomp2D_load();

    prompt = 'Do you want figures? Y/N [N]: ';
    infigure = input(prompt,'s');
    if isempty(infigure) || infigure == 'N' || infigure ==
        'n'
        save_figure = false;
    elseif infigure == 'Y' || infigure == 'y'
        save_figure = true;
    else
        error('Input of save_figure must be boolean.')
    end
```

```
prompt = 'Set limit for omega [none]: ';
inomega = input(prompt);
if ~isempty(inomega) && isnumeric(inomega)
    omega_limit = inomega;
elseif ~isnumeric(inomega)
    error('Input of omega_limit must be numeric.')
end
clear infigure inomega
isimport = true;          % Boolean to decide how many
   calculations

% Check if grid is provided according to requirements
   in meshgrid
% Check if ngrid is present
if x_mesh(1,1) > x_mesh(end,1)
    x_mesh = flipud(x_mesh);
    u = flipud(u);
    v = flipud(v);
    flipped_ud = true;
end
if (x_mesh(1,2)- x_mesh(1,1)) == 0
    x_mesh = transpose(x_mesh);
    u = transpose(u);
    v = transpose(v);
    transposed = true;
end
if y_mesh(1,1) > y_mesh(1,end)
    y_mesh = fliplr(y_mesh);
    u = fliplr(u);
    v = fliplr(v);
    flipped_lr = true;
end
if (y_mesh(2,1)- y_mesh(1,1)) == 0
    y_mesh = transpose(y_mesh);
    if ~exist('transposed','var')
    u = transpose(u);
    v = transpose(v);
    transposed = true;
    end
end
```

```
% Check wether input is struct
elseif isstruct(varargin{1})
    vel_comps = varargin{1};
    isimport = false;

    % Check correct size and alignment of meshgrid
    if size(vel_comps.x_meshgrid) ~= size(vel_comps.
       y_meshgrid)
        error('Input meshgrids are not of same size.')
    end
    % Check if grid is provided according to requirements
       in meshgrid
    if (vel_comps.x_meshgrid(1,2)- vel_comps.x_meshgrid
       (1,1)) == 0
        temp = vel_comps.x_meshgrid;
        vel_comps.x_meshgrid = vel_comps.y_meshgrid;
        vel_comps.y_meshgrid = temp;
        clear temp
    end
    if nargin >= 2
    save_figure = varargin{2};
    end
    if nargin == 3
    omega_limit = varargin{3};
    end
% Check wether input is char array.
elseif ismatrix(varargin{1})
    x_mesh = varargin{1};
    y_mesh = varargin{2};
    u      = varargin{3};
    v      = varargin{4};
    isimport = true;          % Boolean to decide how many
       calculations
    if nargin >= 5
        save_figure = varargin{5};
    end
    if nargin == 6
        omega_limit = varargin{6};
    end
```

```
    % Check if grid is provided according to requirements
       in meshgrid
    % Check if ngrid is present
    if x_mesh(1,1) > x_mesh(end,1)
        x_mesh = flipud(x_mesh);
        u = flipud(u);
        v = flipud(v);
        flipped_ud = true;
    end
    if (x_mesh(1,2)- x_mesh(1,1)) == 0
        x_mesh = transpose(x_mesh);
        u = transpose(u);
        v = transpose(v);
        transposed = true;
    end
    if y_mesh(1,1) > y_mesh(1,end)
        y_mesh = fliplr(y_mesh);
        u = fliplr(u);
        v = fliplr(v);
        flipped_lr = true;
    end
    if (y_mesh(2,1)- y_mesh(1,1)) == 0
        y_mesh = transpose(y_mesh);
        if ~exist('transposed','var')
        u = transpose(u);
        v = transpose(v);
        transposed = true;
        end
    end

% If input is not correct, abort function
else
    error('Input does not have right format.')
end

% Check and set variable 'save_figure'
if ~exist('save_figure','var')
    save_figure = 0;
end
```

```
%% Numerical Derivation with Finite-difference 2nd order,
   central
if isimport
      omega = derivation_2c(x_mesh,y_mesh,u,v);
else
    omega_2c_lin = derivation_2c(vel_comps.x_meshgrid,
       vel_comps.y_meshgrid,...
        vel_comps.u_lin  ,vel_comps.v_lin);
    omega_2c_near = derivation_2c(vel_comps.x_meshgrid,
       vel_comps.y_meshgrid,...
        vel_comps.u_near  ,vel_comps.v_near);
    omega_2c_nat = derivation_2c(vel_comps.x_meshgrid,
       vel_comps.y_meshgrid,...
        vel_comps.u_nat  ,vel_comps.v_nat);
end

%% Numerical Derivation with Finite-difference 4th order,
   central
if ~isimport
    omega_4c_lin = derivation_4c(vel_comps.x_meshgrid,
       vel_comps.y_meshgrid,...
        vel_comps.u_lin  ,vel_comps.v_lin);
    omega_4c_near = derivation_4c(vel_comps.x_meshgrid,
       vel_comps.y_meshgrid,...
        vel_comps.u_near  ,vel_comps.v_near);
    omega_4c_nat = derivation_4c(vel_comps.x_meshgrid,
       vel_comps.y_meshgrid,...
        vel_comps.u_nat  ,vel_comps.v_nat);
end

%% Plot results
if save_figure
    % Set properties for plot loop
    % Define location to save figures
    if exist('rootfigures.txt','file')
        import = importdata('rootfigures.txt');
        rootfigures = cell2mat(import(1,1));
    else
        % UI for selecting the data folders
        disp('Please select the root folder for figures.')
```

```
        rootfigures = uigetdir('','Please select the root 
            folder for figures.');

        % Save path to txt-file
        fileID = fopen('rootfigures.txt','wt');
        fprintf(fileID, '%s\n', rootfigures);
        fclose(fileID);
    end

    % Check existence of folder figures. If not present,
        create one
    latest = 1;
    scriptfolder = pwd;
    if exist(rootfigures,'dir')~=7
        mkdir(rootfigures);
    elseif size(dir(rootfigures),1) > 2    % Check wether
        folder is empty
        % Check numbering of figures. If figures are
            already present, the number
        % should increase by one number
        cd(rootfigures);
        list = dir('vorticity2d*');
        list = struct2cell(list);
        list = list(1,:);              % Crop list to
            filenames
        list = char(list);
        A = NaN(size(list,1),1);
        for k = 1:size(list,1)
            temp = strsplit(list(k,:), {'-', '.'},'
                CollapseDelimiters',true);
            A(k) = str2num(cell2mat(temp(2)));
        end
        latest = max(A) + 1;
        if isempty('latest')
            error('Existing plots in figures could not be 
                identified.')
        end
        cd(scriptfolder)
    end
```

```
disp(['Vorticity figures will be saved with number ',
    num2str(latest), '.']);
% Function handels
printeps = @(fname) print('-depsc2',... % print eps
    file
      fullfile(rootfigures,sprintf('%s-%02d',fname,latest
         )), '-r300');

savefig = @(fname) hgsave(gcf,...         % Save figure
      fullfile(rootfigures,sprintf('%s-%02d',fname,latest
         )));

set(0,...                                 % figure
    settings
      'DefaultFigureColormap',gray,...
      'DefaultAxesVisible','on',...
      'DefaultAxesNextPlot','add',...
      'DefaultAxesFontSize',10);

if isimport
     ssource = {omega};
     sfilename = {'vorticity2d_2c'};
     stitle = {'vorticity in 2D plane (2c)'};
else
     ssource = {omega_2c_lin, omega_2c_near,
        omega_2c_nat,...
         omega_4c_lin, omega_4c_near, omega_4c_nat};
     sfilename = {'vorticity2d_lin_2c','
        vorticity2d_near_2c',...
          'vorticity2d_nat_2c','vorticity2d_lin_4c','
             vorticity2d_near_4c',...
          'vorticity2d_nat_4c'};
     stitle = {'vorticity in 2D plane (Linear
        Interpolation, 2c)',...
          'vorticity in 2D plane (Nearest Neighbour, 2c)'
             ,...
          'vorticity in 2D plane (Natural Neighbour, 2c)'
             ,...
          'vorticity in 2D plane (Linear Interpolation, 4
             c)',...
```

```
            'vorticity in 2D plane (Nearest Neighbour, 4c)'
                ,...
            'vorticity in 2D plane (Natural Neighbour, 4c)'
                };
        x_mesh = vel_comps.x_meshgrid;
        y_mesh = vel_comps.y_meshgrid;
    end

    % Save plot entries to struct
    plots = struct('source', ssource, 'filename', sfilename
        , 'title', stitle);

    % Plot the struct
    for k=1:size(plots,2)
        set(gcf,'PaperUnits', 'centimeters', 'PaperType', '
            A4',...
            'PaperOrientation','portrait','
                PaperPositionMode', 'manual',...
            'PaperPosition', [2 1 27.7 21])
        contourf(x_mesh, y_mesh, plots(k).source)
        title(plots(k).title)
        xlabel('horizontal distance to LIDAR in m')
        ylabel('vertical distance to LIDAR in m')
        colorbar('location','SouthOutside')
        colormap('jet')
        if exist('omega_limit','var')
            caxis([-abs(omega_limit) abs(omega_limit)])
        end
        printeps(plots(k).filename);
        savefig(plots(k).filename);
        hfigure(gcf) = figure(gcf+1);
    end
end
close all

%% Create output struct
% - omega: struct with data of vorticity
%   x_meshgrid:      x-coordinates in meshgrid format
%   y_meshgrid:      y-coordinates in meshgrid format
```

```
%   omega_2c_lin:   vorticity field, 2c, linear
    interpolation
%   omega_2c_near:  vorticity field, 2c, nearest neighbour
%   omega_2c_nat:   vorticity field, 2c, natural neighbour
%   omega_4c_lin:   vorticity field, 4c, linear
    interpolation
%   omega_4c_near:  vorticity field, 4c, nearest neighbour
%   omega_4c_nat:   vorticity field, 4c, natural neighbour
if ~isimport
    omega = struct(...
        'x_meshgrid', vel_comps.x_meshgrid, 'y_meshgrid',
            vel_comps.y_meshgrid,...
        'omega_2c_lin', omega_2c_lin, 'omega_2c_near',
            omega_2c_near,...
        'omega_2c_nat', omega_2c_nat, 'omega_4c_lin',
            omega_4c_lin,...
        'omega_4c_near', omega_4c_near, 'omega_4c_nat',
            omega_4c_nat);
else
    % Transform result back to original mesh
    if exist('transposed','var')
        omega = transpose(omega);
    end
    if exist('flipped_ud','var')
        omega = flipud(omega);
    end
    if exist('flipped_lr','var')
        omega = fliplr(omega);
    end
end

end

function [ omega_2c ] = derivation_2c(x_mesh, y_mesh, u ,v)
% Set up matrix for numerical derivation
delta_x = x_mesh(1,2)- x_mesh(1,1);
delta_y = y_mesh(2,1)- y_mesh(1,1);

% Calculate derivaties du/dy and dv/dx
uy_2c = NaN(size(x_mesh));
```

```
vx_2c = NaN(size(x_mesh));

%Fill matrix with values
% Formula for algorithm: u' = (-u_(i-1) + u_(i+1))/(2*
    delta_X)
for k=2:size(u,1)-1          % Borders cannot be calculated
    for l=1:size(u,2)
        uy_2c(k,l) = (-u(k-1,l) + u(k+1,l))/(2*delta_y);
    end
end

for k=1:size(u,1)
    for l=2:size(u,2)-1      % Borders cannot be calculated
        vx_2c(k,l) = (-v(k,l-1) + v(k,l+1))/(2*delta_x);
    end
end
% Output
omega_2c = -uy_2c + vx_2c;
end

function [ omega_4c ] = derivation_4c(x_mesh, y_mesh, u ,v)
% Set up matrix for numerical derivation
delta_x = x_mesh(1,2)- x_mesh(1,1);
delta_y = y_mesh(2,1)- y_mesh(1,1);

% Calculate derivaties du/dy and dv/dx
uy_4c = NaN(size(x_mesh));
vx_4c = NaN(size(x_mesh));

%Fill matrix with values
% % Formula for algorithm:
% u' = (u_(i-2) - 8*u_(i-1)+8*u_(i+1)-u_(j+2))/(12*delta_X)
for k=3:size(u,1)-2          % Borders cannot be calculated
    for l=1:size(u,2)
        uy_4c(k,l) = (u(k-2,l) - 8*u(k-1,l) + 8*u(k+1,l)- u
            (k+2,l))/(12*delta_y);
    end
end

for k=1:size(u,1)
```

```
    for l=3:size(u,2)-2      % Borders cannot be calculated
        vx_4c(k,l) = (v(k,l-2) - 8*v(k,l-1) + 8*v(k,l+1)- v
            (k,l+2))/(12*delta_x);
    end
end

% Output
omega_4c = -uy_4c + vx_4c;
end
```

3.3 shear2D

```
function [ shear_x, shear_y ] = shear2D( varargin )
%% function shear2D
% a) function [ shear_x, shear_y ] = shear2D()
% b) function [ shear_x, shear_y ] = shear2D(x_mesh, y_mesh
    , u, v, save_figure, shear_limit )
% c) function [ shear_x, shear_y ] = shear2D( vel_comps,
    save_figure, shear_limit )
%
% DESCRIPTION
% The function has 3 possible input combinations. Option a)
     is without an
% input argument. An assistant is started to find .txt-
    files with suitable
% data according to the requirements of 'txt_files' from
    option b).
%
% Option b) provides an char array with the locations of
    the .txt-files. In
% addition, parametres for save_figure and shear_limit can
    be provided.
%
% The option c) takes an existing 2D velocity field, which
    has an uniform
% grid and interpolated values (with linear, nearest
    neighbour and natural
% neighbour) and computes the wind shear.
% The numerical derivation is solved with the finite
    difference method,
```

```
% with a 4th order centered (4c) algorithm. Figures can be
    saved with setting 'save_figure' to true and a
% filtering of the final data via shear_limit.
%
% INPUT b)
% - x_mesh:          x-values in grid format
% - y_mesh:          y-values in grid format
% - u:               horizontal velocity according to grid
% - v:               vertical velocity according to grid
% - save_figure:     boolean. If TRUE, figures will be saved
    . Default is FALSE
% - shear_limit:     absolute value of maximal shear to be
    shown. Other values
% will be cropped and set by NaN in the final plots. If not
     set, no
% filtering will be applied.
%
% INPUT c)
% - vel_comps:       struct with data of the velocity
    components
%   x_meshgrid:      x-coordinates in meshgrid format
%   y_meshgrid:      y-coordinates in meshgrid format
%   u_lin:           horizontal wind component (linear)
%   v_lin:           vertical wind component (linear)
%   u_near:          horizontal wind component (nearest
    neighbour)
%   v_near:          vertical wind component (nearest
    neighbour)
%   u_nat:           horizontal wind component (natural
    neighbour)
%   v_nat:           vertical wind component (natural
    neighbour)
% - save_figure:     boolean. If TRUE, figures will be saved
    . Default is FALSE
% - shear_limit:     absolute value of maximal shear to be
    shown. Other values
% will be cropped and set by NaN in the final plots. If not
     set, no
% filtering will be applied.
%
```

```
% OUTPUT
%   shear_x: wind shear in x-direction
%   shearyx: wind shear in y-direction
%
% Code by: Markus Schmidt
%
% $Revision: 0.2$ $Date: 2013/04/10 $
%
% This code is licensed under a Creative Commons
    Attribution-ShareAlike
% 3.0 Unported License
% ( http://creativecommons.org/licenses/by-sa/3.0/deed.
    en_GB )

% Global settings
close all                                   % Close all open
    figures

%% Input check
if nargin>6
    error('Number of input arguments wrong.')
end

% Check wether input is present. If not, start import
    function
if nargin ==0
    [ x_mesh, y_mesh, u, v ] = velcomp2D_load();

    prompt = 'Do you want figures? Y/N [N]: ';
    infigure = input(prompt,'s');
    if isempty(infigure) || infigure == 'N' || infigure ==
        'n'
        save_figure = false;
    elseif infigure == 'Y' || infigure == 'y'
        save_figure = true;
    else
        error('Input of save_figure must be boolean.')
    end

    prompt = 'Set limit for shear [none]: ';
```

```
    inshear = input(prompt);
    if ~isempty(inshear) && isnumeric(inshear)
        shear_limit = inshear;
    elseif ~isnumeric(inshear)
        error('Input of shear_limit must be numeric.')
    end
    clear infigure inshear
    isimport = true;          % Boolean to decide how many
       calculations

    % Check if grid is provided according to requirements
       in meshgrid
    % Check if ngrid is present
    if x_mesh(1,1) > x_mesh(end,1)
        x_mesh = flipud(x_mesh);
        u = flipud(u);
        v = flipud(v);
        flipped_ud = true;
    end
    if (x_mesh(1,2)- x_mesh(1,1)) == 0
        x_mesh = transpose(x_mesh);
        u = transpose(u);
        v = transpose(v);
        transposed = true;
    end
    if y_mesh(1,1) > y_mesh(1,end)
        y_mesh = fliplr(y_mesh);
        u = fliplr(u);
        v = fliplr(v);
        flipped_lr = true;
    end
    if (y_mesh(2,1)- y_mesh(1,1)) == 0
        y_mesh = transpose(y_mesh);
        if ~exist('transposed','var')
        u = transpose(u);
        v = transpose(v);
        transposed = true;
        end
    end
% Check wether input is char array.
```

```
elseif isstruct(varargin{1})
    vel_comps = varargin{1};
    isimport = false;

    % Check correct size and alignment of meshgrid
    if size(vel_comps.x_meshgrid) ~= size(vel_comps.
       y_meshgrid)
        error('Input meshgrids are not of same size.')
    end
    % Check if grid is provided according to requirements
       in meshgrid
    if (vel_comps.x_meshgrid(1,2)- vel_comps.x_meshgrid
       (1,1)) == 0
        temp = vel_comps.x_meshgrid;
        vel_comps.x_meshgrid = vel_comps.y_meshgrid;
        vel_comps.y_meshgrid = temp;
        clear temp
    end
    if nargin >= 2
    save_figure = varargin{2};
    end
    if nargin == 3
    shear_limit = varargin{3};
    end

% Check wether input is char array.
elseif isnumeric(varargin{1})
    x_mesh = varargin{1};
    y_mesh = varargin{2};
    u      = varargin{3};
    v      = varargin{4};
    isimport = true;          % Boolean to decide how many
       calculations
    if nargin >= 5
        save_figure = varargin{5};
    end
    if nargin == 6
        shear_limit = varargin{6};
    end
```

```
    % Check if grid is provided according to requirements
        in meshgrid
    % Check if ngrid is present
    if x_mesh(1,1) > x_mesh(end,1)
        x_mesh = flipud(x_mesh);
        u = flipud(u);
        v = flipud(v);
        flipped_ud = true;
    end
    if (x_mesh(1,2)- x_mesh(1,1)) == 0
        x_mesh = transpose(x_mesh);
        u = transpose(u);
        v = transpose(v);
        transposed = true;
    end
    if y_mesh(1,1) > y_mesh(1,end)
        y_mesh = fliplr(y_mesh);
        u = fliplr(u);
        v = fliplr(v);
        flipped_lr = true;
    end
    if (y_mesh(2,1)- y_mesh(1,1)) == 0
        y_mesh = transpose(y_mesh);
        if ~exist('transposed','var')
        u = transpose(u);
        v = transpose(v);
        transposed = true;
        end
    end
% If input is not correct, abort function
else
    error('Input does not have right format.')
end

% Check and set variable 'save_figure'
if ~exist('save_figure','var')
    save_figure = 0;
end
```

```
%% Numerical Derivation with Finite-difference 2nd order,
    central
if isimport
        [shear_x, shear_y] = derivation_2c(x_mesh,y_mesh,u,v)
            ;
        % Calculate magnitude
        shear_mag = sqrt(shear_x.^2 + shear_y.^2);
else
    [shear_2c_lin_x, shear_2c_lin_y] = derivation_2c(
        vel_comps.x_meshgrid,...
         vel_comps.y_meshgrid,vel_comps.u_lin  ,vel_comps.
             v_lin);
    [shear_2c_near_x, shear_2c_near_y] = derivation_2c(
        vel_comps.x_meshgrid,...
         vel_comps.y_meshgrid, vel_comps.u_near  ,vel_comps.
             v_near);
    [shear_2c_nat_x, shear_2c_nat_y] = derivation_2c(
        vel_comps.x_meshgrid,...
         vel_comps.y_meshgrid, vel_comps.u_nat  ,vel_comps.
             v_nat);

    % Calculate magnitude
    shear_2c_lin_mag = sqrt(shear_2c_lin_x.^2 +
        shear_2c_lin_y.^2);
    shear_2c_near_mag = sqrt(shear_2c_near_x.^2 +
        shear_2c_near_y.^2);
    shear_2c_nat_mag = sqrt(shear_2c_nat_x.^2 +
        shear_2c_nat_y.^2);
end
%% Numerical Derivation with Finite-difference 4th order,
    central
if ~isimport
    [shear_4c_lin_x, shear_4c_lin_y] = derivation_4c(
        vel_comps.x_meshgrid,...
         vel_comps.y_meshgrid,vel_comps.u_lin  ,vel_comps.
             v_lin);
    [shear_4c_near_x, shear_4c_near_y] = derivation_4c(
        vel_comps.x_meshgrid,...
         vel_comps.y_meshgrid, vel_comps.u_near  ,vel_comps.
             v_near);
```

```
    [shear_4c_nat_x, shear_4c_nat_y] = derivation_4c(
        vel_comps.x_meshgrid,...
        vel_comps.y_meshgrid, vel_comps.u_nat ,vel_comps.
            v_nat);

    % Calculate magnitude
    shear_4c_lin_mag = sqrt(shear_4c_lin_x.^2 +
        shear_4c_lin_y.^2);
    shear_4c_near_mag = sqrt(shear_4c_near_x.^2 +
        shear_4c_near_y.^2);
    shear_4c_nat_mag = sqrt(shear_4c_nat_x.^2 +
        shear_4c_nat_y.^2);
end

%% Plot results
if save_figure
    % Set properties for plot loop
    % Define location to save figures
    if exist('rootfigures.txt','file')
        import = importdata('rootfigures.txt');
        rootfigures = cell2mat(import(1,1));
    else
        % UI for selecting the data folders
        disp('Please select the root folder for figures.')
        rootfigures = uigetdir('','Please select the root 
            folder for figures.');

        % Save path to txt-file
        fileID = fopen('rootfigures.txt','wt');
        fprintf(fileID, '%s\n', rootfigures);
        fclose(fileID);
    end

    % Check existence of folder figures. If not present,
        create one
    latest = 1;
    scriptfolder = pwd;
    if exist(rootfigures,'dir')~=7
        mkdir(rootfigures);
```

```
elseif size(dir(rootfigures),1) > 2    % Check wether
    folder is empty
     % Check numbering of figures. If figures are
         already present, the number
     % should increase by one number
     cd(rootfigures);
     list = dir('shear2d*');
     list = struct2cell(list);
     list = list(1,:);              % Crop list to
         filenames
     list = char(list);
     A = NaN(size(list,1),1);
     for k = 1:size(list,1)
          temp = strsplit(list(k,:), {'-', '.'},'
              CollapseDelimiters',true);
          A(k) = str2num(cell2mat(temp(2)));
     end
     latest = max(A) + 1;
     if isempty('latest')
          error('Existing plots in figures could not be
              identified.')
     end
     cd(scriptfolder)
end
disp(['Vorticity figures will be saved with number ',
    num2str(latest), '.']);
% Function handels
printeps = @(fname) print('-depsc2',... % print eps
    file
     fullfile(rootfigures,sprintf('%s-%02d',fname,latest
         )), '-r300');

savefig = @(fname) hgsave(gcf,...          % Save figure
     fullfile(rootfigures,sprintf('%s-%02d',fname,latest
         )));

set(0,...                                  % figure
    settings
     'DefaultFigureColormap',gray,...
     'DefaultAxesVisible','on',...
```

```
        'DefaultAxesNextPlot','add',...
        'DefaultAxesFontSize',10);

    % Set properties for plot loop
    if isimport
        quivx_source = {shear_x};
        quivy_source = {shear_y};
        cont_source = {shear_mag};
        sfilename = {'shear2d_2c'};
        stitle = {'shear in 2D plane (2c)'};
    else
        quivx_source = {shear_2c_lin_x, shear_2c_near_x,
           shear_2c_nat_x,...
            shear_4c_lin_x, shear_4c_near_x,shear_2c_nat_x
               };
        quivy_source = {shear_2c_lin_y, shear_2c_near_y,
           shear_2c_nat_y,...
            shear_4c_lin_y, shear_4c_near_y,shear_2c_nat_y
               };
        cont_source = {shear_2c_lin_mag, shear_2c_near_mag,
            shear_2c_nat_mag,...
            shear_4c_lin_mag, shear_4c_near_mag,
               shear_4c_nat_mag};
        sfilename = {'shear2d_lin_2c','shear2d_near_2c',...
            'shear2d_nat_2c','shear2d_lin_4c','
               shear2d_near_4c',...
            'shear2d_nat_4c'};
        stitle = {'shear in 2D plane (Linear Interpolation,
           2c)',...
            'shear in 2D plane (Nearest Neighbour, 2c)',...
            'shear in 2D plane (Natural Neighbour, 2c)',...
            'shear in 2D plane (Linear Interpolation, 4c)'
               ,...
            'shear in 2D plane (Nearest Neighbour, 4c)',...
            'shear in 2D plane (Natural Neighbour, 4c)'};
        x_mesh = vel_comps.x_meshgrid;
        y_mesh = vel_comps.y_meshgrid;
    end

    % Save plot entries to struct
```

```
    plots = struct('cont_source', cont_source,'quivx_source
       ', quivx_source,...
        'quivy_source', quivy_source,'filename', sfilename,
            'title', stitle);

    % Plot the struct
    for k=1:size(plots,2)
        set(gcf,'PaperUnits', 'centimeters', 'PaperType', '
           A4',...
            'PaperOrientation','portrait','
               PaperPositionMode', 'manual',...
            'PaperPosition', [2 1 27.7 21])
        contourf(x_mesh, y_mesh, plots(k).cont_source)
        quiver(x_mesh, y_mesh, plots(k).quivx_source,plots(
           k).quivy_source)
        title(plots(k).title)
        xlabel('horizontal distance to LIDAR in m')
        ylabel('vertical distance to LIDAR in m')
        colorbar('location','SouthOutside')
        colormap('jet')
        if exist('shear_limit','var')
            caxis([0 abs(shear_limit)])
        end
        printeps(plots(k).filename);
        savefig(plots(k).filename);
        hfigure(gcf) = figure(gcf+1);
    end
end
close all

%% Create output

if ~isimport
    shear_x = shear_2c_lin_x;
    shear_y = shear_2c_lin_y;
else
    % Transform result back to original mesh
    if exist('transposed','var')
        shear_x = transpose(shear_x);
        shear_y = transpose(shear_y);
```

```
    end
    if exist('flipped_ud','var')
        shear_x = flipud(shear_x);
        shear_y = flipud(shear_y);
    end
    if exist('flipped_lr','var')
        shear_x = fliplr(shear_x);
        shear_y = fliplr(shear_y);
    end
end

end

function [shear_x, shear_y] = derivation_2c(x_mesh, y_mesh
    , u ,v)
% Set up matrix for numerical derivation
delta_x = x_mesh(1,2)- x_mesh(1,1);
delta_y = y_mesh(2,1)- y_mesh(1,1);

% Grid check
if delta_x == 0
    error('Wrong x-mesh as input.')
end

if delta_y == 0
    error('Wrong y-mesh as input.')
end

% Calculate derivaties du/dx, du/dy, dv/dx and dv/dy
ux_2c = NaN(size(x_mesh));
uy_2c = NaN(size(x_mesh));
vx_2c = NaN(size(x_mesh));
vy_2c = NaN(size(x_mesh));

%Fill matrix with values
% Formula for algorithm: u' = (-u_(i-1) + u_(i+1))/(2*
    delta_X)

% For du/dx
for k=1:size(u,1)
```

```
    for l=2:size(u,2)-1       % Borders cannot be calculated
        ux_2c(k,l) = (-u(k,l-1) + u(k,l+1))/(2*delta_x);
    end
end

% For du/dy
for k=2:size(u,1)-1           % Borders cannot be calculated
    for l=1:size(u,2)
        uy_2c(k,l) = (-u(k-1,l) + u(k+1,l))/(2*delta_y);
    end
end

% For dv/dx
for k=1:size(u,1)
    for l=2:size(u,2)-1       % Borders cannot be calculated
        vx_2c(k,l) = (-v(k,l-1) + v(k,l+1))/(2*delta_x);
    end
end

% For dv/dy
for k=2:size(u,1)-1           % Borders cannot be calculated
    for l=1:size(u,2)
        vy_2c(k,l) = (-v(k-1,l) + v(k+1,l))/(2*delta_y);
    end
end

% Output
shear_x = ux_2c + vx_2c;
shear_y = uy_2c + vy_2c;
end

function [ shear_x, shear_y ] = derivation_4c(x_mesh,
   y_mesh, u ,v)
% Set up matrix for numerical derivation
delta_x = x_mesh(1,2)- x_mesh(1,1);
delta_y = y_mesh(2,1)- y_mesh(1,1);

% Grid check
if delta_x == 0
    error('Wrong x-mesh as input.')
```

```
end

if delta_y == 0
    error('Wrong y-mesh as input.')
end

% Calculate derivaties du/dx, du/dy, dv/dx and dv/dy
ux_4c = NaN(size(x_mesh));
uy_4c = NaN(size(x_mesh));
vx_4c = NaN(size(x_mesh));
vy_4c = NaN(size(x_mesh));

%Fill matrix with values
% % Formula for algorithm:
% u' = (u_(i-2) - 8*u_(i-1)+8*u_(i+1)-u_(j+2))/(12*delta_X)

% For du/dx
for k=1:size(u,1)
    for l=3:size(u,2)-2      % Borders cannot be calculated
        ux_4c(k,l) = (u(k,l-2) - 8*u(k,l-1) + 8*u(k,l+1)- u
            (k,l+2))/(12*delta_x);
    end
end

% For du/dy
for k=3:size(u,1)-2          % Borders cannot be calculated
    for l=1:size(u,2)
        uy_4c(k,l) = (u(k-2,l) - 8*u(k-1,l) + 8*u(k+1,l)- u
            (k+2,l))/(12*delta_y);
    end
end

% For dv/dx
for k=1:size(u,1)
    for l=3:size(u,2)-2      % Borders cannot be calculated
        vx_4c(k,l) = (v(k,l-2) - 8*v(k,l-1) + 8*v(k,l+1)- v
            (k,l+2))/(12*delta_x);
    end
end
```

```
% For dv/dy
for k=3:size(u,1)-2          % Borders cannot be calculated
    for l=1:size(u,2)
        vy_4c(k,l) = (v(k-2,l) - 8*v(k-1,l) + 8*v(k+1,l)- v
            (k+2,l))/(12*delta_y);
    end
end

% Output
shear_x = ux_4c + vx_4c;
shear_y = uy_4c + vy_4c;
end
```

3.4 phase_vorticity2D

```
function [ phase_vorticity ] = phase_vorticity2D(
    phase_velocity, save_figure, limit )
%% function phase_vorticity2D
% [ phase_vorticity ] = phase_vorticity2D( phase_velocity )
% [ phase_vorticity ] = phase_vorticity2D( phase_velocity,
    save_figure )
% [ phase_vorticity ] = phase_vorticity2D( phase_velocity,
    save_figure, limit )
%
% DESCRIPTION
% The function takes an input struct phase_velocity,
    computed by function
% phase_average2D, and calculates the vorticity with a
    Finite difference
% method of 4th order. If save_figure is set, the function
    saves figures of
% each result in a subfolder of rootfigures. The input '
    limit' can limit
% the shown vorticity range in those figures.
%
% INPUT
% - phase_velocity: struct with data of the velocity
    components
%         phase_range: range of phases in rad
%         x_meshgrid: x-coordinates in meshgrid format
```

```
%         y_meshgrid: y-coordinates in meshgrid format
%         u_lin: horizontal wind component (linear)
%         v_lin: vertical wind component (linear)
%         velocity_2D_lin: absolute value of wind component (
    linear)
%         u_near: horizontal wind component (nearest
    neighbour)
%         v_near: vertical wind component (nearest neighbour)
%         velocity_2D_near: absolute value of wind component
    (nearest neighbour)
%         u_nat: horizontal wind component (natural neighbour
    )
%         v_nat: vertical wind component (natural neighbour)
%         velocity_2D_nat: absolute value of wind component (
    natural neighbour)
%- save_figure: boolean. If TRUE, figures will be saved.
    Default is FALSE
%- limit: absolute value of maximal vorticity to be shown
    in plots. If not
% set, no limit will be applied to the final plots.
%
% OUTPUT
%- phase_vorticity: struct with data of the velocity
    components
%         phase_range: range of phases in rad
%         x_meshgrid: x-coordinates in meshgrid format
%         y_meshgrid: y-coordinates in meshgrid format
%         vorticity_2D_lin: absolute value of wind component
    (linear)
%         vorticity_2D_near: absolute value of wind component
     (nearest neighbour)
%         vorticity_2D_nat: absolute value of wind component
    (natural neighbour)
%
% Code by: Markus Schmidt
%
% $Revision: 0.2$ $Date: 2013/05/15 $
%
% This code is licensed under a Creative Commons
    Attribution-ShareAlike
```

```
% 3.0 Unported License
% ( http://creativecommons.org/licenses/by-sa/3.0/deed.
    en_GB )

close all

% Global variables
interp_method = 3;            % 1: Linear, 2: Nearest N.,3:
    Natural N.

% Input Check
if nargin > 3 || nargin == 0
    error('Incorrect number of input arguments')
end

if ~exist('save_figure','var')
    save_figure = false;
end

scriptfolder = pwd;

%% Numerical derivation of vorticity
% Compute number of discrete phases
Nphase = size(phase_velocity,2);

for k=1:Nphase
    % % Numerical Derivation with Finite-difference 2nd
        order, central
    % omega_2c_lin = derivation_2c(phase_velocity.
        x_meshgrid, phase_velocity.y_meshgrid,...
    %       phase_velocity.u_lin ,phase_velocity.v_lin);
    % omega_2c_near = derivation_2c(phase_velocity.
        x_meshgrid, phase_velocity.y_meshgrid,...
    %       phase_velocity.u_near ,phase_velocity.v_near);
    % omega_2c_nat = derivation_2c(phase_velocity.
        x_meshgrid, phase_velocity.y_meshgrid,...
    %       phase_velocity.u_nat ,phase_velocity.v_nat);

    % Numerical Derivation with Finite-difference 4th order
        , central
```

```
    omega_4c_lin = derivation_4c(phase_velocity(k).
       x_meshgrid, phase_velocity(k).y_meshgrid,...
        phase_velocity(k).u_lin ,phase_velocity(k).v_lin);
    omega_4c_near = derivation_4c(phase_velocity(k).
       x_meshgrid, phase_velocity(k).y_meshgrid,...
        phase_velocity(k).u_near ,phase_velocity(k).v_near)
            ;
    omega_4c_nat = derivation_4c(phase_velocity(k).
       x_meshgrid, phase_velocity(k).y_meshgrid,...
        phase_velocity(k).u_nat ,phase_velocity(k).v_nat);

    % Save results
    phase_vorticity(1,k) = struct('phase_range',
       phase_velocity(k).phase_range,...
          'x_meshgrid', phase_velocity(k).x_meshgrid,...
          'y_meshgrid', phase_velocity(k).y_meshgrid,...
          'vorticity_2D_lin',omega_4c_lin, '
             vorticity_2D_near', omega_4c_near,...
          'vorticity_2D_nat',omega_4c_nat);
end

%% Visualisation of results
% Check existence of folder figures. If not present, create
    one
if save_figure
    % Define location to save figures
    if exist('rootfigures.txt','file')
        import = importdata('rootfigures.txt');
        rootfigures = cell2mat(import(1,1));
    else
        % UI for selecting the data folders
        disp('Please select the root folder for figures.')
        rootfigures = uigetdir('','Please select the root 
           folder for figures.');

        % Save path to txt-file
        fileID = fopen('rootfigures.txt','wt');
        fprintf(fileID, '%s\n', rootfigures);
        fclose(fileID);
    end
```

```
latest = 1;
if exist(rootfigures,'dir')~=7
    mkdir(rootfigures);
end
cd(rootfigures);
list = dir('phase_vorticity-*');
list = struct2cell(list);
list = list(1,:);              % Crop list to filenames
list = char(list);
A = NaN(size(list,1),1);
for k = 1:size(list,1)
    temp = strsplit(list(k,:), {'-', '.'},'
        CollapseDelimiters',true);
    A(k) = str2num(cell2mat(temp(2)));
end
latest = max(A) + 1;
if isempty(latest)
    warning('Existing plots-folder in figures could not
        be found.')
    latest = 1;
end
currentfolder = sprintf('%s-%02d','phase_vorticity',
   latest);
mkdir(currentfolder);
disp(['Figures will be saved in folder number ',
   num2str(latest), '.']);

% Function handels
printeps = @(fname) print('-depsc2',... % print eps
   file
     fullfile(rootfigures,currentfolder,fname),'-zbuffer
        ','-r200');
printpng = @(fname) print('-dpng',... % print png file
     fullfile(rootfigures,currentfolder,fname),'-zbuffer
        ','-r200');
savefig = @(fname) hgsave(gcf,...          % Save figure
     fullfile(rootfigures,currentfolder,fname));

set(0,...                                  % figure
   settings
```

```
        'DefaultFigureColormap',gray,...
        'DefaultAxesVisible','on',...
        'DefaultAxesNextPlot','add',...
        'DefaultAxesFontSize',10);

% 2D velocities with linear interpolation
switch interp_method
    case 1
        titlestart = 'vorticity␣in␣s^{-1}␣in␣2D␣plane␣(
           Linear␣Interpolation)␣for␣';
        namestart = 'phase_vort_lin';
        vorticity = NaN(size(phase_vorticity(1).
           x_meshgrid,1),...
            size(phase_vorticity(1).x_meshgrid,2),
               Nphase);
        for m=1:Nphase
            if ~isempty(phase_vorticity(m).
               vorticity_2D_lin)
                vorticity(:,:,m) = phase_vorticity(m).
                   vorticity_2D_lin;
            else
                vorticity(:,:,m) = [];
            end
        end
    case 2
        titlestart = 'vorticity␣in␣s^{-1}␣in␣2D␣plane␣(
           Nearest␣N.)␣for␣';
        namestart = 'phase_vort_near';
        vorticity = NaN(size(phase_vorticity(1).
           x_meshgrid,1),...
            size(phase_vorticity(1).x_meshgrid,2),
               Nphase);
        for m=1:Nphase
            if ~isempty(phase_vorticity(m).
               vorticity_2D_near)
                vorticity(:,:,m) = phase_vorticity(m).
                   vorticity_2D_near;
            else
                vorticity(:,:,m) = [];
            end
```

```
            end
        case 3
            titlestart = 'vorticity in s^{-1} in 2D plane (
                Natural N.) for ';
            namestart = 'phase_vort_nat';
            vorticity = NaN(size(phase_vorticity(1).
                x_meshgrid,1) ,...
                size(phase_vorticity(1).x_meshgrid,2),
                    Nphase);
            for m=1:Nphase
                if ~isempty(phase_vorticity(m).
                    vorticity_2D_nat)
                    vorticity(:,:,m) = phase_vorticity(m).
                        vorticity_2D_nat;
                else
                    vorticity(:,:,m) = [];
                end
            end
    end
    for l=1:Nphase
        if ~isempty(vorticity(:,:,l))
        hfigure(gcf) = figure(gcf+1);
        contourf(phase_vorticity(l).x_meshgrid,...
            phase_vorticity(l).y_meshgrid,vorticity(:,:,l))
        title([titlestart,...
            num2str(phase_vorticity(l).phase_range(1),3), '
                to ',...
            num2str(phase_vorticity(l).phase_range(2),3), '
                radians.']);
        xlabel('horizontal distance to LIDAR in m')
        ylabel('vertical distance to LIDAR in m')
        if exist('limit','var')
            caxis([-abs(limit) abs(limit)])
        end
        colorbar
        colormap('jet')
        name = sprintf('%s%02d',namestart,l);
        printeps(name);
        printpng(name);
        savefig(name);
```

```
        else
            % If no data is present, create .txt file dummy
                to indicate it.
            name = sprintf('%s%02d%s',namestart,l,'.txt');
            fid = fopen(fullfile(rootfigures,currentfolder,
                name), 'w');
            fprintf(fid, '%s', 'No interpolation available'
                );
            fclose(fid);
        end
    end
end
close all
cd(scriptfolder)
end

function [ omega_2c ] = derivation_2c(x_mesh, y_mesh, u ,v)
% Set up matrix for numerical derivation
delta_x = x_mesh(1,2)- x_mesh(1,1);
delta_y = y_mesh(2,1)- y_mesh(1,1);

% Calculate derivaties du/dy and dv/dx
uy_2c = NaN(size(x_mesh));
vx_2c = NaN(size(x_mesh));

%Fill matrix with values
% Formula for algorithm: u' = (-u_(i-1) + u_(i+1))/(2*
    delta_X)
for k=2:size(u,1)-1          % Borders cannot be calculated
    for l=1:size(u,2)
        uy_2c(k,l) = (-u(k-1,l) + u(k+1,l))/(2*delta_y);
    end
end

for k=1:size(u,1)
    for l=2:size(u,2)-1      % Borders cannot be calculated
        vx_2c(k,l) = (-v(k,l-1) + v(k,l+1))/(2*delta_x);
    end
end
% Output
```

```
omega_2c = -uy_2c + vx_2c;
end

function [ omega_4c ] = derivation_4c(x_mesh, y_mesh, u ,v)
% Set up matrix for numerical derivation
delta_x = x_mesh(1,2)- x_mesh(1,1);
delta_y = y_mesh(2,1)- y_mesh(1,1);

% Calculate derivaties du/dy and dv/dx
uy_4c = NaN(size(x_mesh));
vx_4c = NaN(size(x_mesh));

%Fill matrix with values
% % Formula for algorithm:
% u' = (u_(i-2) - 8*u_(i-1)+8*u_(i+1)-u_(j+2))/(12*delta_X)
for k=3:size(u,1)-2          % Borders cannot be calculated
    for l=1:size(u,2)
        uy_4c(k,l) = (u(k-2,l) - 8*u(k-1,l) + 8*u(k+1,l)- u
           (k+2,l))/(12*delta_y);
    end
end

for k=1:size(u,1)
    for l=3:size(u,2)-2      % Borders cannot be calculated
        vx_4c(k,l) = (v(k,l-2) - 8*v(k,l-1) + 8*v(k,l+1)- v
           (k,l+2))/(12*delta_x);
    end
end

% Output
omega_4c = -uy_4c + vx_4c;
end
```

3.5 phase_shear2D

```
function [ phase_shear ] = phase_shear2D( phase_velocity,
   save_figure, shear_limit )
%% function shear2D
% function [ phase_shear ] = shear2D(phase_velocity)
```

```
% function [ phase_shear ] = shear2D(phase_velocity,
    save_figure)
% function [ phase_shear ] = shear2D(phase_velocity,
    save_figure, shear_limit)
%
% DESCRIPTION
% The function takes an input struct phase_velocity,
    computed
% by function phase_average2D, and calculates the wind
    shear with a finite
% difference method of 4th order and interpolated values (
    with linear,
% nearest neighbour and natural neighbour) and computes the
     wind shear.
%
% INPUT
% - phase_velocity: struct with data of the velocity
    components
%        phase_range: range of phases in rad
%        x_meshgrid: x-coordinates in meshgrid format
%        y_meshgrid: y-coordinates in meshgrid format
%        u_lin: horizontal wind component (linear)
%        v_lin: vertical wind component (linear)
%        velocity_2D_lin: absolute value of wind component (
    linear)
%        u_near: horizontal wind component (nearest
    neighbour)
%        v_near: vertical wind component (nearest neighbour)
%        velocity_2D_near: absolute value of wind component
    (nearest neighbour)
%        u_nat: horizontal wind component (natural neighbour
    )
%        v_nat: vertical wind component (natural neighbour)
%        velocity_2D_nat: absolute value of wind component (
    natural neighbour)
% - save_figure:     boolean. If TRUE, figures will be saved
    . Default is FALSE
% - shear_limit:     absolute value of maximal shear to be
    shown. Other values
```

```
% will be cropped and set by NaN in the final plots. If not
    set, no
% filtering will be applied.
%
% OUTPUT
% - phase_shear: struct with data of shear
%        phase_range: range of phases in rad
%        x_meshgrid:      x-coordinates in meshgrid format
%        y_meshgrid:      y-coordinates in meshgrid format
%        shear_2D_lin:    shear field, 4c, linear
   interpolation
%        shear_2D_lin_x: shear field, 4c, linear
   interpolation, x-derivative
%        shear_2D_lin_y: shear field, 4c, linear
   interpolation, y-derivative
%        shear_2D_near:  shear field, 4c, nearest neighbour
%        shear_2D_near_x:shear field, 4c, nearest neighbour,
    x-derivative
%        shear_2D_near_y:shear field, 4c, nearest neighbour,
    y-derivative
%        shear_2D_nat:   shear field, 4c, natural neighbour
%        shear_2D_nat_x: shear field, 4c, natural neighbour,
    x-derivative
%        shear_2D_nat_y: shear field, 4c, natural neighbour,
    y-derivative
%
% Code by: Markus Schmidt
%
% $Revision: 0.1$ $Date: 2013/05/14 $
%
% This code is licensed under a Creative Commons
   Attribution-ShareAlike
% 3.0 Unported License
% ( http://creativecommons.org/licenses/by-sa/3.0/deed.
   en_GB )

% Global settings
close all                                    % Close all open
   figures
```

```
% Global variables
interp_method = 1;            % 1: Linear, 2: Nearest N.,3:
    Natural N.

%% Input check
if nargin > 3
    error('Number of input arguments wrong.')
end

% Check and set variable 'save_figure'
if ~exist('save_figure','var')
    save_figure = false;
end

%% Compute the shear
% Compute number of discrete phases
Nphase = size(phase_velocity,2);

for k=1:Nphase
    % Numerical Derivation with Finite-difference 2nd order
        , central
    [shear_2c_lin_x, shear_2c_lin_y] = derivation_2c(
        phase_velocity(1,k).x_meshgrid,...
         phase_velocity(1,k).y_meshgrid,phase_velocity(1,k).
            u_lin  ,phase_velocity(1,k).v_lin);
    [shear_2c_near_x, shear_2c_near_y] = derivation_2c(
        phase_velocity(1,k).x_meshgrid,...
         phase_velocity(1,k).y_meshgrid, phase_velocity(1,k)
            .u_near  ,phase_velocity(1,k).v_near);
    [shear_2c_nat_x, shear_2c_nat_y] = derivation_2c(
        phase_velocity(1,k).x_meshgrid,...
         phase_velocity(1,k).y_meshgrid, phase_velocity(1,k)
            .u_nat  ,phase_velocity(1,k).v_nat);

    % Calculate magnitude
    shear_2c_lin_mag = sqrt(shear_2c_lin_x.^2 +
        shear_2c_lin_y.^2);
    shear_2c_near_mag = sqrt(shear_2c_near_x.^2 +
        shear_2c_near_y.^2);
```

```
shear_2c_nat_mag = sqrt(shear_2c_nat_x.^2 +
    shear_2c_nat_y.^2);

% Numerical Derivation with Finite-difference 4th order
    , central
[shear_4c_lin_x, shear_4c_lin_y] = derivation_4c(
    phase_velocity(1,k).x_meshgrid,...
      phase_velocity(1,k).y_meshgrid,phase_velocity(1,k).
          u_lin  ,phase_velocity(1,k).v_lin);
[shear_4c_near_x, shear_4c_near_y] = derivation_4c(
    phase_velocity(1,k).x_meshgrid,...
      phase_velocity(1,k).y_meshgrid, phase_velocity(1,k)
          .u_near  ,phase_velocity(1,k).v_near);
[shear_4c_nat_x, shear_4c_nat_y] = derivation_4c(
    phase_velocity(1,k).x_meshgrid,...
      phase_velocity(1,k).y_meshgrid, phase_velocity(1,k)
          .u_nat  ,phase_velocity(1,k).v_nat);

% Calculate magnitude
shear_4c_lin_mag = sqrt(shear_4c_lin_x.^2 +
    shear_4c_lin_y.^2);
shear_4c_near_mag = sqrt(shear_4c_near_x.^2 +
    shear_4c_near_y.^2);
shear_4c_nat_mag = sqrt(shear_4c_nat_x.^2 +
    shear_4c_nat_y.^2);

% Save results
phase_shear(1,k) = struct('phase_range', phase_velocity
    (k).phase_range,...
      'x_meshgrid', phase_velocity(k).x_meshgrid,...
      'y_meshgrid', phase_velocity(k).y_meshgrid,...
      'shear_2D_lin',shear_4c_lin_mag,...
      'shear_2D_lin_x',shear_4c_lin_x,'shear_2D_lin_y',
          shear_4c_lin_y,...
      'shear_2D_near',shear_4c_near_mag,...
      'shear_2D_near_x',shear_4c_near_x,'shear_2D_near_y'
          ,shear_4c_near_y,...
      'shear_2D_nat',shear_4c_nat_mag,...
      'shear_2D_nat_x',shear_4c_nat_x,'shear_2D_nat_y',
          shear_4c_nat_y);
```

```
end
%% Plot results
if save_figure
    % Define location to save figures
    if exist('rootfigures.txt','file')
        import = importdata('rootfigures.txt');
        rootfigures = cell2mat(import(1,1));
    else
        % UI for selecting the data folders
        disp('Please select the root folder for figures.')
        rootfigures = uigetdir('','Please select the root 
            folder for figures.');

        % Save path to txt-file
        fileID = fopen('rootfigures.txt','wt');
        fprintf(fileID, '%s\n', rootfigures);
        fclose(fileID);
    end
    scriptfolder = pwd;
    latest = 1;
    if exist(rootfigures,'dir')~=7
        mkdir(rootfigures);
    end
    cd(rootfigures);
    list = dir('phase_shear-*');
    list = struct2cell(list);
    list = list(1,:);            % Crop list to filenames
    list = char(list);
    A = NaN(size(list,1),1);
    for k = 1:size(list,1)
        temp = strsplit(list(k,:), {'-', '.'},'
            CollapseDelimiters',true);
        A(k) = str2num(cell2mat(temp(2)));
    end
    latest = max(A) + 1;
    if isempty(latest)
        warning('Existing plots-folder in figures could not
            be found.')
        latest = 1;
    end
```

```
currentfolder = sprintf('%s-%02d','phase_shear',latest)
    ;
mkdir(currentfolder);
disp(['Figures␣will␣be␣saved␣in␣folder␣number␣',
   num2str(latest), '.']);
cd(scriptfolder)

% Function handels
printeps = @(fname) print('-depsc2',... % print eps
    file
     fullfile(rootfigures,currentfolder,fname),'-zbuffer
        ','-r200');
printpng = @(fname) print('-dpng',... % print png file
     fullfile(rootfigures,currentfolder,fname),'-zbuffer
        ','-r200');
savefig = @(fname) hgsave(gcf,...          % Save figure
     fullfile(rootfigures,currentfolder,fname));

set(0,...                                  % figure
    settings
     'DefaultFigureColormap',gray,...
     'DefaultAxesVisible','on',...
     'DefaultAxesNextPlot','add',...
     'DefaultAxesFontSize',10);

% 2D velocities with linear interpolation
switch interp_method
     case 1
          titlestart = 'Wind␣Shear␣in␣s^{-1}␣in␣2D␣plane␣
             (Linear␣Interpolation)␣for␣';
          namestart = 'phase_shear_lin';
          shear = NaN(size(phase_shear(1).x_meshgrid,1)
              ,...
               size(phase_shear(1).x_meshgrid,2),Nphase);
          shear_x = NaN(size(phase_shear(1).x_meshgrid,1)
              ,...
               size(phase_shear(1).x_meshgrid,2),Nphase);
          shear_y = NaN(size(phase_shear(1).x_meshgrid,1)
              ,...
               size(phase_shear(1).x_meshgrid,2),Nphase);
```

```
        for m=1:Nphase
            if ~isempty(phase_shear(m).shear_2D_lin)
                shear(:,:,m) = phase_shear(m).
                    shear_2D_lin;
                shear_x(:,:,m) = phase_shear(m).
                    shear_2D_lin_x;
                shear_y(:,:,m) = phase_shear(m).
                    shear_2D_lin_y;
            else
                shear(:,:,m) = [];
                shear_x(:,:,m) = [];
                shear_y(:,:,m) = [];
            end
        end
    case 2
        titlestart = 'Wind Shear in s^{-1} in 2D plane
            (Nearest N.) for ';
        namestart = 'phase_shear_near';
        shear = NaN(size(phase_shear(1).x_meshgrid,1)
            ,...
            size(phase_shear(1).x_meshgrid,2),Nphase);
        shear_x = NaN(size(phase_shear(1).x_meshgrid,1)
            ,...
            size(phase_shear(1).x_meshgrid,2),Nphase);
        shear_y = NaN(size(phase_shear(1).x_meshgrid,1)
            ,...
            size(phase_shear(1).x_meshgrid,2),Nphase);
        for m=1:Nphase
            if ~isempty(phase_shear(m).shear_2D_near)
                shear(:,:,m) = phase_shear(m).
                    shear_2D_near;
                shear_x(:,:,m) = phase_shear(m).
                    shear_2D_near_x;
                shear_y(:,:,m) = phase_shear(m).
                    shear_2D_near_y;
            else
                shear(:,:,m) = [];
                shear_x(:,:,m) = [];
                shear_y(:,:,m) = [];
            end
```

```
            end
        case 3
            titlestart = 'Wind Shear in s^{-1} in 2D plane 
                (Natural N.) for ';
            namestart = 'phase_shear_nat';
            shear = NaN(size(phase_shear(1).x_meshgrid,1)
                ,...
                size(phase_shear(1).x_meshgrid,2),Nphase);
            shear_x = NaN(size(phase_shear(1).x_meshgrid,1)
                ,...
                size(phase_shear(1).x_meshgrid,2),Nphase);
            shear_y = NaN(size(phase_shear(1).x_meshgrid,1)
                ,...
                size(phase_shear(1).x_meshgrid,2),Nphase);
            for m=1:Nphase
                if ~isempty(phase_shear(m).shear_2D_nat)
                    shear(:,:,m) = phase_shear(m).
                        shear_2D_nat;
                    shear_x(:,:,m) = phase_shear(m).
                        shear_2D_nat_x;
                    shear_y(:,:,m) = phase_shear(m).
                        shear_2D_nat_y;
                else
                    shear(:,:,m) = [];
                    shear_x(:,:,m) = [];
                    shear_y(:,:,m) = [];
                end
            end
    end
    for l=1:Nphase
        if ~isempty(shear(:,:,l))
        hfigure(gcf) = figure(gcf+1);
        contourf(phase_shear(l).x_meshgrid,...
            phase_shear(l).y_meshgrid,shear(:,:,l))
        title([titlestart,...
            num2str(phase_shear(l).phase_range(1),3), ' to 
                ',...
            num2str(phase_shear(l).phase_range(2),3), '
                radians.']);
        xlabel('horizontal distance to LIDAR in m')
```

```
        ylabel('vertical distance to LIDAR in m')
        if exist('shear_limit','var')
            caxis([0 shear_limit])
        end
        colorbar
        colormap('jet')
        name = sprintf('%s%02d',namestart,l);
        printeps(name);
        printpng(name);
        savefig(name);
        else
            % If no data is present, create .txt file dummy
                to indicate it.
            name = sprintf('%s%02d%s',namestart,l,'.txt');
            fid = fopen(fullfile(rootfigures,currentfolder,
                name), 'w');
            fprintf(fid, '%s', 'No interpolation available'
                );
            fclose(fid);
        end
    end
end
close all
end

function [shear_x, shear_y] = derivation_2c(x_mesh, y_mesh
    , u ,v)
% Set up matrix for numerical derivation
delta_x = x_mesh(1,2)- x_mesh(1,1);
delta_y = y_mesh(2,1)- y_mesh(1,1);

% Grid check
if delta_x == 0
    error('Wrong x-mesh as input.')
end

if delta_y == 0
    error('Wrong y-mesh as input.')
end
```

```
% Calculate derivaties du/dx, du/dy, dv/dx and dv/dy
ux_2c = NaN(size(x_mesh));
uy_2c = NaN(size(x_mesh));
vx_2c = NaN(size(x_mesh));
vy_2c = NaN(size(x_mesh));

%Fill matrix with values
% Formula for algorithm: u' = (-u_(i-1) + u_(i+1))/(2*
    delta_X)

% For du/dx
for k=1:size(u,1)
    for l=2:size(u,2)-1      % Borders cannot be calculated
        ux_2c(k,l) = (-u(k,l-1) + u(k,l+1))/(2*delta_x);
    end
end

% For du/dy
for k=2:size(u,1)-1          % Borders cannot be calculated
    for l=1:size(u,2)
        uy_2c(k,l) = (-u(k-1,l) + u(k+1,l))/(2*delta_y);
    end
end

% For dv/dx
for k=1:size(u,1)
    for l=2:size(u,2)-1      % Borders cannot be calculated
        vx_2c(k,l) = (-v(k,l-1) + v(k,l+1))/(2*delta_x);
    end
end

% For dv/dy
for k=2:size(u,1)-1          % Borders cannot be calculated
    for l=1:size(u,2)
        vy_2c(k,l) = (-v(k-1,l) + v(k+1,l))/(2*delta_y);
    end
end

% Output
shear_x = ux_2c + vx_2c;
```

```
shear_y = uy_2c + vy_2c;
end

function [ shear_x, shear_y ] = derivation_4c(x_mesh,
   y_mesh, u ,v)
% Set up matrix for numerical derivation
delta_x = x_mesh(1,2)- x_mesh(1,1);
delta_y = y_mesh(2,1)- y_mesh(1,1);

% Grid check
if delta_x == 0
    error('Wrong x-mesh as input.')
end

if delta_y == 0
    error('Wrong y-mesh as input.')
end

% Calculate derivaties du/dx, du/dy, dv/dx and dv/dy
ux_4c = NaN(size(x_mesh));
uy_4c = NaN(size(x_mesh));
vx_4c = NaN(size(x_mesh));
vy_4c = NaN(size(x_mesh));

%Fill matrix with values
% % Formula for algorithm:
% u' = (u_(i-2) - 8*u_(i-1)+8*u_(i+1)-u_(j+2))/(12*delta_X)

% For du/dx
for k=1:size(u,1)
    for l=3:size(u,2)-2      % Borders cannot be calculated
        ux_4c(k,l) = (u(k,l-2) - 8*u(k,l-1) + 8*u(k,l+1)- u
           (k,l+2))/(12*delta_x);
    end
end

% For du/dy
for k=3:size(u,1)-2          % Borders cannot be calculated
    for l=1:size(u,2)
```

```
        uy_4c(k,l) = (u(k-2,l) - 8*u(k-1,l) + 8*u(k+1,l)- u
            (k+2,l))/(12*delta_y);
    end
end

% For dv/dx
for k=1:size(u,1)
    for l=3:size(u,2)-2     % Borders cannot be calculated
        vx_4c(k,l) = (v(k,l-2) - 8*v(k,l-1) + 8*v(k,l+1)- v
            (k,l+2))/(12*delta_x);
    end
end

% For dv/dy
for k=3:size(u,1)-2          % Borders cannot be calculated
    for l=1:size(u,2)
        vy_4c(k,l) = (v(k-2,l) - 8*v(k-1,l) + 8*v(k+1,l)- v
            (k+2,l))/(12*delta_y);
    end
end

% Output
shear_x = ux_4c + vx_4c;
shear_y = uy_4c + vy_4c;
end
```

3.6 drivescan2D

```
function [ ] = drivescan2D( start_time, end_time,
   lidar_home, log_home,...
    range, map_file, long, lat,scale, color_code,
        usegradient, limit)
%% function drivescan2D
% [ ] = drivescan2D( start_time, end_time, lidar_home,
   log_home,...
%     range, map_file, long, lat,scale, color_code)
% [ ] = drivescan2D( start_time, end_time, lidar_home,
   log_home,...
%     range, map_file, long, lat,scale, color_code,
   usegradient, limit)
```

```
%
% DESCRIPTION
% Function reads existing measurement files of LIDAR and
   GPS. It computes a
% scan image onto the map and saves the result in the
   folder specified via
% GUI or in root_superposer.txt.
%
% INPUT
% - start_time: numerical array of pattern [yyyy, m, d, h,
   min, sec]
% - end_time:   numerical array of pattern [yyyy, m, d, h,
   min, sec]
% - lidar_home: string containing the path to main folder
   with LIDAR data
% - log_home: string containing the location folder of log
   files.
% - range: radial distance between 2 measurements. Is used
   to resolve the
% range gate of the LIDAR system
% - map_file:   location of image map
% - lat:        latitude of reference point on map
% - lon:        longitude of reference point of map
% - scale:      scale as shown on map_file
% - color_code: location of .txt-file with color code
% - usegradient: boolean. If true, computation of gradient
   instead of
% velocity. False by default.
% - limit: limit of colorbar to be shown on plot
%
% OUTPUT
% - none
%
% Code by: Markus Schmidt
%
% $Revision: 2.1$ $Date: 2013/05/16 $
%
% This code is licensed under a Creative Commons
   Attribution-ShareAlike
% 3.0 Unported License
```

```
% ( http://creativecommons.org/licenses/by-sa/3.0/deed.
    en_GB )

% Input checks
if nargin < 10 || nargin == 11 || nargin > 12
    error('Wrong number of input arguments.')
end

% Import lidar files
% Scan and read all files within time. Returns matrix with
    values
[ lidar_data ] = scan_lidar(start_time, end_time,
    lidar_home, range);

% Import GPS data
[ loc_props ] = ML_software_loadv2(log_home, start_time,
    end_time );

% Interpolate the position for the time entries of the
    beams
[ test_series ] = UTM_interp(lidar_data, loc_props);

% Import of map
% map_calibration
[map, UTM_map_x, UTM_map_y, ~] = map_calibration_v2(
    map_file,lat,long,...
      scale);

% Compute the opacity on the map and create figures
if exist('limit', 'var')
     superposer_v2(test_series, map, UTM_map_x, UTM_map_y,
         color_code,...
           usegradient, limit);
else
     superposer_v2(test_series, map, UTM_map_x, UTM_map_y,
         color_code);
end

end
```