F

Python Notebooks

In this appendix we add some of the notebooks that we have created during the research[1]. We add three notebooks with functions that are used during the research. We also add eight notebooks that were used to obtain the results that are presented in this thesis. See below, for a description of the different notebooks.

Function notebooks

- Notebook 1.1: Basic functions
  Here some basic functions are defined. Such as the projector (see Eq. (2.7), rotation matrices (see Eq. (3.1)) and the best linear unbiased estimator (see Eq. (2.10)
- Notebook 1.2: Null line
  Here the functions are defined to compute the orientation of the null line for situations with only two LoS observations, see section 2.4.1
- Notebook 1.3: Functions strap down method
  In this notebook all functions related to the different Modules (as described in section 4.2) are presented.

Notebooks to generate results

- Notebook 2.1: Biased estimates
  In this notebook we show that neglecting the north component results in biased estimates for the east and up component. We also show the relation with the orientation of the null line as discussed in section 2.4.2.
- Notebook 2.2: Solution and variance-covariance matrix
  In this notebook we visualized the solution for situations where three LoS observations are available (as in section 2.3.3) and we visualized the variance-covariance matrix as discussed in section 2.3.4.
- Notebook 2.3: Orientation of the null line
  Notebook in which we visualize the orientation of the null line for situations where only two LoS observations are available, see section 2.3.3.
- Notebook 2.4: The stakeholder's perspective: Switzerland
  This notebook shows an example of the first perspective and it contains the code to estimate the MDDs in the transversal and normal direction for the case in Switzerland, see section 5.1.

[1] Note that we created more notebooks during the research, here we show the most important ones.

- Notebook 2.5: user of an existing InSAR product - Decomposition
  This notebook contains the code to estimate the transversal and normal displacements for the different RUMs in Veendam, see section 5.2. it is based on the second perspective of the user of an existing InSAR product.
- Notebook 2.6: Sentinel-1 viewing geometry
  This notebook shows how the viewing geometry of Sentinel-1 should be determined for a particular locatio on Earth.
- Notebook 2.7: Optimal viewing geometry
  Within this notebook the optimal viewing geometry of a new mission is estimated, which is related to the third perspective, see section 5.4.
- Notebook 2.8: Monte Carlo simulations
  This notebook contains the Monte Carlo simulations as discussed in section 3.4.

1.1 functions

Notebook with different basic functions used in all codes

In [ ]:

```
"""
File with different basic functions used in in the modules. This file contains functions for:
1. Rotation matrices
2. The projection vector
3. The projection matrix (matrix which projects the deformation in ENU or TLN to LoS)
4. The A matrix (functional model) when we want to estimate deformation rates from deformation TS
5. Qx_hat
6. BLUE


By Wietske Brouwer, Jan 2021

V2.1 - 14-04-2021

"""

get_ipython().run_line_magic('matplotlib', 'inline')
from math import *
import numpy as np
from matplotlib import pyplot as plt
import pandas as pd

def rotation_matrix(beta, gamma_l, gamma_t):
    """
    Function to calculate the rotation matrices

    Input:
        beta = azimuth of the longitudinal direction with respect to the North [radians]
        gamma_l = longitudinal slope [radians]
        gamma_t = cant/slope of the def. phenomena in the transversal direction  [radians]

    Output:
        R1, R2 and R3 are rotation matrices to transfer from a N-S, E-W and Up system to a strap-down coordinat

    """

    # from degrees to radians

    # Define the rotation matrices R1, R2 and R3
    R1 = np.matrix([[np.cos(beta), np.sin(beta), 0], [-np.sin(beta), np.cos(beta), 0],[ 0, 0, 1]])
    R2 = np.matrix([[1, 0, 0], [0, np.cos(gamma_l), -np.sin(gamma_l)], [0, np.sin(gamma_l), np.cos(gamma_l)]])
    R3 = np.matrix([[np.cos(gamma_t), 0, np.sin(gamma_t)],[0, 1, 0], [-np.sin(gamma_t), 0, np.cos(gamma_t)]])

    return R1, R2, R3


def projection_vector(inc, alpha_d):
    """
    Input:
        inc = incidence angle of the LoS vector [radians]
        alpha_d = azimuth of the zero-doppler plane [radians]

    Output:
        The projection vector of the 3D displacement (in East, North, Up) to the radar LoS

    """

    # Define the projection matrix
    p = (np.array([np.sin(inc)*np.sin(alpha_d), np.sin(inc)*np.cos(alpha_d), np.cos(inc)])).T
    return p


def projection_matrix(inc, alpha_d, beta, gamma_l, gamma_t):
    """
    Input:
        inc = array of incidence angles of different satellite geometries [radians]
        alpha_d = azimuth of the zero-doppler plane [radians]
        beta = azimuth of the longitudinal direction with respect to the North [radians]
        gamma_l = longitudinal slope [radians]
        gamma_t = cant/slope of the def. phenomena in the transversal direction  [radians]

    Output:
        The projection matrix A which projects the displacements in Transversal, Longitudinal and Normal direct
        LoS direction. A is a mx3 matrix where m is the number of LoS observations

    """
    # calulate rotation matrices
    R1, R2, R3 = rotation_matrix(beta, gamma_l, gamma_t)


    # the projection vector
    p = (np.array([np.sin(inc)*np.sin(alpha_d), np.sin(inc)*np.cos(alpha_d), np.cos(inc)])).T

    #compute projection matrix A
    A = p@R1@R2@R3

    return A

def projection_matrix_TS_rate(inc, alpha_d, beta, gamma_l, gamma_t, time):
    """
    Set up the projection matrix when you want to do the decomposition for cases where the input consists of In
    And the output should consisti op deformtion rates.

    This function computes the projection matrix.

    Input:
        inc = array of incidence angles of different satellite geometries [radians]
        alpha_d = azimuth of the zero-doppler plane [radians]
        beta = azimuth of the longitudinal direction with respect to the North [radians]
        gamma_l = longitudinal slope [radians]
        gamma_t = cant/slope of the def. phenomena in the transversal direction  [radians]
        time = vector with epochs for the LoS given as input

    Output:
        The projection matrix A which projects the displacements in Transversal, Longitudinal and Normal direct
        LoS direction. This A matrx should be used as not only the decomposition is calcualted but also the def
        when complete TS are given as input

    """


    A_proj = projection_matrix(inc, alpha_d, beta, gamma_l, gamma_t)
    A = np.matlib.repmat(A_proj, 1, 2)

    A[:,0] = np.array([np.squeeze(np.asarray(A[:,0]))*time]).T
    A[:,1] = np.array([np.squeeze(np.asarray(A[:,1]))*time]).T
    A[:,2] = np.array([np.squeeze(np.asarray(A[:,2]))*time]).T


    return A


def Q_xhat(A, Qyy):
    """
    Function to calculate the variance-covariance matrix for the estimates

    Input:
        A = A matrix (often the projection matrix of the 3D displacements to the LoS)
        Qyy = Varaiance covariance matrix of the observations

    Output:
        The projection matrix A which projects the displacements in Transversal, Longitudinal and Normal direct
        LoS direction. This A matrx should be used as not only the decomposition is calcualted but also the def
        when complete TS are given as input

    """

    Qx_hat = np.linalg.inv(A.T@np.linalg.inv(Qyy)@A)

    return Qx_hat


def BLUE(A, y, Qyy):
    """
    Function to calculate the Best Linear Unbiased Estimator

    Input:
        A = A matrix (often the projection matrix of the 3D displacements to the LoS) mxn
        y = vector with obervations mx1
        Qyy = Varaiance covariance matrix of the observations (mxm)

    Output:
        x_hat = vector with the estimates (nx1)
        Qx_hat = variance-covariance matrix of the unknown parameters (nxn)

    """


    x_hat = np.linalg.inv(A.T@np.linalg.inv(Qyy)@A)@A.T@np.linalg.inv(Qyy)@y
    Qx_hat = np.linalg.inv(A.T@np.linalg.inv(Qyy)@A)
    return x_hat, Qx_hat
```

1.2 null_line

Notebook with functions to compute the orientation of the null line when only two LoS observations are available

In [ ]:
```
"""
Functions to parameterize the null line for the case where only observations from two viewing geometries are av
The functions also visualize the null line.

"""
get_ipython().run_line_magic('matplotlib', 'inline')
%matplotlib notebook

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib.colors
from scipy.linalg import null_space
from math import *


from functions_strap_down_method_nodrama import *
from functions import *

from scipy.linalg import null_space
from mpl_toolkits.mplot3d import Axes3d
from mpl_toolkits.mplot3d import Axes3D



def phi_zeta(inc, alpha_d):
    """
    Compute phi and zeta when the viewing geometry of the two satellites is known

    Based on the cross product between the two normal vectors

    The cross product between the two normal LoS vectors determines the direction of the line.
    From the direction, we can determine phi and zeta

    """

    # The cross product between the two LoS vectors
    dir_null_space_e = np.sin(inc[0])*np.cos(alpha_d[0])*np.cos(inc[1]) - np.cos(inc[0])*np.sin(inc[1])*np.cos(
    dir_null_space_n = -np.sin(inc[0])*np.sin(alpha_d[0])*np.cos(inc[1]) + np.cos(inc[0])*np.sin(inc[1])*np.sin
    dir_null_space_u = np.sin(inc[0])*np.sin(alpha_d[0])*np.sin(inc[1])*np.cos(alpha_d[1]) - np.sin(inc[0])*np.

    # Compute ground projection of the line
    ground_proj = np.sqrt(dir_null_space_e**2 + dir_null_space_n**2)

    # Compute phi (angle between NS line)
    phi = np.rad2deg(np.arctan(dir_null_space_e / dir_null_space_n))

    # Compute zeta (angle between NE plane and the line)
    zeta = np.rad2deg(np.arctan(dir_null_space_u / ground_proj))

    return phi, zeta


def vector2plane(normal_vec, point, x, y):
    """
    function to compute the value for d in the formula

    Compute the surfaces perpendicular to the normal vector (normal_vec)
    When we have a vector [a,b,c] then we have the plane equation: a*x+b*y+c*z+d=0
    We need to compute d --> when we know a point on the plane we can compute d.
    The point at the plane is given by the end point of the unit LoS vector [a,b,c]

    When x and y are the limits for the planes this functions computes the values for z
    """

    # d can be computed with the dot product of the point and the normal vector
    d = -point.dot(normal_vec)

    # Create the meshgrid for x and y
    xx, yy = np.meshgrid(x,y)

    # Calculate corresponding z
    z = (-normal_vec[0] * xx - normal_vec[1] * yy - d) * 1. /normal_vec[2]

    return d, z, xx, yy


def plane_intersect(a, b):
    """
    Compute the intersection line of the equations from two lines

    a, b   4-tuples/lists
           Ax + By +Cz + D = 0
           A,B,C,D in order

    output: 2 points on line of intersection, np.arrays, shape (3,)
    """
    a_vec, b_vec = np.array(a[:3]), np.array(b[:3])

    aXb_vec = np.cross(a_vec, b_vec)

    A = np.array([a_vec, b_vec, aXb_vec])
    d = np.array([-a[3], -b[3], 0.]).reshape(3,1)

# could add np.linalg.det(A) == 0 test to prevent linalg.solve throwing error

    p_inter = np.linalg.solve(A, d).T

    return p_inter[0], (p_inter + aXb_vec)[0]


a, b = (1, -1, 0, 2), (-1, -1, 1, 3)
plane_intersect(a, b)



def null_space_2sat(inc, alpha_d, beta, gamma_t, gamma_l, plot):

    """
    Based on the viewing geometry from the two satellites, this functions computes the orientation of the null
    It also plots the null spaces of the two observations and the null line

    Input:
        inc = array with inc angles of satellites [radians]
        alpha_d = array with azimuth of the ZDF of the satellites [radians]
        frame orientation: [radians]
        plot: 0 = no plot, 1 = make the figure

    Output:
        Orientation of the null linne, Phi and Zeta [radians] and psi [radians]
    """


    ######## Compute unit LoS vectors ############
    A = projection_matrix(inc, alpha_d, beta, gamma_l, gamma_t)


    r = 1
    t_los = r*A[:,0]
    l_los = r*A[:,1]
    n_los = r*A[:,2]

    # Compute the unit LoS vectors
    unit_los_tln = np.zeros((3,1))

    for i in range(len(inc)):
        temp = np.matrix([[t_los[i,0]], [l_los[i,0]], [n_los[i,0]]])
        unit_los_tln = np.hstack((unit_los_tln, temp))

    unit_los_tln = np.delete(unit_los_tln, 0, 1)
    print (unit_los_tln)

    ############ Compute null space surfaces for both the satellites #############
    # Compute the surfaces perpendicular to the unit LoS vectors
    # When we have a vector [a,b,c] then we have the plane equation: a*x+b*y+c*z+d=0
    # We need to compute d --> when we know a point on the plane we can compute d.
    # The point at the plane is given by the end point of the unit LoS vector [a,b,c]

    x, y = np.linspace(-1,1,100), np.linspace(-1,1,100)

    # define normal vector and point for the plane corresponding to the asc obs.
    point_asc  = np.array([unit_los_tln[0,0], unit_los_tln[1,0], unit_los_tln[2,0]])
    normal_asc = np.array([unit_los_tln[0,0], unit_los_tln[1,0], unit_los_tln[2,0]])

    # define normal vector and point for the plane corresponding to the asc obs.
    point_dsc  = np.array([unit_los_tln[0,1], unit_los_tln[1,1], unit_los_tln[2,1]])
    normal_dsc = np.array([unit_los_tln[0,1], unit_los_tln[1,1], unit_los_tln[2,1]])

    # Compute the z coordinates for the two planes and the d values
    d_asc, z_asc, xx, yy = vector2plane(normal_asc, point_asc, x, y)
    d_dsc, z_dsc, xx, yy = vector2plane(normal_dsc, point_dsc, x, y)


    ############# Compute equation for the intersection line of the two planes ############
    # Compute the equation for the intersection line of the two planes
    # Based on the functions of the two planes we can compute the intersection line.

    # Define coefficients of the two null spaces (for asc en dsc)
    coef_asc = (normal_asc[0], normal_asc[1], normal_asc[2], d_asc)
    coef_dsc= (normal_dsc[0], normal_dsc[1], normal_dsc[2], d_dsc)

    # Compute the coordinates for two points at the intersection line
    pnt_one, pnt_two = plane_intersect(coef_asc, coef_dsc)

    # Try the vector equation for the two points (such that the line becomes longer)
    t = np.linspace(-1,1,100)
    dir_vec = pnt_two-pnt_one

    x_line = np.zeros(100)
    y_line = np.zeros(100)
    z_line = np.zeros(100)


    for i in range(len(t)):
        line_coor = pnt_one + t[i]*dir_vec
        x_line[i] = line_coor[0]
        y_line[i] = line_coor[1]
        z_line[i] = line_coor[2]


    ############ Compute angles for the intersection line ################

    # Compute differences between the two points
    e_diff = -1*(pnt_one[0] - pnt_two[0])
    n_diff = -1*(pnt_one[1] - pnt_two[1])
    u_diff = -1*(pnt_one[2] - pnt_two[2])

    # Compute ground projection of the line
    ground_proj = np.sqrt(e_diff**2 + n_diff**2)

    # Compute phi (angle between NS line)
    phi = np.rad2deg(np.arctan(e_diff / n_diff))

    # Compute zeta (angle between NE plane and the line)
    zeta = np.rad2deg(np.arctan(u_diff / ground_proj))

    z_ground = np.zeros(100)

    psi = np.rad2deg(np.arctan(u_diff / e_diff))


    if plot == 1:
        plt3d = plt.figure(figsize=(8,8)).gca(projection='3d')
        # plot the two surfaces
        plt3d.plot_surface(xx, yy, z_asc, alpha=0.5, rstride=100, cstride=100, color = 'b')
        plt3d.plot_surface(xx, yy, z_dsc, alpha=0.5, rstride=100, cstride=100, color = 'green')

        # Plot the unit LoS vectors
        plt3d.plot([0,unit_los_tln[0,0]],[0,unit_los_tln[1,0]], [0,unit_los_tln[2,0]], label='LoS asc', linewid
        plt3d.plot([0,unit_los_tln[0,1]],[0,unit_los_tln[1,1]], [0,unit_los_tln[2,1]], label='LoS desc', linewi

        # Plot the 'projections' of the LoS vectors
        plt3d.plot([unit_los_tln[0,0],unit_los_tln[0,0]],[unit_los_tln[1,0],unit_los_tln[1,0]], [0,unit_los_tln
                   linewidth=2, color = 'b', ls = '--', alpha = 0.6)
        plt3d.plot([0,unit_los_tln[0,0]],[0,unit_los_tln[1,0]], [0,0], linewidth=2, color = 'b',
                   ls = '--', alpha = 0.6)

        plt3d.plot([unit_los_tln[0,1],unit_los_tln[0,1]],[unit_los_tln[1,1],unit_los_tln[1,1]], [0,unit_los_tln
                   linewidth=2, color = 'g', ls = '--', alpha = 0.6)
        plt3d.plot([0,unit_los_tln[0,1]],[0,unit_los_tln[1,1]], [0,0], linewidth=2, color = 'g',
                   ls = '--', alpha = 0.6)



        # Plot the intersection line
        plt3d.plot(x_line, y_line, z_line, lw=4, c='r', label = 'Null space')
        plt3d.plot(x_line, y_line, z_ground, lw=2, c='r', ls = '--', alpha = 0.7)


        plt.legend(fontsize = 12)
        plt.ylabel('North', fontsize = 12)
        plt.xlabel('East', fontsize = 12)
#         plt.title(r'$\phi$ = '+ str(np.around(phi, 2)) + ', $\zeta$ = ' + str(np.around(zeta, 2))
#                   + ', $\psi$ = ' + str(np.around(psi, 2)), fontsize = 20)

        plt.show()


    return zeta, phi, psi
```

1.3 functions_strap_down_method

File with the different functions used in the Modules as descrbied in Chapter 4 of the thesis.

In [ ]:

```python
get_ipython().run_line_magic('matplotlib', 'inline')
import numpy as np
import matplotlib.pyplot as plt
from math import *

from drama.performance.sar import SARModeFromCfg
from drama.io import cfg
from drama.mission.timeline import LatLonTimeline

import scipy.stats as st
from scipy.stats import norm
from scipy import stats
from scipy.stats.distributions import chi2

"""
Functions for the strap down method that can run under the assumption that the longitudinal displacements are z

Version 2.1

Date: 07-05-2021

Author: Wietske Brouwer


"""

def BLUE(A, y, Qyy):
    """
    Function to calculate the Best Linear Unbiased Estimator

    Input:
        A = A matrix (often the projection matrix of the 3D displacements to the LoS) mxn
        y = vector with obervations mx1
        Qyy = Varaiance covariance matrix of the observations (mxm)

    Output:
        x_hat = vector with the estimates (nx1)
        Qx_hat = variance-covariance matrix of the unknown parameters (nxn)

    """


    x_hat = np.linalg.inv(A.T@np.linalg.inv(Qyy)@A)@A.T@np.linalg.inv(Qyy)@y
    Qx_hat = np.linalg.inv(A.T@np.linalg.inv(Qyy)@A)
    return x_hat, Qx_hat

##########################################################################################
# MODULE 1: LoS Geometry
##########################################################################################

def viewing_geometry(lat, lon, mission, orbit_resolution, par_file, mode):
    """
    Input:
        1. Lat: latitudinal coordinate(s). i) can be a grid or ii) one value
        2. lon: longitudinal coordinate(s). i) can be a grid or ii) one value
        3. Satellite mission: name of the missen e.g. 'sentinel' or 'terrasar' etc...
        4. Orbit resolution

    Output:
        1. nr_asc obs
        2. asc_inc [radians]
        3. asc_zero_doppler [radians]
        4. nr_dsc_obs
        5. dsc_inc [radians]
        6. dsc_zero_doppler [radians]

    """
    nr_asc_obs = 0
    asc_inc = 0
    asc_zero_doppler = 0
    nr_dsc_obs = 0
    dsc_inc = 0
    dsc_zero_doppler = 0


#     if mission == 'sentinel':
#         par_file = Path(r"C:\Users\wiets\Python\Afstuderen\Python - Discplacement vector decomposition\drama-
#         mode = SARModeFromCfg(cfg.ConfigFile(par_file), "IWS")

#     else:
#         print ('The mission that is asked does not exists in the library')


    #For the case where our location consists of only 1 coordinate (so not a range)
    if np.size(lat) ==1 & np.size(lon)==1:
        # Compute acquisitionn geometry with DRaMA for the given location
        timeline = LatLonTimeline(par_file, lat, lon,
                                  inc_angle_range=(mode.incs[0, 0], mode.incs[-1, 1]),
                                  dlat=orbit_resolution, dlon=orbit_resolution)

        nr_asc_obs = len(timeline.asc_acqs[0].theta_i)
        asc_inc = timeline.asc_acqs[0].theta_i
        asc_zero_doppler = timeline.asc_acqs[0].northing+2*np.pi

        nr_dsc_obs = len(timeline.desc_acqs[0].theta_i)
        dsc_inc = timeline.desc_acqs[0].theta_i
        dsc_zero_doppler = timeline.desc_acqs[0].northing

    # For the case where we have a list of coordinates
    if (np.size(np.shape(lat))) ==1:
        timeline = LatLonTimeline(par_file, np.ravel(lat), np.ravel(lon),
                                  inc_angle_range=(mode.incs[0, 0], mode.incs[-1, 1]),
                                  dlat=orbit_resolution, dlon=orbit_resolution)

        nr_asc_obs = np.array([len(acq.theta_i) for acq in timeline.asc_acqs]).reshape(lon.shape)
        asc_inc = np.zeros((len(nr_asc_obs), nr_asc_obs[0]))
        asc_zero_doppler = np.zeros((len(nr_asc_obs), nr_asc_obs[0]))

        nr_dsc_obs = np.array([len(acq.theta_i) for acq in timeline.desc_acqs]).reshape(lon.shape)
        dsc_inc = np.zeros((len(nr_dsc_obs), nr_dsc_obs[0]))
        dsc_zero_doppler = np.zeros((len(nr_dsc_obs), nr_dsc_obs[0]))

        for i in range (len(nr_asc_obs)):
            asc_inc[i,:] = timeline.asc_acqs[i].theta_i
            asc_zero_doppler[i,:] = timeline.asc_acqs[i].northing+2*np.pi

        for i in range (len(nr_dsc_obs)):
            dsc_inc[i,:] = timeline.desc_acqs[i].theta_i
            dsc_zero_doppler[i,:] = timeline.desc_acqs[i].northing


    #For the case where we have a grid of coordinates
    if (np.size(np.shape(lat))) ==2:
        timeline = LatLonTimeline(par_file, np.ravel(lat), np.ravel(lon),
                                  inc_angle_range=(mode.incs[0, 0], mode.incs[-1, 1]),
                                  dlat=orbit_resolution, dlon=orbit_resolution)

        # Bepaal hoevaak elke locatie in het grid bekeken wordt door asc en dsc
        nr_asc_obs = np.array([len(acq.theta_i) for acq in timeline.asc_acqs]).reshape(lon.shape)
        nr_dsc_obs = np.array([len(acq.theta_i) for acq in timeline.desc_acqs]).reshape(lon.shape)


        # Maak zero arrays aan om te gaan vullen met de juiste inc en heading angles
        asc_inc = np.zeros((np.shape(lat)[0], np.shape(lat)[1], nr_asc_obs[0,0]))
        asc_zero_doppler = np.zeros((np.shape(lat)[0], np.shape(lat)[1], nr_asc_obs[0,0]))

        dsc_inc = np.zeros((np.shape(lat)[0], np.shape(lat)[1], nr_dsc_obs[0,0]))
        dsc_zero_doppler = np.zeros((np.shape(lat)[0], np.shape(lat)[1], nr_dsc_obs[0,0]))

        # Fill zero arrays with correct values for the inc and heading angles per location
        for i in range(nr_asc_obs[0,0]):
            asc_inc[:,:,i] = np.array([acq.theta_i[i] for acq in timeline.asc_acqs]).reshape(lon.shape)
            asc_zero_doppler[:,:,i] = np.array([acq.northing[i]+2*np.pi for acq in timeline.asc_acqs]).reshape(

        for i in range(nr_dsc_obs[0,0]):
            dsc_inc[:,:,i] = np.array([acq.theta_i[i] for acq in timeline.desc_acqs]).reshape(lon.shape)
            dsc_zero_doppler[:,:,i] = np.array([acq.northing[i] for acq in timeline.desc_acqs]).reshape(lon.sha



    return (timeline, nr_asc_obs, asc_inc, asc_zero_doppler, nr_dsc_obs, dsc_inc, dsc_zero_doppler)

##########################################################################################
# MODULE 2: Parameterization of the Deformation phenomenon
##########################################################################################

def parameterization(DP):
    """
    A function to determine which reference system should be used, the ENU or TLN?
    Also the unknown parameters are defined.

    For line-infrastructure, correct for different possible values of beta if gamma_t equals 0

    input:
        DP = i.2 Deformation phenomenon

    Output:
        TLN = 1 or 0. 1 means we should use the TLN reference system with the assumption of no longitudinal def
        ENU = 1 or 0. 1 means we should use the ENU reference system
        unknowns = [x1, x2, x3] with a value of 0 of 1. A value of 1 means that a displacement in that directio
        zero value means no displacement
        beta, gamma_t, gamma_l: the correct angles. When the ENU system is used, all angles are zero. For flat
                                angle of beta needs to be correct to the smallest value.

    """
    beta = DP[1]
    gamma_t = DP[2]
    gamma_l = DP[3]

    TLN = 0
    ENU = 0
    unknowns = 0

    # For the landslide case the value of gamma_t needs to be greater than 10 degrees to define a preferred di
    if DP[0] == 'slope instability':
        if gamma_t > 10: # The tresholdvalue for the landslide deformation phenomena
            TLN = 1
            unknowns = np.array([1,0,1])
        else:
            beta =0
            gamma_l = 0
            gamma_t = 0
            ENU = 1
            unknowns = np.array([1,1,1])
            print ('else')

    if DP[0] == 'subsidence' or DP[0] =='uplift':
        TLN = 1
        unknowns = np.array([1,0,1])

    # For line-infrastructure we always have a preferred direction.
    # When gamma_t = 0 there is a directional ambiguity, the smallest azimuth angle should be chosen.
    # The value for beta needs to be corrected if not the smallest beta value is chosen
    if DP[0] == 'line-infrastructure':
        TLN = 1
        unknowns = np.array([1,0,1])
        if gamma_t == 0:
            if beta > 270:
                beta = beta - 360 # Correct for smallest beta when we have directional ambiguity

            elif beta > 90 & beta <= 270:
                beta = beta - 180 # Correct for smallest beta when we have directional ambiguity

    if DP[0] == 'other': # For cases where we can specify a TLN reference system with certain deformation restr
                         # do not fall between the four specified def. phenomena
        unknowns = DP[4] #TLN_other are the possible deformation directions

    if DP[0] == 'ENU':
        beta =0
        gamma_l = 0
        gamma_t = 0
        ENU = 1
        unknowns = np.array([1,1,1])

    # When the ENU referece system is used, the angles for beta, gamma_t and gamma_l are equal to zero
    if ENU ==1:
        beta =0
        gamma_l = 0
        gamma_t = 0
        ENU = 1
        unknowns = np.array([1,1,1])

    return beta, gamma_t, gamma_l, ENU, TLN, unknowns


##########################################################################################
# MODULE 3: Compute functional model and Qyy

# WARNING: Those functions are only working when the TLN frame can be used and we asume zero
# longitudinal displacements
##########################################################################################

def solvable(nr_asc_obs, nr_dsc_obs, unknowns):
    """
    A function to test whether there are enough observations to solve for the unknowns

    Input:
        1. nr_asc_obs, nr_dsc_ons, unknowns

    Output:
        1. m = observations
        2. n = unknowns
        3. s = solvable 1 = yes, 0 = no

    """

    s = 0

    n = np.sum(unknowns)
    m = nr_asc_obs+nr_dsc_obs

    if m < n:
        print ('There are not enough observations to solve for the unknowns')
    if m >= n:
        s = 1
        print ('There are m=',m, ' observations available to solve for n=',n, ' unknowns')

        if nr_asc_obs ==0 or nr_dsc_obs ==0:
            print ('Potentially the viewing geometry is too low, only observations of 1 geometry are available'

    return m, n, s

def TLN_forward_model(inc, alpha_d, beta, gamma_t, gamma_l, d_T, d_N):
    """
    Compute the forward model when using the TLN frame and the assumption that d_L = 0.
    Compute LoS observations and pseudo observations for the three orientation angles

    Input:
        1. inc: array with incidence angles from observations available over the RUM [radians] (has the same le
        2. alpha_d: array with zero-doppler azimuth angles from observations available over the RUM [radians]
        3. beta [radians]
        4. gamma_t [radians]
        5. gamma_l [radians]
        6. d_T: displacement in transversal direction
        7. d_N: displacement in normal direction

    Output:
        y: vector with LoS observations and pseudo observations for beta, gamma_t and gamma_l

    """

    nr_sat = len(inc)

    # make arrays for the angles that have the same length as the number of satellites that are used
    beta = np.ones(nr_sat)*beta
    gamma_t = np.ones(nr_sat)*gamma_t
    gamma_l = np.ones(nr_sat)*gamma_l

    # Make arrays of the displacement vector estimates that have te same length as the numer of satellites that
    d_T = np.ones(nr_sat)*d_T
    d_N = np.ones(nr_sat)*d_N

    # Create the y-hat vector, the vector consists of two components, the transversal part and the normal part
    y = np.zeros((nr_sat+3, 1))

    dt = ((np.sin(inc)*np.sin(alpha_d)*np.cos(beta) - np.sin(inc)*np.cos(alpha_d)*np.sin(beta))*np.cos(gamma_t)
    dn = ((np.sin(inc)*np.sin(alpha_d)*np.cos(beta) - np.sin(inc)*np.cos(alpha_d)*np.sin(beta))*np.sin(gamma_t)


    y1 = dt+dn

    for i in range(nr_sat):
        y[i] = y1[i]

    # Add estimate of beta to the vector
    y[-1] = gamma_l[0]
    y[-2] = gamma_t[0]
    y[-3] = beta[0]

    return y


def TLN_jacobian(inc, alpha_d, beta, gamma_t, gamma_l, d_T, d_N):
    """
    Fill in the Jacobian matrix for the TLN frame under the assumption that d_L = 0

    Input:
        1. inc: array with incidence angles from observations available over the RUM [radians]
        2. alpha_d: array with zero-doppler azimuth angles from observations available over the RUM [radians]
        3. beta [radians]
        4. gamma_t [radians]
        5. gamma_l [radians]
        6. d_T: displacement in transversal direction
        7. d_N: displacement in normal direction

    Output:
        y: vector with LoS observations and pseudo observations for beta, gamma_t and gamma_l

    """

    nr_sat = len(inc)

    # make arrays for the angles that have the same length as the number of satellites that are used
    beta = np.ones(nr_sat)*beta
    gamma_t = np.ones(nr_sat)*gamma_t
    gamma_l = np.ones(nr_sat)*gamma_l

    # Make arrays of the displacement vector estimates that have te same length as the numer of satellites that
    d_T = np.ones(nr_sat)*d_T
    d_N = np.ones(nr_sat)*d_N

    # Compute the values for the Jacobian matrix

    #j1 = d_T, j2 = d_N, j3=beta, j4=gamma_t, j5 = gamma_l

    j1 = (np.sin(inc)*np.sin(alpha_d)*np.cos(beta) - np.sin(inc)*np.cos(alpha_d)*np.sin(beta))*np.cos(gamma_t)
    j2 = (np.sin(inc)*np.sin(alpha_d)*np.cos(beta) - np.sin(inc)*np.cos(alpha_d)*np.sin(beta))*np.sin(gamma_t)
    j3 = ((-np.sin(inc)*np.sin(alpha_d)*np.sin(beta) - np.sin(inc)*np.cos(alpha_d)*np.cos(beta))*np.cos(gamma_t
    j4 = (-(np.sin(inc)*np.sin(alpha_d)*np.cos(beta) - np.sin(inc)*np.cos(alpha_d)*np.sin(beta))*np.sin(gamma_t
    j5 = -(-(np.sin(inc)*np.sin(alpha_d)*np.sin(beta) + np.sin(inc)*np.cos(alpha_d)*np.cos(beta))*np.cos(gamma_


    # Create the Jacobian matrix
    J = np.array([j1, j2, j3, j4, j5]).T

    # Add row with the derivatives to beta to the Jacobian matrix
    beta_row = np.matrix([0, 0, 1, 0, 0])
    gamma_t_row = np.matrix([0, 0, 0, 1, 0])
    gamma_l_row = np.matrix([0, 0, 0, 0, 1])


    jacobian = np.vstack((J,beta_row, gamma_t_row, gamma_l_row))

    return jacobian


def TLN_Qyy(std_los, std_beta, std_gamma_t, std_gamma_l):
    """
    A function to compute the Qyy matrix

    Input:
        1. std_los: array with standard deviations for the LoS obsverations
        2. std_beta, std_gamma_t, std_gamma_l: standard deviations for the orientation angles

    Output:
        1. Qyy matrix

    """

    Qyy = np.zeros((len(std_los)+3, len(std_los)+3))
    np.fill_diagonal(Qyy[0:len(std_los)], std_los**2)
    Qyy[-3,-3] = std_beta**2
    Qyy[-2,-2] = std_gamma_t**2
    Qyy[-1,-1] = std_gamma_l**2

    return Qyy


##########################################################################################
# MODULE 4: Hypothesis testing

# WARNING: Those functions are only working when the TLN frame can be used and we asume zero
# longitudinal displacements
##########################################################################################

def TLN_Qxhat(inc, alpha_d, beta, gamma_t, gamma_l, d_T, d_N, Qyy):
    """
    Function to determine the Qxhat matrix for a displacement signal
        Qx_hat is computed for the values of d_T and d_N that are used as input

    Input:
        1. inc: array with incidence angles from observations available over the RUM [radians]
        2. alpha_d: array with zero-doppler azimuth angles from observations available over the RUM [radians]
        3. beta [radians]
        4. gamma_t [radians]
        5. gamma_l [radians]
        6. d_T: displacement in transversal direction
        7. d_N: displacement in normal direction
        8. Qyy matrix

    Output:
        1. Qx_hat matrix

    """

    J = TLN_jacobian(inc, alpha_d, beta, gamma_t, gamma_l, d_T, d_N)
    Qx_hat = np.linalg.inv(J.T@np.linalg.inv(Qyy)@J)

    return Qx_hat



def compute_dp(alpha_2, sigma, mdd, plot):
    """
    Function to compute the Detectability power when the MDD, alpha and sigma are known.
    This functions uses the 'integral' method (so it uses the area under the Gaussian distribution )

    Input:
        1. alpha_2 = alpha/2 (significance level) e.g. 5% significance level means alpha_2 = 0.025
        2. sigma = standard deviation of H_0
        3. mdd: value for the MDD
        4. plot: 1 = create figure, 0 create no figure

    Output:
        The detectability power in percent, e.g. 80%

    """

    k = st.norm.ppf(1-alpha_2, loc=0, scale=sigma)
    dp = 1-st.norm.cdf(k, loc=mdd, scale=sigma )
    dp = dp*100

    if plot ==1:

        # The distribution of the null hypothesis
        x = np.linspace(st.norm.ppf(0.00001, loc = 0, scale = sigma), st.norm.ppf(0.99999, loc = 0, scale = sig
        # the shape/ location of the alternative hypothesis
        x_a= np.linspace(st.norm.ppf(0.00001, loc = mdd, scale = sigma), st.norm.ppf(0.99999, loc = mdd, scale

        plt.figure()
        plt.plot(x, st.norm.pdf(x, loc = 0, scale = sigma),'r-', lw=5, alpha=0.6, label='H0')
        plt.plot(x_a, st.norm.pdf(x_a, loc = mdd, scale = sigma),'g-', lw=5, alpha=0.6, label='Ha')
        plt.axvline(k, label='The critical value')
        plt.legend()

    return dp

def compute_mdd(alpha_2, sigma, dp, plot):

    """
    Function to compute the Detectability power when the MDD, alpha and sigma are known.
    This functions uses the 'integral' method (so it uses the area under the Gaussian distribution )


    Input:
        1. alpha_2 = alpha/2 (significance level) e.g. 5% significance level means alpha_2 = 0.025
        2. sigma = standard deviation of H_0
        3. dp: value for the detectability power in percentage (e.g. a typical value = 80%)
        4. plot: 1 = create figure, 0 create no figure

    Output:
        the MDD

    """
    dp_1 = dp/100
    x_temp = norm.ppf(1-dp_1, loc = 0, scale = sigma)
    K = norm.ppf(1-alpha_2, loc=0, scale = sigma)
    mdd = K - x_temp

    if plot ==1:

        # The distribution of the null hypothesis
        x = np.linspace(st.norm.ppf(0.00001, loc = 0, scale = sigma), st.norm.ppf(0.99999, loc = 0, scale = sig
        # the shape/ location of the alternative hypothesis
        x_a= np.linspace(st.norm.ppf(0.00001, loc = mdd, scale = sigma), st.norm.ppf(0.99999, loc = mdd, scale

        plt.figure()
        plt.plot(x, st.norm.pdf(x, loc = 0, scale = sigma),'r-', lw=5, alpha=0.6, label='H0')
        plt.plot(x_a, st.norm.pdf(x_a, loc = mdd, scale = sigma),'g-', lw=5, alpha=0.6, label='Ha')
        plt.axvline(K, label='The critical value')
        plt.legend()
    return mdd

def Qee(A,Qyy):

    """
    Compute the Qee matrix

    Input:
        1. A: Functional A matrix
        2. Qyy = variance covariance matrix

    Output:
        Qee
    """

    a = np.linalg.inv(A.T@np.linalg.inv(Qyy)@A)
    b = A@a@A.T
    Q_ee = Qyy - b

    return Q_ee


def MDD(L, cy, Qyy, A):

    """
    Function to compute the MDD with formula 131 from Testing Theory book Peter Teunissen

    Input:
        L = Lambda_0
        cy: cy vector of the alternative hypothesis
        Qyy: variance-covarinace matrix of the unknowns
        A: Functional A matrix

    Output:
        MDD: Minimal Detectable Displacement under the null hypothesis
    """

    Q_ee = Qee(A,Qyy)
    c = cy.T@np.linalg.inv(Qyy)@Q_ee@np.linalg.inv(Qyy)@cy
    MDD = np.sqrt(L/c)

    return MDD


def y_p(A, x_hat):

    """
    Function to compute the predicted y value (for the condition model)

    """

    y_p = A@x_hat

    return y_p


def Qt(A, Qxhat, std_ym, B):

    """
    Function to compute Qt (for the condition model)

    Input:
        1. Forward model prediction
        2. Qxhat: precision of the estimated parameter with which we predict y_p
        3. std_ym: the standard deviation of the measured observation
        4. B: The B matrix

    Output:
        1. Qt
    """

    var_yp = A@Qxhat@A.T
    #print ('var_yp:', var_yp)
    #print ('std', std_ym)

    Qyy = np.zeros((2,2))
    Qyy[0,0] = var_yp
    Qyy[1,1] = std_ym**2

    Qt = B.T@Qyy@B

    return Qt, var_yp

def MDD_condition(L, Qt, c):

    """
    Function to compute the MDD for the condition model

    Input:
        1. L = lambda_0
        2. Qt
        3. c vector (usually 0)

    Output:
        1. MDD computed with the condition equation
    """

    MDD = np.sqrt(L/(c*1/Qt * c))

    return MDD


##########################################################################################
# MODULE 6: Compute functional model and Qyy

# WARNING: Those functions are only working when the TLN frame can be used and we asume zero
# longitudinal displacements
##########################################################################################

def TLN_Qxhat(inc, alpha_d, beta, gamma_t, gamma_l, d_T, d_N, Qyy):
    """
    Function to determine the Qxhat matrix for a displacement signal

    Input:
        1. inc: array with incidence angles from observations available over the RUM [radians]
        2. alpha_d: array with zero-doppler azimuth angles from observations available over the RUM [radians]
        3. beta [radians]
        4. gamma_t [radians]
        5. gamma_l [radians]
        6. d_T: displacement in transversal direction
        7. d_N: displacement in normal direction
        8. Qyy matrix

    """

    J = TLN_jacobian(inc, alpha_d, beta, gamma_t, gamma_l, d_T, d_N)
    Qx_hat = np.linalg.inv(J.T@np.linalg.inv(Qyy)@J)

    return Qx_hat


def iteration_estimate_x(x0, y, Qyy, epsilon, inc, alpha_d):
    """
    A function to estimate x based on an initial guess and the linearized method

    Input:
        1. x0: initial guess vector: [dt, dn, beta, gamma_t, gamma_l]
        2. y: observation vector
        3. Qyy: variance covariance matrix
        4. Epsilon: stop criterium
        5. inc: array with incidence angles
        6. alpha_d: array with zero-doppler azimuth angles

    Output:
        1. estimates for unknonws x = [d_t, d_n, beta, gamma_t, gamma_l]^t
        2. x_trans: how the transversal estimate changes during the iterations
        3. x_normal: how the normal estimate changes during the iterations
        4. x_beta, x_gamma_t, x_gamma_l: how the angles estimates change during the iterations
        5. Qx_hat: the precisions of the unknowns for the last iteration step
    """

    # Initial guess
    x = x0

    # max number of iterations
    N = 1000
    x_trans = np.zeros(N)
    x_normal = np.zeros(N)
    x_beta = np.zeros(N)
    x_gamma_t = np.zeros(N)
    x_gamma_l = np.zeros(N)

    for i in range(N):

        xg = np.squeeze(np.asarray(x))

        x_trans[i] = xg[0]
        x_normal[i] = xg[1]
        x_beta[i] = xg[2]
        x_gamma_t[i] = xg[3]
        x_gamma_l[i] = xg[4]

        # Observations for initial guess
        y_x0 = TLN_forward_model(inc, alpha_d, xg[2], xg[3], xg[4], xg[0], xg[1])

        delta_y = y - y_x0

        # fill in Jacobian (with with the values for x)
        jacobian = TLN_jacobian(inc, alpha_d, xg[2], xg[3], xg[4], xg[0], xg[1])

        # BLUE for the Jacobian and delta_y
        d_xhat, Qx_hat = BLUE(jacobian, delta_y, Qyy)


        x = x+d_xhat

        if np.sqrt(d_xhat[0]**2+d_xhat[1]**2+d_xhat[2]**2+d_xhat[3]**2+d_xhat[4]**2)<=epsilon:
            #print ('The number of iterations was', i)
            break


    x_trans = x_trans[0:i+1]
    x_normal = x_normal[0:i+1]
    x_beta = x_beta[0:i+1]
    x_gamma_t = x_gamma_t[0:i+1]
    x_gamma_l = x_gamma_l[0:i+1]

    return x, x_trans, x_normal, x_beta, x_gamma_t, x_gamma_l, Qx_hat

def x0_TLN(dt0, dn0, beta0, gamma_t0, gamma_l0):
    """
    Function to compute the initial guess vector for x
    """

    x = np.matrix([[dt0], [dn0], [beta0], [gamma_t0], [gamma_l0]])

    return x


def y_pseudo(y_los, beta_pseudo, gamma_t_pseudo, gamma_l_pseudo):
    """
    Function to compute the y vector with pseudo observations for the angles


    """

    y = np.zeros((len(y_los)+3, 1))
    y[0:len(y_los)] = y_los
    y[-1] = gamma_l_pseudo
    y[-2] = gamma_t_pseudo
    y[-3] = beta_pseudo

    return y


##########################################################################################
# MODULE 10: Estimate optimal viewing geometry

# WARNING: Those functions are only working when the TLN frame can be used and we asume zero
# longitudinal displacements
##########################################################################################


def best_geometry(inc_sat, alpha_d_sat, std_sat, inc_new, alpha_d_new, std_new, TLN_frame, def_signal):
    """
    Function to estimate the best viewing geometry

    Input:
        Inc_sat: array with incidence angles of the available satellites [radians]
        alpha_d_sat: array with alpha_d values of the available satellites [radians]
        std_sat: array with std_values of the available satellites
        inc_new: array with possible incidence angles [radians]
        alpha_d_new: array with possible values for alpha_d [radians]
        std_new: value for the standard deviation for the new satellite
        def_signal: 3x1 vector with estimated displacement signal: [d_t, d_l, d_n]^T

    Output:
        std_trans0: values for std transversal computed at method without TLN frame uncertainty for all inc and
        std_normal0: values for std normal computed at method without TLN frame uncertainty for all inc and alp

        std_trans: values for std transversal computed at method WITH TLN frame uncertainty for all inc and alp
        std_normal: values for std normal computed at method WITH TLN frame uncertainty for all inc and alpha_d

        inc_opt_t: best value for the incidence angle [radians] (for the transversal comp)
        alpha_d_opt_t: best value for the alpha_d [radians] (for the normal comp)
    """

    N = len(inc_new)
    M = len(alpha_d_new)

    #Uncertainty radar signal (0 method)
    std_los = np.append(std_sat, std_new)
    Qyy0 = np.zeros((len(std_sat)+1, len(std_sat)+1))
    np.fill_diagonal(Qyy0,std_los)

    # Create zero matrices
    std_trans = np.zeros((M,N))
    std_normal = np.zeros((M,N))
    std_trans0 = np.zeros((M,N))
    std_normal0 = np.zeros((M,N))

    for j in range(M):
        for i in range(N):
            inc2 = np.deg2rad(inc_new[i])
            alpha_d2 = np.deg2rad(alpha_d_new[j])

            inc = np.append(inc_sat, inc2)
            alpha_d = np.append(alpha_d_sat, alpha_d2)


            # Compute STD_trans and normal without taking uncertainty TLN frame into account
            A_proj = projection_matrix(inc, alpha_d, TLN_frame[0], TLN_frame[1], TLN_frame[2])
            los_obs = A_proj@def_signal

            A2 = np.delete(A_proj, 1, 1)
            x_hat0, Qx_hat0 = BLUE(A2, los_obs, Qyy0)
            std_trans0[j,i] = np.sqrt(Qx_hat0[0,0])
            std_normal0[j,i] = np.sqrt(Qx_hat0[1,1])

            # Compute STD_trans and normal WITH taking uncertainty TLN frame into account
            # 1. Compute 'expected y vector (needed for linearization)'
            y = TLN_forward_model(inc, alpha_d, TLN_frame[0], TLN_frame[1], TLN_frame[2], def_signal[0,0], def_
            # 2.Compute the Qyy matrix for TLN method with uncertainty TLN frame
            Qyy = TLN_Qyy(std_los, TLN_frame[3], TLN_frame[4], TLN_frame[5])

            # 3. Compute Qx_hat for the TLN jacobian matrix
            Qx_hat = TLN_Qxhat(inc, alpha_d, TLN_frame[0], TLN_frame[1], TLN_frame[2], def_signal[0,0], def_sig

            std_trans[j,i] = np.sqrt(Qx_hat[0,0])
            std_normal[j,i] = np.sqrt(Qx_hat[1,1])

    best_trans = np.where(std_trans == np.amin(std_trans))
    alpha_d_opt_t = alpha_d_new[best_trans[0]]
    inc_opt_t = inc_new[best_trans[1]]

    best_normal = np.where(std_normal == np.amin(std_normal))
    alpha_d_opt_n = alpha_d_new[best_normal[0]]
    inc_opt_n = inc_new[best_normal[1]]

    return std_trans, std_normal, alpha_d_opt_t, inc_opt_t, alpha_d_opt_n, inc_opt_n
```

2.1 Biased estimates

Notebook to compute the biases when the north component is neglected. Also the relation with the orientation of the null line is taken into account.

In [ ]:
```
import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib.colors

import scipy.stats as st
from scipy.stats import norm
from scipy import stats
from scipy.stats.distributions import chi2
from basic_geometric_functions import *
from functions_strap_down_method import *
```

In [2]:
```
def phi_zeta(inc, alpha_d):
    """
    Compute phi and zeta when the viewing geometry of the two satellites is known

    Based on the cross product between the two normal vectors

    The cross product between the two normal LoS vectors determines the direction of the line.
    From the direction, we can determine phi and zeta

    """

    # The cross product between the two LoS vectors
    dir_null_space_e = np.sin(inc[0])*np.cos(alpha_d[0])*np.cos(inc[1]) - np.cos(inc[0])*np.sin(inc[1])*np.cos(
    dir_null_space_n = -np.sin(inc[0])*np.sin(alpha_d[0])*np.cos(inc[1]) + np.cos(inc[0])*np.sin(inc[1])*np.sin
    dir_null_space_u = np.sin(inc[0])*np.sin(alpha_d[0])*np.sin(inc[1])*np.cos(alpha_d[1]) - np.sin(inc[0])*np.

    # Compute ground projection of the line
    ground_proj = np.sqrt(dir_null_space_e**2 + dir_null_space_n**2)

    # Compute phi (angle between NS line)
    phi = np.rad2deg(np.arctan(dir_null_space_e / dir_null_space_n))

    # Compute zeta (angle between NE plane and the line)
    zeta = np.rad2deg(np.arctan(dir_null_space_u / ground_proj))

    return phi, zeta
```

In [3]:
```
def vector2plane(normal_vec, point, x, y):
    """
    function to compute the value for d in the formula

    Compute the surfaces perpendicular to the normal vector (normal_vec)
    When we have a vector [a,b,c] then we have the plane equation: a*x+b*y+c*z+d=0
    We need to compute d --> when we know a point on the plane we can compute d.
    The point at the plane is given by the end point of the unit LoS vector [a,b,c]

    When x and y are the limits for the planes this functions computes the values for z
    """

    # d can be computed with the dot product of the point and the normal vector
    d = -point.dot(normal_vec)

    # Create the meshgrid for x and y
    xx, yy = np.meshgrid(x,y)

    # Calculate corresponding z
    z = (-normal_vec[0] * xx - normal_vec[1] * yy - d) * 1. /normal_vec[2]

    return d, z, xx, yy
```

In [4]:
```
def plane_intersect(a, b):
    """
    Compute the intersection line of the equations from two lines

    a, b   4-tuples/lists
           Ax + By +Cz + D = 0
           A,B,C,D in order

    output: 2 points on line of intersection, np.arrays, shape (3,)
    """
    a_vec, b_vec = np.array(a[:3]), np.array(b[:3])

    aXb_vec = np.cross(a_vec, b_vec)

    A = np.array([a_vec, b_vec, aXb_vec])
    d = np.array([-a[3], -b[3], 0.]).reshape(3,1)

# could add np.linalg.det(A) == 0 test to prevent linalg.solve throwing error

    p_inter = np.linalg.solve(A, d).T

    return p_inter[0], (p_inter + aXb_vec)[0]


a, b = (1, -1, 0, 2), (-1, -1, 1, 3)
plane_intersect(a, b)
```

Out[4]:
```
(array([ 0.,  2., -1.]), array([-1.,  1., -3.]))
```

In [5]:
```
def null_space_2sat(inc, alpha_d, plot):


    ######## Compute unit LoS vectors ############
    r = 1
    east_los = r*np.sin(inc)*np.sin(alpha_d)
    north_los = r*np.sin(inc)*np.cos(alpha_d)
    up_los = r*np.cos(inc)

    # Compute the unit LoS vectors
    unit_los_enu = np.zeros((3,1))

    for i in range(len(inc)):
        temp = np.matrix([[east_los[i]], [north_los[i]], [up_los[i]]])
        unit_los_enu = np.hstack((unit_los_enu, temp))

    unit_los_enu = np.delete(unit_los_enu, 0, 1)
    #print (unit_los_enu)

    ############# Compute null space surfaces for both the satellites ##############
    # Compute the surfaces perpendicular to the unit LoS vectors
    # When we have a vector [a,b,c] then we have the plane equation: a*x+b*y+c*z+d=0
    # We need to compute d --> when we know a point on the plane we can compute d.
    # The point at the plane is given by the end point of the unit LoS vector [a,b,c]

    x, y = np.linspace(-1,1,100), np.linspace(-1,1,100)

    # define normal vector and point for the plane corresponding to the asc obs.
    point_asc  = np.array([unit_los_enu[0,0], unit_los_enu[1,0], unit_los_enu[2,0]])
    normal_asc = np.array([unit_los_enu[0,0], unit_los_enu[1,0], unit_los_enu[2,0]])

    # define normal vector and point for the plane corresponding to the asc obs.
    point_dsc  = np.array([unit_los_enu[0,1], unit_los_enu[1,1], unit_los_enu[2,1]])
    normal_dsc = np.array([unit_los_enu[0,1], unit_los_enu[1,1], unit_los_enu[2,1]])

    # Compute the z coordinates for the two planes and the d values
    d_asc, z_asc, xx, yy = vector2plane(normal_asc, point_asc, x, y)
    d_dsc, z_dsc, xx, yy = vector2plane(normal_dsc, point_dsc, x, y)


    ############## Compute equation for the intersection line of the two planes ############
    # Compute the equation for the intersection line of the two planes
    # Based on the functions of the two planes we can compute the intersection line.

    # Define coefficients of the two null spaces (for asc en dsc)
    coef_asc = (normal_asc[0], normal_asc[1], normal_asc[2], d_asc)
    coef_dsc= (normal_dsc[0], normal_dsc[1], normal_dsc[2], d_dsc)

    # Compute the coordinates for two points at the intersection line
    pnt_one, pnt_two = plane_intersect(coef_asc, coef_dsc)

    # Try the vector equation for the two points (such that the line becomes longer)
    t = np.linspace(-1,1,100)
    dir_vec = pnt_two-pnt_one

    x_line = np.zeros(100)
    y_line = np.zeros(100)
    z_line = np.zeros(100)


    for i in range(len(t)):
        line_coor = pnt_one + t[i]*dir_vec
        x_line[i] = line_coor[0]
        y_line[i] = line_coor[1]
        z_line[i] = line_coor[2]


    ############# Compute angles for the intersection line ##################

    # Compute differences between the two points
    e_diff = -1*(pnt_one[0] - pnt_two[0])
    n_diff = -1*(pnt_one[1] - pnt_two[1])
    u_diff = -1*(pnt_one[2] - pnt_two[2])

    # Compute ground projection of the line
    ground_proj = np.sqrt(e_diff**2 + n_diff**2)

    # Compute phi (angle between NS line)
    phi = np.rad2deg(np.arctan(e_diff / n_diff))

    # Compute zeta (angle between NE plane and the line)
    zeta = np.rad2deg(np.arctan(u_diff / ground_proj))

    z_ground = np.zeros(100)

    psi = np.rad2deg(np.arctan(u_diff / e_diff))


    if plot == 1:
        plt3d = plt.figure(figsize=(8,8)).gca(projection='3d')
        # plot the two surfaces
        plt3d.plot_surface(xx, yy, z_asc, alpha=0.5, rstride=100, cstride=100, color = 'b')
        plt3d.plot_surface(xx, yy, z_dsc, alpha=0.5, rstride=100, cstride=100, color = 'green')

        # Plot the unit LoS vectors
        plt3d.plot([0,unit_los_enu[0,0]],[0,unit_los_enu[1,0]], [0,unit_los_enu[2,0]], label='LoS asc', linewid
        plt3d.plot([0,unit_los_enu[0,1]],[0,unit_los_enu[1,1]], [0,unit_los_enu[2,1]], label='LoS desc', linewi

        # Plot the 'projections' of the LoS vectors
        plt3d.plot([unit_los_enu[0,0],unit_los_enu[0,0]],[unit_los_enu[1,0],unit_los_enu[1,0]], [0,unit_los_enu
                   linewidth=2, color = 'b', ls = '--', alpha = 0.6)
        plt3d.plot([0,unit_los_enu[0,0]],[0,unit_los_enu[1,0]], [0,0], linewidth=2, color = 'b',
                  ls = '--', alpha = 0.6)

        plt3d.plot([unit_los_enu[0,1],unit_los_enu[0,1]],[unit_los_enu[1,1],unit_los_enu[1,1]], [0,unit_los_enu
                   linewidth=2, color = 'g', ls = '--', alpha = 0.6)
        plt3d.plot([0,unit_los_enu[0,1]],[0,unit_los_enu[1,1]], [0,0], linewidth=2, color = 'g',
                  ls = '--', alpha = 0.6)



        # Plot the intersection line
        plt3d.plot(x_line, y_line, z_line, lw=4, c='r', label = 'Null space')
        plt3d.plot(x_line, y_line, z_ground, lw=2, c='r', ls = '--', label = 'Ground projection', alpha = 0.7)


        #plt.legend(fontsize = 12)
        plt.ylabel('North', fontsize = 12)
        plt.xlabel('East', fontsize = 12)
        plt.title(r'$\phi$ = '+ str(np.around(phi, 2)) + ', $\zeta$ = ' + str(np.around(zeta, 2))
                  + ', $\psi$ = ' + str(np.around(psi, 2)), fontsize = 20)

        plt.show()


    return zeta, phi, psi
```

Simulate LoS observations

In [6]:
```
# Displacement signal
d_t = 4
d_l = 10
d_n = 4
d_tln = np.matrix([[d_t], [d_l], [d_n]])


# The orientation (for the forword model, to simulate the observations)
beta = 0
gamma_t = 0
gamma_l = 0


# Satellite characteristics
asc_inc = np.array([32.2])
asc_zero_doppler = np.array([258.5])
dsc_inc = np.array([32.0])
dsc_zero_doppler = np.array([101.5])

inc = np.hstack([asc_inc, dsc_inc])
alpha_d = np.hstack([asc_zero_doppler, dsc_zero_doppler])

# Noise
std_los = 0.0000001 #mm

A_proj = projection_matrix(inc, alpha_d, beta, gamma_t, gamma_l)
#print (A_proj)
y_los = A_proj@d_tln
#print ('The LoS observations are:', y_los)

# Add noise to LoS observations
noise = np.random.normal(0, std_los, len(inc))
n = np.zeros((len(inc),1))
n[:,0] = noise

y_los_noise = y_los + n

# Estimating the displacement signal (A) for the full case (also NS) and (B) with NS removed

Qyy = np.identity(len(inc))*std_los**2
x_hat_full, Qx_hat_full = BLUE(A_proj, y_los_noise, Qyy)

A2 = np.hstack((A_proj[:,0], A_proj[:,2]))
x_hat, Qx_hat = BLUE(A2, y_los_noise, Qyy)

D = np.zeros((3,7))
D[:,0] = np.array([d_t, d_l, d_n])
D[:,1] = np.array([x_hat[0,0], 0, x_hat[1,0]])
D[:,2] = np.array([np.abs((x_hat[0,0]-d_t)/d_t)*100, 0, np.abs((x_hat[1,0]-d_n)/d_n)*100])
D[:,3] = np.array([np.sqrt(Qx_hat[0,0]), 0, np.sqrt(Qx_hat[1,1])])
D[:,4] = np.array([x_hat_full[0,0], x_hat_full[1,0], x_hat_full[2,0]])
D[:,5] = np.array([np.abs((x_hat_full[0,0]-d_t)/d_t)*100, np.abs((x_hat_full[1,0]-d_l)/d_l)*100, np.abs((x_hat_
D[:,6] = np.array([np.sqrt(Qx_hat_full[0,0]), np.sqrt(Qx_hat_full[1,1]), np.sqrt(Qx_hat_full[2,2])])

c = ['Simulated value', 'Estimated (no NS)', '% error', 'Precision no NS', 'Estimated full', '% error', 'Precis
r = ['East', 'North', 'Up']

DATA = pd.DataFrame(data=D, columns=c, index = r)

display(DATA)
```

	Simulated value	Estimated (no NS)	% error	Precision no NS	Estimated full	% error	Precision full
East	4.0	4.007888	0.197199	1.357912e-07	4.023750	0.593761	0.050228
North	10.0	0.000000	0.000000	0.000000e+00	-11.453784	214.537838	63.676064
Up	4.0	2.749381	31.265482	8.347186e-08	-1.331958	133.298939	7.963451

In [7]:
```
print (A_proj)
print ((y_los))
```

```
[[-0.52217863 -0.10623844  0.84619317]
 [ 0.51928098 -0.10564891  0.8480481 ]]
[[0.23367373]
 [4.41282721]]
```

In [8]:
```
# Compute the projection of the vector onto the Up-East plane

#components of the LoS vector
los_comp1 = np.matrix([[A_proj[0,0]*y_los[0,0]], [A_proj[0,1]*y_los[0,0]], [A_proj[0,2]*y_los[0,0]]])
#print (los_comp1)

# Compute projection onto plane
dot1 = los_comp1[1,0]
normal_vec = np.matrix([[0], [1], [0]])
#print (normal_vec*dot1)

los_plane1 = los_comp1 - normal_vec*dot1
print (los_plane1)


#components of the LoS vector
los_comp2 = np.matrix([[A_proj[1,0]*y_los[1,0]], [A_proj[1,1]*y_los[1,0]], [A_proj[1,2]*y_los[1,0]]])
#print (los_comp1)

# Compute projection onto plane
dot2 = los_comp2[1,0]
normal_vec = np.matrix([[0], [1], [0]])
#print (normal_vec*dot1)

los_plane2 = los_comp2 - normal_vec*dot2
print (los_plane2)

los1 = los_plane1
```

```
[[-0.12201943]
 [ 0.        ]
 [ 0.19773311]]
[[2.29149723]
 [0.        ]
 [3.74228971]]
```

In [9]:
```
los_plane1+los_plane2
```

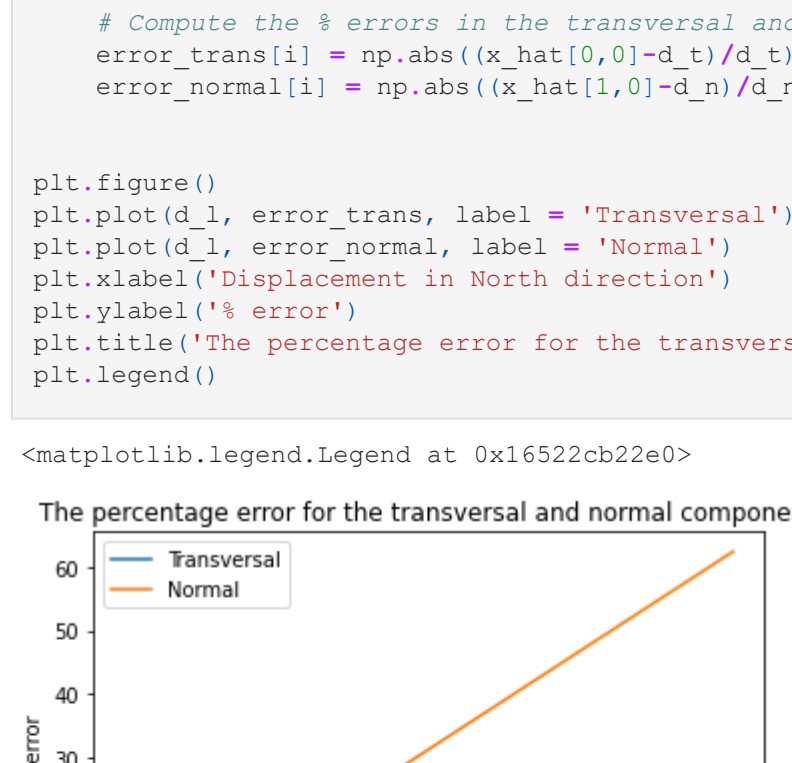
Out[9]:
```
matrix([[2.1694778 ],
        [0.        ],
        [3.94002283]])
```

Create figures where bias is plotted vs:

1. Size of displacement signal
2. Change in orientation of the incidence angle (desc)
3. Change in orientation of the heading angle (desc)

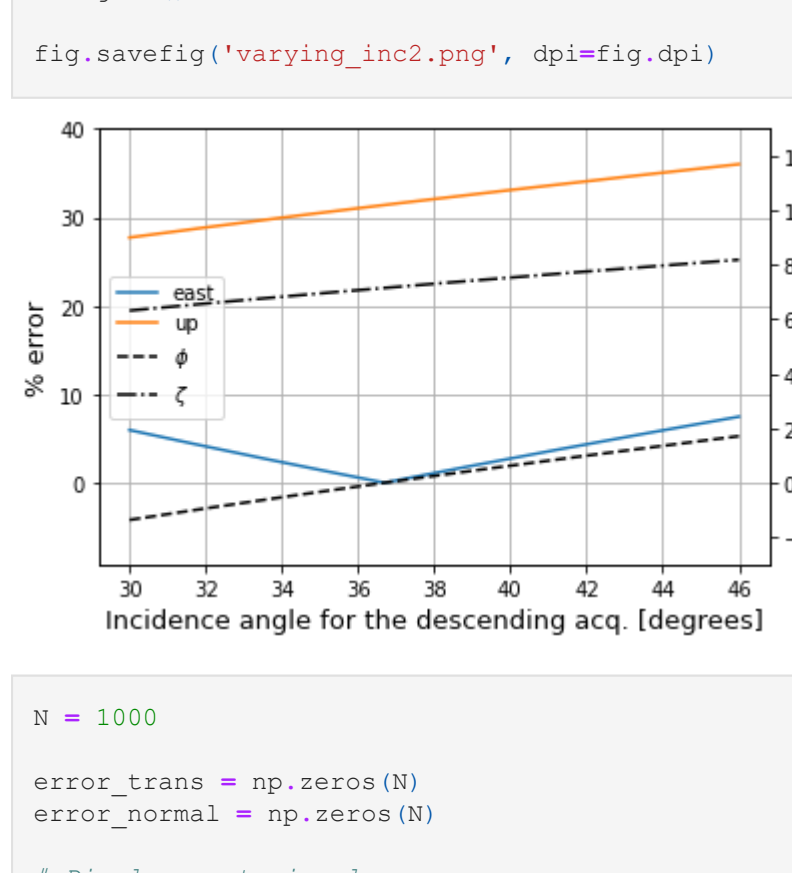
In [10]:
```
N = 100

error_trans = np.zeros(N)
error_normal = np.zeros(N)



# Displacement signal
d_t = 4
d_l = np.linspace(0,20,N)
d_n = 4


# The orientation (for the forword model, to simulate the observations)
beta = 0
gamma_t = 0
gamma_l = 0


# Satellite characteristics
asc_inc = np.array([32.2])
asc_zero_doppler = np.array([258.5])
dsc_inc = np.array([31.9])
dsc_zero_doppler = np.array([101.5])



# Noise
std_los = 0.0000001 #mm

for i in range(N):
    d_tln = np.matrix([[d_t], [d_l[i]], [d_n]])

    inc = np.hstack([asc_inc, dsc_inc])
    alpha_d = np.hstack([asc_zero_doppler, dsc_zero_doppler])

    A_proj = projection_matrix(inc, alpha_d, beta, gamma_t, gamma_l)
    #print (A_proj)
    y_los = A_proj@d_tln
    #print ('The LoS observations are:', y_los)

    # Add noise to LoS observations
    noise = np.random.normal(0, std_los, len(inc))
    n = np.zeros((len(inc),1))
    n[:,0] = noise

    y_los_noise = y_los + n

    # Estimating the displacement signal (A) for the full case (also NS) and (B) with NS removed

    Qyy = np.identity(len(inc))*std_los**2
    x_hat_full, Qx_hat_full = BLUE(A_proj, y_los_noise, Qyy)

    A2 = np.hstack((A_proj[:,0], A_proj[:,2]))
    x_hat, Qx_hat = BLUE(A2, y_los_noise, Qyy)

    # Compute the % errors in the transversal and normal direction
    error_trans[i] = np.abs((x_hat[0,0]-d_t)/d_t)*100
    error_normal[i] = np.abs((x_hat[1,0]-d_n)/d_n)*100


plt.figure()
plt.plot(d_l, error_trans, label = 'Transversal')
plt.plot(d_l, error_normal, label = 'Normal')
plt.xlabel('Displacement in North direction')
plt.ylabel('% error')
plt.title('The percentage error for the transversal and normal component')
plt.legend()
```

Out[10]:
```
<matplotlib.legend.Legend at 0x16522cb22e0>
```



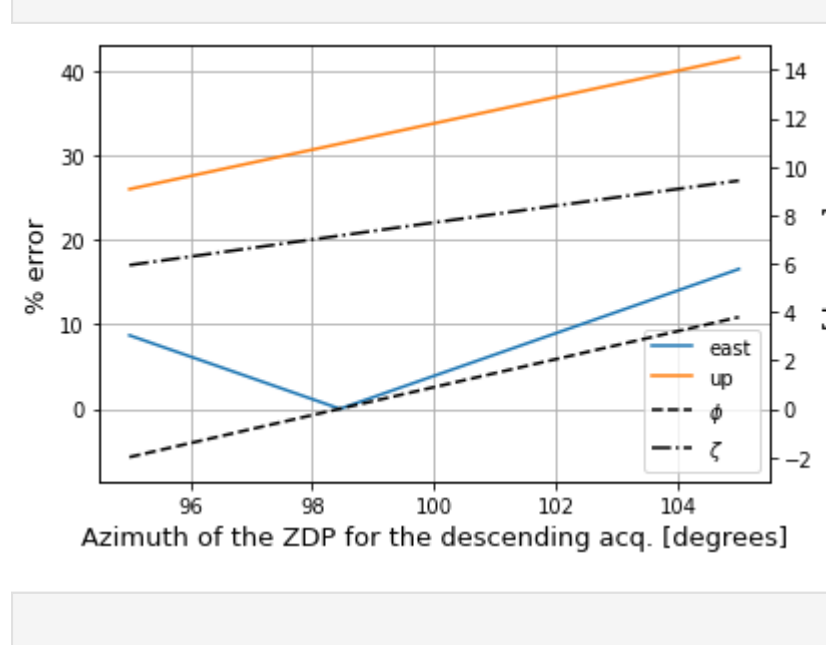
In [11]:
```
N = 1000

error_trans = np.zeros(N)
error_normal = np.zeros(N)
phi = np.zeros(N)
zeta = np.zeros(N)

# Displacement signal
d_t = 4
d_l = 10
d_n = 4


# The orientation (for the forword model, to simulate the observations)
beta = 0
gamma_t = 0
gamma_l = 0


# # Satellite characteristics
# asc_inc = np.array([35])
# asc_zero_doppler = np.array([350-90])
# dsc_inc = np.linspace(30,46,N)
# dsc_zero_doppler = np.array([190-90])

# Satellite characteristics
asc_inc = np.array([32.2])
asc_zero_doppler = np.array([258.5])
dsc_inc = np.linspace(30,46,N)
dsc_zero_doppler = np.array([99.7])



# Noise
std_los = 0.0000001 #mm

for i in range(N):
    d_tln = np.matrix([[d_t], [d_l], [d_n]])

    inc = np.hstack([asc_inc, dsc_inc[i]])
    alpha_d = np.hstack([asc_zero_doppler, dsc_zero_doppler])

    A_proj = projection_matrix(inc, alpha_d, beta, gamma_t, gamma_l)
    #print (A_proj)
    y_los = A_proj@d_tln
    #print ('The LoS observations are:', y_los)

    # Add noise to LoS observations
    noise = np.random.normal(0, std_los, len(inc))
    n = np.zeros((len(inc),1))
    n[:,0] = noise

    y_los_noise = y_los + n

    # Estimating the displacement signal (A) for the full case (also NS) and (B) with NS removed

    Qyy = np.identity(len(inc))*std_los**2
    x_hat_full, Qx_hat_full = BLUE(A_proj, y_los_noise, Qyy)

    A2 = np.hstack((A_proj[:,0], A_proj[:,2]))
    x_hat, Qx_hat = BLUE(A2, y_los_noise, Qyy)

    # Compute the % errors in the transversal and normal direction
    error_trans[i] = np.abs((x_hat[0,0]-d_t)/d_t)*100
    error_normal[i] = np.abs((x_hat[1,0]-d_n)/d_n)*100
    zeta[i], phi[i], psi = null_space_2sat(np.deg2rad(inc), np.deg2rad(alpha_d), 0)
```

In [12]:
```
fig, ax1 = plt.subplots()
a = ax1.plot(dsc_inc, error_trans, label = 'east')
b = ax1.plot(dsc_inc, error_normal, label = 'up')
ax1.set_xlabel('Incidence angle for the descending acq. [degrees]', fontsize = 13)
ax1.set_ylabel('% error', fontsize = 13)
#plt.title('The percentage error for the east and up com', fontsize = 15)

ax2 = ax1.twinx()

c = ax2.plot(dsc_inc, phi, label = r'$\phi$', ls = '--', color = 'black')
d = ax2.plot(dsc_inc, zeta, label = r'$\zeta$', ls = 'dashdot', color = 'black')
ax2.set_xlabel('Incidence angle for the descending acq. [degrees]', fontsize = 13)
ax2.set_ylabel('[degrees]', fontsize = 13)

a1 = 13
b1 = -3
c1 = 40
ax2.set_ylim(b1,a1)
r = np.abs(b1)/(a1+np.abs(b1))
d1 = (r*c1)/(r-1)
ax1.set_ylim(d1,c1)

#plt.title('The percentage error for the east and up com', fontsize = 15)

lns = a+b+c+d
labs = [l.get_label() for l in lns]
ax1.legend(lns, labs, loc=0, fontsize = 10)
ax1.grid()

fig.savefig('varying_inc2.png', dpi=fig.dpi)
```



In [13]:
```
N = 1000

error_trans = np.zeros(N)
error_normal = np.zeros(N)

# Displacement signal
d_t = 4
d_l = 10
d_n = 4


# The orientation (for the forword model, to simulate the observations)
beta = 0
gamma_t = 0
gamma_l = 0


# Satellite characteristics

asc_inc = np.array([32.2])
asc_zero_doppler = np.array([258.5])
dsc_inc = np.array([40.5])
dsc_zero_doppler = np.linspace(95,105,N)



# Noise
std_los = 0.0000001 #mm

for i in range(N):
    d_tln = np.matrix([[d_t], [d_l], [d_n]])

    inc = np.hstack([asc_inc, dsc_inc])
    alpha_d = np.hstack([asc_zero_doppler, dsc_zero_doppler[i]])

    A_proj = projection_matrix(inc, alpha_d, beta, gamma_t, gamma_l)
    #print (A_proj)
    y_los = A_proj@d_tln
    #print ('The LoS observations are:', y_los)

    # Add noise to LoS observations
    noise = np.random.normal(0, std_los, len(inc))
    n = np.zeros((len(inc),1))
    n[:,0] = noise

    y_los_noise = y_los + n

    # Estimating the displacement signal (A) for the full case (also NS) and (B) with NS removed

    Qyy = np.identity(len(inc))*std_los**2
    x_hat_full, Qx_hat_full = BLUE(A_proj, y_los_noise, Qyy)

    A2 = np.hstack((A_proj[:,0], A_proj[:,2]))
    x_hat, Qx_hat = BLUE(A2, y_los_noise, Qyy)

    # Compute the % errors in the transversal and normal direction
    error_trans[i] = np.abs((x_hat[0,0]-d_t)/d_t)*100
    error_normal[i] = np.abs((x_hat[1,0]-d_n)/d_n)*100
    zeta[i], phi[i], psi = null_space_2sat(np.deg2rad(inc), np.deg2rad(alpha_d), 0)
```

In [14]:
```
fig, ax1 = plt.subplots()
a = ax1.plot(dsc_zero_doppler, error_trans, label = 'east')
b = ax1.plot(dsc_zero_doppler, error_normal, label = 'up')
ax1.set_xlabel('Azimuth of the ZDP for the descending acq. [degrees]', fontsize = 13)
ax1.set_ylabel('% error', fontsize = 13)
#plt.title('The percentage error for the east and up com', fontsize = 15)

ax2 = ax1.twinx()

c = ax2.plot(dsc_zero_doppler,  phi, label = r'$\phi$', ls = '--', color = 'black')
d = ax2.plot(dsc_zero_doppler,  zeta, label = r'$\zeta$', ls = 'dashdot', color = 'black')
ax2.set_xlabel('Azimuth of the ZDP for the descending acq. [degrees]', fontsize = 13)
ax2.set_ylabel('[degrees]', fontsize = 13)
a1 = 15
b1 = -3
c1 = 43
ax2.set_ylim(b1,a1)
r = np.abs(b1)/(a1+np.abs(b1))
d1 = (r*c1)/(r-1)
ax1.set_ylim(d1,c1)

lns = a+b+c+d
labs = [l.get_label() for l in lns]
ax1.legend(lns, labs, loc = 4, fontsize = 10)
ax1.grid()
fig.savefig('varying_alpha2.png', dpi=fig.dpi)
```



In [ ]:

2.2 Solution and variance-covariance matrix

Notebook to visualize the solution in the ENU reference system when three observations are availble. The variance covariance matrices are also visualized

In [ ]:
```
%matplotlib notebook

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib.colors
from scipy.linalg import null_space

from functions_strap_down_method_nodrama import *
from functions import *

from scipy.linalg import null_space
from mpl_toolkits.mplot3d import axes3d
from mpl_toolkits.mplot3d import Axes3D
```

In [2]:
```
def vector2plane(normal_vec, point, x, y):
    """
    function to compute the value for d in the formula

    Compute the surfaces perpendicular to the normal vector (normal_vec)
    When we have a vector [a,b,c] then we have the plane equation: a*x+b*y+c*z+d=0
    We need to compute d --> when we know a point on the plane we can compute d.
    The point at the plane is given by the end point of the unit LoS vector [a,b,c]

    When x and y are the limits for the planes this functions computes the values for z
    """

    # d can be computed with the dot product of the point and the normal vector
    d = -point.dot(normal_vec)

    # Create the meshgrid for x and y
    xx, yy = np.meshgrid(x,y)

    # Calculate corresponding z
    z = (-normal_vec[0] * xx - normal_vec[1] * yy - d) * 1. /normal_vec[2]

    return d, z, xx, yy
```

In [3]:
```
def plane_intersect(a, b):
    """
    Compute the intersection line of the equations from two lines

    a, b   4-tuples/lists
           Ax + By +Cz + D = 0
           A,B,C,D in order

    output: 2 points on line of intersection, np.arrays, shape (3,)
    """
    a_vec, b_vec = np.array(a[:3]), np.array(b[:3])

    aXb_vec = np.cross(a_vec, b_vec)

    A = np.array([a_vec, b_vec, aXb_vec])
    d = np.array([-a[3], -b[3], 0.]).reshape(3,1)

# could add np.linalg.det(A) == 0 test to prevent linalg.solve throwing error

    p_inter = np.linalg.solve(A, d).T

    return p_inter[0], (p_inter + aXb_vec)[0]


a, b = (1, -1, 0, 2), (-1, -1, 1, 3)
plane_intersect(a, b)
```

Out[3]:
```
(array([ 0.,  2., -1.]), array([-1.,  1., -3.]))
```

Define the viewing geometry + def.signal

In [35]:
```
# define right/left looking right = 1, left = 2
acq = np.array([1,1,1])

# Define AZD and incidence angle:
inc = np.deg2rad(np.array([31, 41, 44])) # two asc satellites and one dsc satellite
alpha_d = np.deg2rad(np.array([351-90,350-90,190-90]))

# define std of the LoS observations
std_los = np.array([1,1,1])

d_e = -5
d_n = -5
d_u = 10

def_signal = np.matrix([[d_e], [d_n], [d_u]])
A_proj = projection_matrix(inc, alpha_d, 0, 0, 0)
los = A_proj@def_signal

print (los)
```

```
[[11.51800704]
 [11.34717317]
 [ 4.37600406]]
```

In [36]:
```
Qyy = np.identity(3)
x_hat, Qx_hat = BLUE(A_proj, los, Qyy)
std_e = np.sqrt(Qx_hat[0,0])
std_n = np.sqrt(Qx_hat[1,1])
std_u = np.sqrt(Qx_hat[2,2])
std_e, std_n, std_u, Qx_hat
```

Out[36]:
```
(1.185445630787017,
 27.588027396963277,
 3.898103399686959,
 matrix([[  1.40528134,  18.41531762,   2.77723525],
         [ 18.41531762, 761.09925566, 105.38641408],
         [  2.77723525, 105.38641408,  15.19521011]]))
```

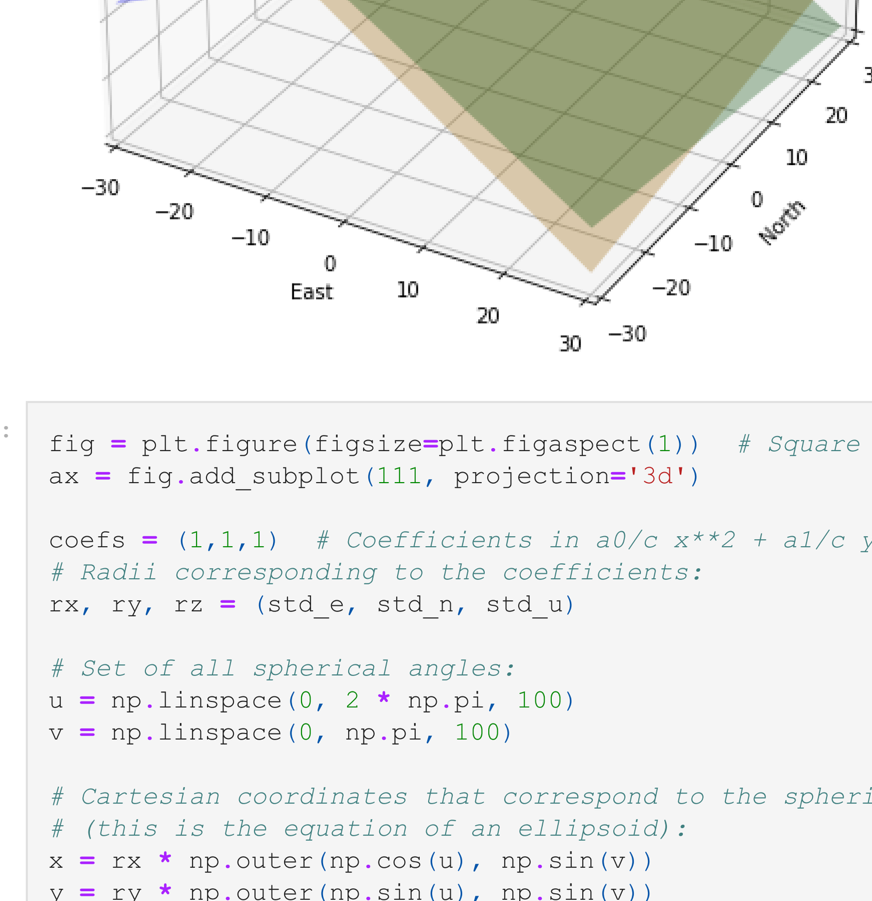
In [27]:
```
east_los = np.zeros(len(acq))
north_los = np.zeros(len(acq))
up_los = np.zeros(len(acq))

x, y = np.linspace(-30,30,100), np.linspace(-30,30,100)
z = np.zeros((len(x), len(y), len(acq)))

for i in range(len(acq)):
    if acq[i] == 1:
        # Compute LoS unit vectors
        r = los[i,0]
        print (r)
        east_los[i] = r*np.sin(inc[i])*np.sin(alpha_d[i])
        north_los[i] = r*np.sin(inc[i])*np.cos(alpha_d[i])
        up_los[i] = r*np.cos(inc[i])

        # Compute the unit LoS vectors
        unit_los_enu = np.matrix([[east_los[i]], [north_los[i]], [up_los[i]]])

    print ('right looking')

    if acq[i] == 2:
        # Compute LoS unit vectors
        r = 1

        east_los[i] = -r*np.sin(inc[i])*np.sin(alpha_d[i])
        north_los[i] = -r*np.sin(inc[i])*np.cos(alpha_d[i])
        up_los[i] = r*np.cos(inc[i])

        # Compute the unit LoS vectors
        unit_los_enu = np.matrix([[east_los[i]], [north_los[i]], [up_los[i]]])


    # Compute the surfaces perpendicular to the unit LoS vectors
    # When we have a vector [a,b,c] then we have the plane equation: a*x+b*y+c*z+d=0
    # We need to compute d --> when we know a point on the plane we can compute d.
    # The point at the plane is given by the end point of the unit LoS vector [a,b,c]

    # define normal vector and point for the plane corresponding to the los obs.
    point  = np.array([east_los[i], north_los[i], up_los[i]])
    normal = np.array([east_los[i], north_los[i], up_los[i]])
    # Compute the z coordinates for the plane
    d, z[:,:,i], xx, yy = vector2plane(normal, point, x, y)
```

```
11.518007043396775
right looking
right looking
4.3760040594286975
right looking
```

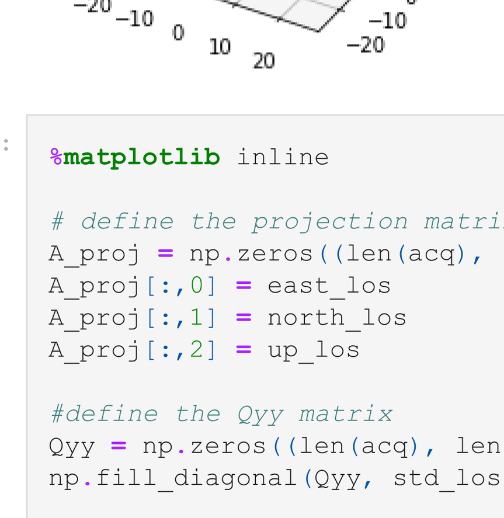
In [28]:
```
plt3d = plt.figure(figsize=(8,8)).gca(projection='3d')

# plot the two surfaces
plt3d.plot_surface(xx, yy, z[:,:,0], alpha=0.3, rstride=100, cstride=100, color = 'b')
plt3d.plot_surface(xx, yy, z[:,:,1], alpha=0.3, rstride=100, cstride=100, color = 'g')
plt3d.plot_surface(xx, yy, z[:,:,2], alpha=0.3, rstride=100, cstride=100, color = 'orange')




# Plot the unit LoS vectors
plt3d.plot([0,east_los[0]],[0,north_los[0]], [0,up_los[0]], label='LoS asc 1', linewidth=4, color = 'b')
plt3d.plot([0,east_los[1]],[0,north_los[1]], [0,up_los[1]], label='LoS asc 2', linewidth=4, color = 'g')
plt3d.plot([0,east_los[2]],[0,north_los[2]], [0,up_los[2]], label='LoS dsc', color = 'orange', linewidth=4)


# Plot the projections on the ground
# Plot the 'projections' of the LoS vectors
plt3d.plot([east_los[0],east_los[0]],[north_los[0],north_los[0]], [0,up_los[0]], linewidth=2, color = 'b', ls =
plt3d.plot([east_los[1],east_los[1]],[north_los[1],north_los[1]], [0,up_los[1]], linewidth=2, color = 'g', ls =
plt3d.plot([east_los[2],east_los[2]],[north_los[2],north_los[2]], [0,up_los[2]], linewidth=2, color = 'orange',

plt3d.plot([0,east_los[0]],[0,north_los[0]], [0,0], linewidth=2, color = 'b', ls = '--', alpha = 0.6)
plt3d.plot([0,east_los[1]],[0,north_los[1]], [0,0], linewidth=2, color = 'g', ls = '--', alpha = 0.6)
plt3d.plot([0,east_los[2]],[0,north_los[2]], [0,0], linewidth=2, color = 'orange', ls = '--', alpha = 0.6)

plt3d.scatter(d_e, d_n, d_u, color = 'red', s = 40)

rx, ry, rz = (std_e, std_n, std_u)

# Set of all spherical angles:
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)

# Cartesian coordinates that correspond to the spherical angles:
# (this is the equation of an ellipsoid):
x = rx * np.outer(np.cos(u), np.sin(v))
y = ry * np.outer(np.sin(u), np.sin(v))
z2 = rz * np.outer(np.ones_like(u), np.cos(v))

# Plot:
plt3d.plot_surface(x+d_e, y+d_n, z2+d_u,  rstride=4, cstride=4, color='r', alpha = 0.4)



plt.legend()
plt.ylabel('North')
plt.xlabel('East')
plt3d.set_zlim3d(-30,30)
plt3d.set_ylim3d(-30,30)
plt3d.set_xlim3d(-30,30)


plt.show
```

```
<ipython-input-28-f26a7bbee856>:1: MatplotlibDeprecationWarning: Calling gca() with keyword arguments was depre
cated in Matplotlib 3.4. Starting two minor releases later, gca() will take no keyword arguments. The gca() fun
ction should only be used to get the current axes, or if no axes exist, create new axes with default keyword ar
guments. To create a new axes with non-default arguments, use plt.axes() or plt.subplot().
  plt3d = plt.figure(figsize=(8,8)).gca(projection='3d')
```

Out[28]:
```
<function matplotlib.pyplot.show(close=None, block=None)>
```

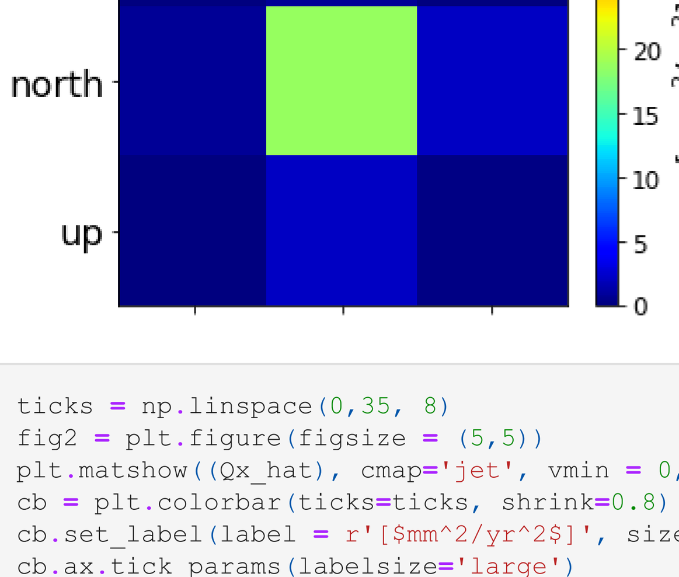


In [29]:
```
fig = plt.figure(figsize=plt.figaspect(1))  # Square figure
ax = fig.add_subplot(111, projection='3d')

coefs = (1,1,1)  # Coefficients in a0/c x**2 + a1/c y**2 + a2/c z**2 = 1
# Radii corresponding to the coefficients:
rx, ry, rz = (std_e, std_n, std_u)

# Set of all spherical angles:
u = np.linspace(0, 2 * np.pi, 100)
v = np.linspace(0, np.pi, 100)

# Cartesian coordinates that correspond to the spherical angles:
# (this is the equation of an ellipsoid):
x = rx * np.outer(np.cos(u), np.sin(v))
y = ry * np.outer(np.sin(u), np.sin(v))
z = rz * np.outer(np.ones_like(u), np.cos(v))

# Plot:
ax.plot_surface(x, y, z+10,  rstride=4, cstride=4, color='b')

# Adjustment of the axes, so that they all have the same span:
max_radius = max(rx, ry, rz)
for axis in 'xyz':
    getattr(ax, 'set_{}lim'.format(axis))((-max_radius, max_radius))

plt.show()
```

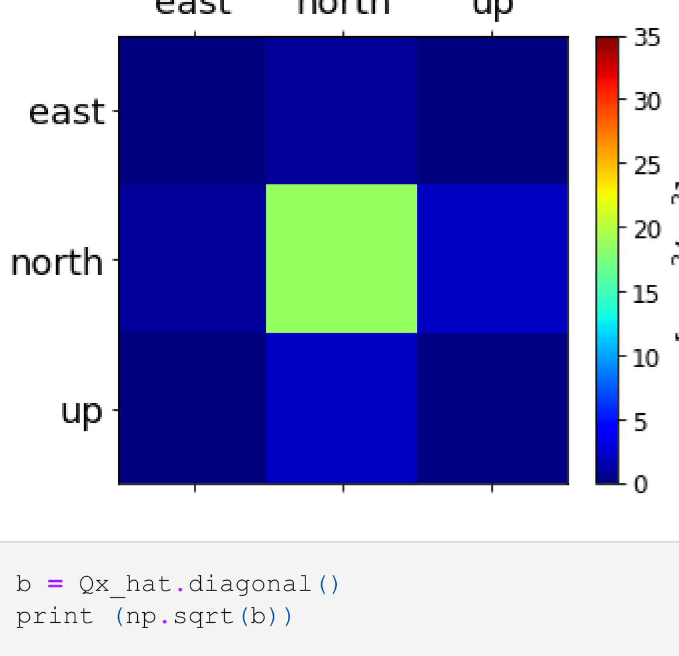


In [30]:
```
%matplotlib inline

# define the projection matrix
A_proj = np.zeros((len(acq), 3))
A_proj[:,0] = east_los
A_proj[:,1] = north_los
A_proj[:,2] = up_los

#define the Qyy matrix
Qyy = np.zeros((len(acq), len(acq)))
np.fill_diagonal(Qyy, std_los)

Qx_hat = np.linalg.inv(A_proj.T@np.linalg.inv(Qyy)@A_proj)
print (Qx_hat)


ticks = np.linspace(0,35, 8)
fig2 = plt.figure(figsize = (5,5))
plt.matshow((Qx_hat), cmap='jet', vmin = 0, vmax = 35, fignum = 1)
cb = plt.colorbar(ticks=ticks, shrink=0.8)
cb.set_label(label = r'[$mm^2/yr^2$]', size=16)
cb.ax.tick_params(labelsize='large')

#plt.title('Full covariance matrix for the deformation estimates', pad = 20)
#plt.xticks.set_ticks_position('bottom')
plt.xticks(np.linspace(0, 2, 3),['east', 'north', 'up'], fontsize=18)
plt.yticks(np.linspace(0, 2, 3),['east', 'north', 'up'], fontsize=18)
plt.show()
#plt.savefig("Full_covariance_matrix_threeLOS_right.jpg")
```

```
[[ 0.07066462  0.69811638  0.10081821]
 [ 0.69811638 18.69546614  2.16958474]
 [ 0.10081821  2.16958474  0.26983425]]
```



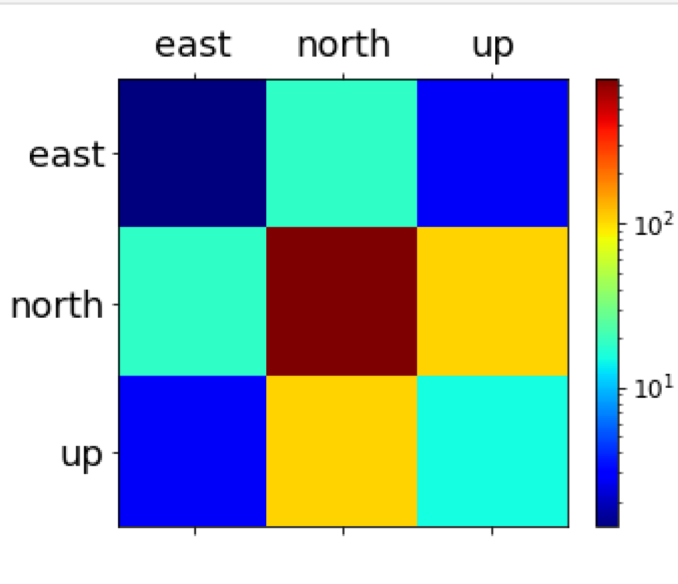
In [31]:
```
ticks = np.linspace(0,35, 8)
fig2 = plt.figure(figsize = (5,5))
plt.matshow((Qx_hat), cmap='jet', vmin = 0, vmax = 35, fignum = 1)
cb = plt.colorbar(ticks=ticks, shrink=0.8)
cb.set_label(label = r'[$mm^2/yr^2$]', size=16)
cb.ax.tick_params(labelsize='large')

#plt.title('Full covariance matrix for the deformation estimates', pad = 20)
#plt.xticks.set_ticks_position('bottom')
plt.xticks(np.linspace(0, 2, 3),['east', 'north', 'up'], fontsize=18)
plt.yticks(np.linspace(0, 2, 3),['east', 'north', 'up'], fontsize=18)
plt.show()
#plt.savefig("Full_covariance_matrix_threeLOS_left.jpg")
```



In [8]:
```
b = Qx_hat.diagonal()
print (np.sqrt(b))
```

```
[1.05844399 5.92446145 0.86231423]
```

In [37]:
```
from matplotlib import colors

ticks = np.linspace(0,800, 6)
plt.figure(figsize = (5,5))
plt.matshow((Qx_hat), cmap='jet', norm = colors.LogNorm(), fignum = 1)
cb = plt.colorbar(shrink=0.8)
cb.set_label(label = r'[$mm^2/yr^2$]', size=16)
cb.ax.tick_params(labelsize='large')

#plt.title('Full covariance matrix for the deformation estimates', pad = 20)
#plt.xticks.set_ticks_position('bottom')
plt.xticks(np.linspace(0, 2, 3),['east', 'north', 'up'], fontsize=18)
plt.yticks(np.linspace(0, 2, 3),['east', 'north', 'up'], fontsize=18)
plt.savefig("Full_covariance_matrix_three_LOS_rightlog.png")
```



2.3 Orientation of the null line

Notebook to compute the orientation of the null line for two LoS observations

In [ ]:
```
%matplotlib notebook

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib.colors
from scipy.linalg import null_space

from functions_strap_down_method_nodrama import *
from functions import *

from scipy.linalg import null_space
from mpl_toolkits.mplot3d import axes3d
from mpl_toolkits.mplot3d import Axes3D
```

In [2]:
```
def phi_zeta(inc, alpha_d):
    """
    Compute phi and zeta when the viewing geometry of the two satellites is known

    Based on the cross product between the two normal vectors

    The cross product between the two normal LoS vectors determines the direction of the line.
    From the direction, we can determine phi and zeta

    """


    # The cross product between the two LoS vectors
    dir_null_space_e = np.sin(inc[0])*np.cos(alpha_d[0])*np.cos(inc[1]) - np.cos(inc[0])*np.sin(inc[1])*np.cos(
    dir_null_space_n = -np.sin(inc[0])*np.sin(alpha_d[0])*np.cos(inc[1]) + np.cos(inc[0])*np.sin(inc[1])*np.sin
    dir_null_space_u = np.sin(inc[0])*np.sin(alpha_d[0])*np.sin(inc[1])*np.cos(alpha_d[1]) - np.sin(inc[0])*np.

    # Compute ground projection of the line
    ground_proj = np.sqrt(dir_null_space_e**2 + dir_null_space_n**2)

    # Compute phi (angle between NS line)
    phi = np.rad2deg(np.arctan(dir_null_space_e / dir_null_space_n))

    # Compute zeta (angle between NE plane and the line)
    zeta = np.rad2deg(np.arctan(dir_null_space_u / ground_proj))

    return phi, zeta
```

In [3]:
```
def vector2plane(normal_vec, point, x, y):
    """
    function to compute the value for d in the formula

    Compute the surfaces perpendicular to the normal vector (normal_vec)
    When we have a vector [a,b,c] then we have the plane equation: a*x+b*y+c*z+d=0
    We need to compute d --> when we know a point on the plane we can compute d.
    The point at the plane is given by the end point of the unit LoS vector [a,b,c]

    When x and y are the limits for the planes this functions computes the values for z
    """

    # d can be computed with the dot product of the point and the normal vector
    d = -point.dot(normal_vec)

    # Create the meshgrid for x and y
    xx, yy = np.meshgrid(x,y)

    # Calculate corresponding z
    z = (-normal_vec[0] * xx - normal_vec[1] * yy - d) * 1. /normal_vec[2]

    return d, z, xx, yy
```

In [4]:
```
def plane_intersect(a, b):
    """
    Compute the intersection line of the equations from two lines

    a, b   4-tuples/lists
           Ax + By +Cz + D = 0
           A,B,C,D in order

    output: 2 points on line of intersection, np.arrays, shape (3,)
    """
    a_vec, b_vec = np.array(a[:3]), np.array(b[:3])

    aXb_vec = np.cross(a_vec, b_vec)

    A = np.array([a_vec, b_vec, aXb_vec])
    d = np.array([-a[3], -b[3], 0.]).reshape(3,1)

# could add np.linalg.det(A) == 0 test to prevent linalg.solve throwing error

    p_inter = np.linalg.solve(A, d).T

    return p_inter[0], (p_inter + aXb_vec)[0]


a, b = (1, -1, 0, 2), (-1, -1, 1, 3)
plane_intersect(a, b)
```

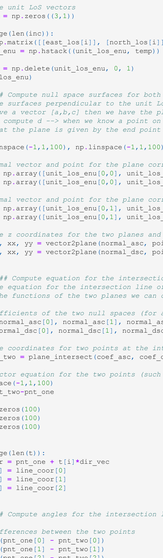
Out[4]:
```
(array([ 0.,  2., -1.]), array([-1.,  1., -3.]))
```

In [5]:
```
def null_space_2sat(inc, alpha_d, plot):


    ######## Compute unit LoS vectors ############
    r = 1
    east_los = r*np.sin(inc)*np.sin(alpha_d)
    north_los = r*np.sin(inc)*np.cos(alpha_d)
    up_los = r*np.cos(inc)

    # Compute the unit LoS vectors
    unit_los_enu = np.zeros((3,1))

    for i in range(len(inc)):
        temp = np.matrix([[east_los[i]], [north_los[i]], [up_los[i]]])
        unit_los_enu = np.hstack((unit_los_enu, temp))

    unit_los_enu = np.delete(unit_los_enu, 0, 1)
    print (unit_los_enu)

    ############ Compute null space surfaces for both the satellites #############
    # Compute the surfaces perpendicular to the unit LoS vectors
    # When we have a vector [a,b,c] then we have the plane equation: a*x+b*y+c*z+d=0
    # We need to compute d --> when we know a point on the plane we can compute d.
    # The point at the plane is given by the end point of the unit LoS vector [a,b,c]

    x, y = np.linspace(-1,1,100), np.linspace(-1,1,100)

    # define normal vector and point for the plane corresponding to the asc obs.
    point_asc = np.array([unit_los_enu[0,0], unit_los_enu[1,0], unit_los_enu[2,0]])
    normal_asc = np.array([unit_los_enu[0,0], unit_los_enu[1,0], unit_los_enu[2,0]])

    # define normal vector and point for the plane corresponding to the asc obs.
    point_dsc  = np.array([unit_los_enu[0,1], unit_los_enu[1,1], unit_los_enu[2,1]])
    normal_dsc = np.array([unit_los_enu[0,1], unit_los_enu[1,1], unit_los_enu[2,1]])

    # Compute the z coordinates for the two planes and the d values
    d_asc, z_asc, xx, yy = vector2plane(normal_asc, point_asc, x, y)
    d_dsc, z_dsc, xx, yy = vector2plane(normal_dsc, point_dsc, x, y)


    ############ Compute equation for the intersection line of the two planes ############
    # Compute the equation for the intersection line of the two planes
    # Based on the functions of the two planes we can compute the intersection line.

    # Define coefficients of the two null spaces (for asc en dsc)
    coef_asc = (normal_asc[0], normal_asc[1], normal_asc[2], d_asc)
    coef_dsc= (normal_dsc[0], normal_dsc[1], normal_dsc[2], d_dsc)

    # Compute the coordinates for two points at the intersection line
    pnt_one, pnt_two = plane_intersect(coef_asc, coef_dsc)

    # Try the vector equation for the two points (such that the line becomes longer)
    t = np.linspace(-1,1,100)
    dir_vec = pnt_two-pnt_one

    x_line = np.zeros(100)
    y_line = np.zeros(100)
    z_line = np.zeros(100)


    for i in range(len(t)):
        line_coor = pnt_one + t[i]*dir_vec
        x_line[i] = line_coor[0]
        y_line[i] = line_coor[1]
        z_line[i] = line_coor[2]


    ############ Compute angles for the intersection line #################

    # Compute differences between the two points
    e_diff = -1*(pnt_one[0] - pnt_two[0])
    n_diff = -1*(pnt_one[1] - pnt_two[1])
    u_diff = -1*(pnt_one[2] - pnt_two[2])

    # Compute ground projection of the line
    ground_proj = np.sqrt(e_diff**2 + n_diff**2)

    # Compute phi (angle between NS line)
    phi = np.rad2deg(np.arctan(e_diff / n_diff))

    # Compute zeta (angle between NE plane and the line)
    zeta = np.rad2deg(np.arctan(u_diff / ground_proj))

    z_ground = np.zeros(100)

    psi = np.rad2deg(np.arctan(u_diff / e_diff))


    if plot == 1:
        plt3d = plt.figure(figsize=(8,8)).gca(projection='3d')
        # plot the two surfaces
        plt3d.plot_surface(xx, yy, z_asc, alpha=0.5, rstride=100, cstride=100, color = 'b')
        plt3d.plot_surface(xx, yy, z_dsc, alpha=0.5, rstride=100, cstride=100, color = 'green')

        # Plot the unit LoS vectors
        plt3d.plot([0,unit_los_enu[0,0]],[0,unit_los_enu[1,0]], [0,unit_los_enu[2,0]], label='LoS asc', linewid
        plt3d.plot([0,unit_los_enu[0,1]],[0,unit_los_enu[1,1]], [0,unit_los_enu[2,1]], label='LoS desc', linew

        # Plot the 'projections' of the LoS vectors
        plt3d.plot([unit_los_enu[0,0],unit_los_enu[0,0]],[unit_los_enu[1,0],unit_los_enu[1,0]], [0,unit_los_en
                   linewidth=2, color = 'b', ls = '--', alpha = 0.6)
        plt3d.plot([0,unit_los_enu[0,0]],[0,unit_los_enu[1,0]], [0,0], linewidth=2, color = 'b',
                   ls = '--', alpha = 0.6)

        plt3d.plot([unit_los_enu[0,1],unit_los_enu[0,1]],[unit_los_enu[1,1],unit_los_enu[1,1]], [0,unit_los_en
                   linewidth=2, color = 'g', ls = '--', alpha = 0.6)
        plt3d.plot([0,unit_los_enu[0,1]],[0,unit_los_enu[1,1]], [0,0], linewidth=2, color = 'g',
                   ls = '--', alpha = 0.6)


        # Plot the intersection line
        plt3d.plot(x_line, y_line, z_line, lw=4, c='r', label = 'Null space')
        plt3d.plot(x_line, y_line, z_ground, lw=2, c='r', ls = '--', label = 'Ground projection', alpha = 0.7)


        #plt.legend(fontsize = 12)
        plt.ylabel('North', fontsize = 12)
        plt.xlabel('East', fontsize = 12)
        plt.title(r'$\phi$ = '+ str(np.around(phi, 2)) + ', $\zeta$ = ' + str(np.around(zeta, 2))
                  + ', $\psi$ = ' + str(np.around(psi, 2)), fontsize = 20)

        plt.show()


    return zeta, phi, psi
```

In [13]:
```
# Define AZD and incidence angle:
inc = np.deg2rad(np.array([35,35]))
alpha_d = np.deg2rad(np.array([253, 115]))

zeta, phi, psi = null_space_2sat(inc, alpha_d, 1)
```

```
[[-0.54851387  0.51983679]
 [-0.16769752 -0.24240388]
 [ 0.81915204  0.81915204]]
```

$\phi = 4.0$, $\zeta = 14.09$, $\psi = 74.46$




```
<ipython-input-5-3a552e58c29f>:90: MatplotlibDeprecationWarning: Calling gca() with keyword arguments was depre
cated in Matplotlib 3.4. Starting two minor releases later, gca() will take no keyword arguments. The gca() fun
ction should only be used to get the current axes, or if no axes exist, create new axes with default keyword ar
guments. To create a new axes with non-default arguments, use plt.axes() or plt.subplot().
  plt3d = plt.figure(figsize=(8,8)).gca(projection='3d')
```

In [7]:
```
phi_zeta(inc, alpha_d)
```

Out[7]:
```
(-0.7036884012627574, 16.878845812012585)
```

2.4 P1 - Stakeholder's perspective: Switzerland

Notebook to estimate the MDD in the transversal and normal direction (for every RUM in Switzerland)

In [ ]:

```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from pandas import read_csv
import pandas as pd
#from matplotlib import cm

from pathlib import Path
from math import *
from functions import *
from functions_strap_down_method import *

import scipy.stats

from matplotlib.patches import Ellipse
import matplotlib
#import tilemapbase

from matplotlib import pyplot as plt
import rasterio, rasterio.plot

import geopandas as gpd
import shapely
import contextily as ctx
from shapely.geometry import Point, Polygon
import shapely.speedups
```

In [2]:

```
def regression(time, y):
    """
    A function to determine to estimate the linear rates and standard deviations by different ways


    time = time matrix
    y = mean displacement array

    output:
        start = start value in mm
        vel = velocitiy in mm/year


    """
    Qyy = np.identity(len(y))

    A = np.ones((len(y),1))
    A = np.hstack((A, time))
    y_new = np.array([y]).T
    x_hat, Qx_hat = BLUE(A, y_new, Qyy)

    start = x_hat[0,0]
    vel = x_hat[1,0]*365*1000
    vel_reg = scipy.stats.linregress(time[:,0], y_new[:,0])[0]*365*1000
    std_vel = scipy.stats.linregress(time[:,0], y_new[:,0])[4]*365*1000
    std_ts = np.std(y_new[:,0] - time[:,0]*x_hat[1,0]+x_hat[0,0])*1000

    return start, vel, vel_reg, std_vel, std_ts
```

Load different RUMS

In [3]:

```
gpd.io.file.fiona.drvsupport.supported_drivers['KML'] = 'rw'
```

In [4]:

```
kml1 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Raintal
kml2 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Raintal
kml3 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Raintal
kml4 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Raintal
kml5 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Raintal
kml6 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Raintal
kml7 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Raintal
kml8 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Raintal
kml9 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Raintal
kml10 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml11 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml12 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml13 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml14 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml15 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml16 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml17 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml18 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml19 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml20 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml21 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml22 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml23 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml24 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml25 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml26 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml27 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml28 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml29 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml30 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml31 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml32 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml33 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml34 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml35 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
kml36 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rainta
```

In [5]:

```
s1 = gpd.read_file(kml1, driver='KML', crs='EPSG:4326')
s2 = gpd.read_file(kml2, driver='KML', crs='EPSG:4326')
s3 = gpd.read_file(kml3, driver='KML', crs='EPSG:4326')
s4 = gpd.read_file(kml4, driver='KML', crs='EPSG:4326')
s5 = gpd.read_file(kml5, driver='KML', crs='EPSG:4326')
s6 = gpd.read_file(kml6, driver='KML', crs='EPSG:4326')
s7 = gpd.read_file(kml7, driver='KML', crs='EPSG:4326')
s8 = gpd.read_file(kml8, driver='KML', crs='EPSG:4326')
s9 = gpd.read_file(kml9, driver='KML', crs='EPSG:4326')
s10 = gpd.read_file(kml10, driver='KML', crs='EPSG:4326')
s11 = gpd.read_file(kml11, driver='KML', crs='EPSG:4326')
s12 = gpd.read_file(kml12, driver='KML', crs='EPSG:4326')
s13 = gpd.read_file(kml13, driver='KML', crs='EPSG:4326')
s14 = gpd.read_file(kml14, driver='KML', crs='EPSG:4326')
s15 = gpd.read_file(kml15, driver='KML', crs='EPSG:4326')
s16 = gpd.read_file(kml16, driver='KML', crs='EPSG:4326')
s17 = gpd.read_file(kml17, driver='KML', crs='EPSG:4326')
s18 = gpd.read_file(kml18, driver='KML', crs='EPSG:4326')
s19 = gpd.read_file(kml19, driver='KML', crs='EPSG:4326')
s20 = gpd.read_file(kml20, driver='KML', crs='EPSG:4326')
s21 = gpd.read_file(kml21, driver='KML', crs='EPSG:4326')
s22 = gpd.read_file(kml22, driver='KML', crs='EPSG:4326')
s23 = gpd.read_file(kml23, driver='KML', crs='EPSG:4326')
s24 = gpd.read_file(kml24, driver='KML', crs='EPSG:4326')
s25 = gpd.read_file(kml25, driver='KML', crs='EPSG:4326')
s26 = gpd.read_file(kml26, driver='KML', crs='EPSG:4326')
s27 = gpd.read_file(kml27, driver='KML', crs='EPSG:4326')
s28 = gpd.read_file(kml28, driver='KML', crs='EPSG:4326')
s29 = gpd.read_file(kml29, driver='KML', crs='EPSG:4326')
s30 = gpd.read_file(kml30, driver='KML', crs='EPSG:4326')
s31 = gpd.read_file(kml31, driver='KML', crs='EPSG:4326')
s32 = gpd.read_file(kml32, driver='KML', crs='EPSG:4326')
s33 = gpd.read_file(kml33, driver='KML', crs='EPSG:4326')
s34 = gpd.read_file(kml34, driver='KML', crs='EPSG:4326')
s35 = gpd.read_file(kml35, driver='KML', crs='EPSG:4326')
s36 = gpd.read_file(kml36, driver='KML', crs='EPSG:4326')


s34.head()

#Create an array with all the different segments in in
```

```
C:\Programs\Anaconda3\envs\drama-3.9\lib\site-packages\geopandas\geodataframe.py:577: RuntimeWarning: Sequentia
l read of iterator was interrupted. Resetting iterator. This can negatively impact the performance.
  for feature in features_lst:
```

Out[5]:

	Name	Description	geometry
0	s34		POLYGON Z ((7.53711 46.89209 0.00000, 7.53798 ...

In [6]:

```
segments = [s1, s2, s3, s4, s5, s6, s7, s8, s9, s10,
            s11, s12, s13, s14, s15, s16, s17, s18, s19, s20,
            s21, s22, s23, s24, s25, s26, s27, s28, s29, s30,
            s31, s32, s33, s34, s35, s35]
```

Load satellite LoS data

In [7]:

```
# Location of the data:
loc_t88 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rair
loc_t15 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rair
loc_t139 = r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Rai
loc_t66= r"D:\Documents\Civiele_techniek\GRS\Afstuderen\Displacement Vector Decomposition\05 Programmeren\Raint



# Load in data of all satellites
#S1_asc_t88 = pd.read_csv(loc_t88, sep=',')
S1_asc_t15 = pd.read_csv(loc_t15, sep=',')
#S1_dsc_t139 = pd.read_csv(loc_t139, sep=',')
S1_dsc_t66 = pd.read_csv(loc_t66, sep=',')
```

In [8]:

```
display(S1_dsc_t66.head())
```

	pnt_id	pnt_flags	pnt_lat	pnt_lon	pnt_line	pnt_pixel	pnt_height	pnt_demheight	pnt_geoid	pnt_incidangle	...	d_202005
0	L00002646P00055214	11	46.934776	7.509778	2646	55214	618.54063	620.76953	48.71657	43.74449	...	-0.001
1	L00002646P00055215	11	46.934820	7.509786	2646	55215	615.77873	620.52644	48.71649	43.74440	...	0.000
2	L00002646P00055216	101	46.934932	7.509858	2646	55216	614.53142	620.28341	48.71626	43.74446	...	0.001
3	L00002646P00055217	11	46.934787	7.509681	2646	55217	615.79000	620.04033	48.71660	43.74476	...	0.000
4	L00002646P00055218	11	46.934779	7.509640	2646	55218	615.29068	619.79724	48.71664	43.74489	...	0.000

5 rows × 228 columns

In [9]:

```
# Compute time arrays per dataset:
t_asc = pd.to_datetime(tuple(map(lambda f: f[2:], filter(lambda f: f[0] == 'd', S1_asc_t15.columns))))
t_dsc = pd.to_datetime(tuple(map(lambda f: f[2:], filter(lambda f: f[0] == 'd', S1_dsc_t66.columns))))

#Compute time difference in days
days_asc = np.zeros(((len(t_asc)),1))
days_asc = np.array((t_asc - t_asc[0]).days)

days_dsc = np.zeros(((len(t_dsc)),1))
days_dsc = np.array((t_dsc - t_dsc[0]).days)
```

Export dataframes to GeoPandas DF

In [10]:

```
# From the pandas dataframe to a GeoPandas dataframe
asc_data_gdf = gpd.GeoDataFrame(S1_asc_t15, geometry=gpd.points_from_xy(S1_asc_t15.pnt_lon, S1_asc_t15.pnt_lat))
dsc_data_gdf = gpd.GeoDataFrame(S1_dsc_t66, geometry=gpd.points_from_xy(S1_dsc_t66.pnt_lon, S1_dsc_t66.pnt_lat))

asc_data_gdf = asc_data_gdf.set_crs(epsg=4326)
dsc_data_gdf = dsc_data_gdf.set_crs(epsg=4326)
```

In [11]:

```
#display(dsc_data_gdf.head())
```

Find the data for every RUM

In [12]:

```
orientation_segment = pd.read_csv('orientation_segments.csv', sep=';')
display(orientation_segment.head(5))

# Heading angles over the area. Provided by SkyGeo, assumed to be equal over the whole area
heading_t15 = 350.0055
heading_t139 = 190.4865

# Correction due to meridean convergence
alpha_d_asc = np.deg2rad(heading_t15-90+1)
alpha_d_dsc = np.deg2rad(heading_t139-90-2)
```

	segment	aspect	beta_slope	beta_old	slope_rad	nn	nn.1	slope_deg	beta_road	slope_road
0	1	51	321	-39	0.0	NaN	NaN	0.0	328	0
1	2	55	325	-35	0.0	NaN	NaN	0.0	328	0
2	3	49	319	-41	0.0	NaN	NaN	0.0	328	0
3	4	51	321	-39	0.0	NaN	NaN	0.0	328	0
4	5	53	323	-37	0.0	NaN	NaN	0.0	328	0

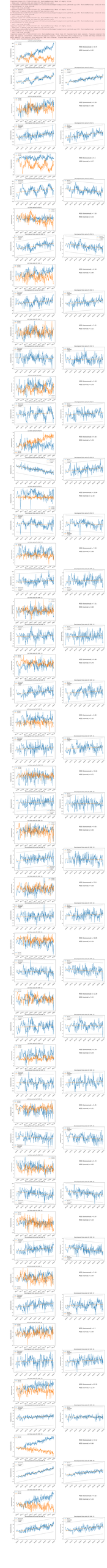
Loop trough all RUMS to estimate MDD based in the LoS data

In [14]:

```
segment_data = np.zeros((len(segments),22))
plot = 1
```

In [15]:

```
for i in range(len(segments)):
    nr = i+1

    # Extract the data per segment
    asc_segment = gpd.sjoin(asc_data_gdf, segments[i], how="inner", op='within')
    dsc_segment = gpd.sjoin(dsc_data_gdf, segments[i], how="inner", op='within')

    # Determine mean_inc, mean_coordinate, nr_obs per segment
    inc_asc = np.deg2rad(np.mean(asc_segment.pnt_incidangle))
    inc_dsc = np.deg2rad(np.mean(dsc_segment.pnt_incidangle))

    segment_data[i,6] = np.mean(asc_segment.pnt_lat)
    segment_data[i,7] = np.mean(asc_segment.pnt_lon)

    segment_data[i,4] = len(asc_segment)
    segment_data[i,5] = len(dsc_segment)

    # Determine orienation angles of the frame
    beta = np.deg2rad(orientation_segment.beta_slope[i] - 180)  ## -180 is needed since beta angles in excel fi
    gamma_t = np.deg2rad(orientation_segment.slope_deg[i])
    gamma_l = 0

    std_beta = np.deg2rad(7)
    std_gamma_t = np.deg2rad(5)
    std_gamma_l = np.deg2rad(3)

    # Compute mean time series over the segment
    space_time_asc = np.array(asc_segment.iloc[:, 18:18+len(t_asc)]) # Locations where the time series are stor
    space_time_dsc = np.array(dsc_segment.iloc[:, 18:18+len(t_dsc)])

    # Determine the mean value per epoch over the segment (complete time series) and get LoS data
    mean_asc = space_time_asc.mean(0)*np.cos(inc_asc)
    mean_dsc = space_time_dsc.mean(0)*np.cos(inc_asc)

    if len(asc_segment)> 0 and len(dsc_segment)> 0:

        # ----------- DECOMP PER EPOCH --------------------------##

        # Interpolate the asc values such that they have same dates as dsc
        mean_asc_inter = np.interp(days_dsc, days_asc,mean_asc )

        # Compute trans and normal value per epoch

        #first estimates trans and normal direction
        t0 = 0.01
        n0 = 0.01

        Qyy1 = np.zeros((5,5))
        Qyy1[0,0] = np.std(mean_asc_inter)**2
        Qyy1[1,1] = np.std(mean_dsc)**2
        Qyy1[2,2] = std_beta**2
        Qyy1[3,3] = std_gamma_t**2
        Qyy1[4,4] = std_gamma_l**2

        inc = np.array([inc_asc, inc_dsc])
        alpha_d = np.array([alpha_d_asc, alpha_d_dsc])

        transversal = np.zeros(len(t_dsc))
        normal = np.zeros(len(t_dsc))

        for j in range(len(t_dsc)):
            y_los = np.array([[mean_asc_inter[j]], [mean_dsc[j]]])
            #print (y_los)

            y1 = y_pseudo(y_los, beta, gamma_t, gamma_l)
            x01 = x0_TLN(t0, n0, beta, gamma_t, gamma_l)

            x, x_trans, x_normal, x_beta, x_gamma_t, x_gamma_l, Qx_hat = iteration_estimate_x(x01, y1, Qyy1, 10

            transversal[j] = x[0,0]
            normal[j] = x[1,0]

        ## ----------------- COMPUTE SIGMA AND VELOCITY PER DECOMP. TIME SERIES ----------------##

        T = np.zeros((len(days_dsc),1))
        T[:,0] = transversal
        N = np.zeros((len(days_dsc),1))
        N[:,0] = normal

        days = np.zeros((len(days_dsc),1))
        days[:,0] = days_dsc

        start_t, vel_t, vel_reg_t, std_vel_t, std_ts_t = regression(days, transversal)
        start_n, vel_n, vel_reg_n, std_vel_n, std_ts_n = regression(days, normal)

        ## ----------------- COMPUTE MDD PER SEGMENT -------------------------------------------##
        # Create the A matrix
        A = np.ones((len(t_dsc),2))
        A[:,1] = days_dsc

        # Create the QYY matrices
        Qyy_t = np.identity(len(t_dsc))*(std_ts_t/1000)**2
        Qyy_n = np.identity(len(t_dsc))*(std_ts_n/1000)**2

        # Compute the velocities and the start values
        x_hat_trans, Qxhat_trans = BLUE(A, T, Qyy_t)
        x_hat_normal, Qxhat_normal = BLUE(A, N, Qyy_n)

        B = np.matrix([[1], [-1]])
        L = 7.848860082330860

        std_dt = np.sqrt(Qx_hat[0,0])*1000

        Qt_trans, var_yp_trans = Qt(A[-1,:], Qxhat_trans, std_dt, B)
        MDD_trans = MDD_condition (L, Qt_trans, 1)

        std_dn = np.sqrt(Qx_hat[1,1])*1000
        Qt_normal, var_yp_normal = Qt(A[-1,:], Qxhat_normal, std_dn, B)
        MDD_normal = MDD_condition (L, Qt_normal, 1)

        ## ----------------- CREATE PLOTS -------------------------------------------##

        if plot == 1:
            fig = plt.figure(figsize=(18,10))
            plt.subplot(221)
            plt.plot(t_dsc, mean_asc_inter*1000,  '.-' , label = 'LoS asc')
            plt.plot(t_dsc, mean_dsc*1000,  '.-', label = 'LoS dsc')
            plt.xticks(rotation = 20)
            plt.title('LoS time series for RUM  ' + str(nr))
            plt.ylabel('Displacement [mm]')
            plt.legend()

            plt.subplot(223)
            plt.plot(t_dsc, transversal*1000,  '.-', label = 'Transversal')
            plt.plot(t_dsc, days_dsc*vel_t/(365*1000)*1000+ start_t*1000, label = 'Estimated')
            plt.plot(t_dsc, days_dsc*vel_t/(365*1000)*1000+ start_t*1000 + MDD_trans[0,0], label = '+/-MDD', co
            plt.plot(t_dsc, days_dsc*vel_t/(365*1000)*1000+ start_t*1000 - MDD_trans[0,0], color = 'grey', line
            plt.xticks(rotation = 40)
            plt.title('Decomposed time series for RUM  ' + str(nr))
            plt.ylabel('Displacement [mm]')
            plt.legend()

            plt.subplot(224)
            plt.plot(t_dsc, normal*1000,  '.-', label = 'Normal')
            plt.plot(t_dsc, days_dsc*vel_n/(365*1000)*1000+ start_n*1000, label = 'Estimated')
            plt.plot(t_dsc, days_dsc*vel_n/(365*1000)*1000+ start_n*1000 + MDD_normal[0,0], label = '+/-MDD', c
            plt.plot(t_dsc, days_dsc*vel_n/(365*1000)*1000+ start_n*1000 - MDD_normal[0,0], color = 'grey', lin
            plt.xticks(rotation = 40)
            plt.title('Decomposed time series for RUM  ' + str(nr))
            plt.ylabel('Displacement [mm]')
            plt.legend()

            plt.subplot(222)
            plt.text(0.75, 1.5, 'MDD (transversal) = ' + str(np.around(MDD_trans[0,0],2)), fontsize = 15)
            plt.text(0.75, 1.2, 'MDD (normal) = ' + str(np.around(MDD_normal[0,0],2)), fontsize = 15)
            plt.ylim([0,2])
            plt.xlim([0,2])
            plt.axis('off')

            fig.savefig("RUM "+ str(nr) +".png")


        ## ----------------- SAVE DATA TO FILE -------------------------------------------##
        segment_data[i,0] = nr
        segment_data[i,1] = np.rad2deg(beta)
        segment_data[i,2] = np.rad2deg(gamma_t)
        segment_data[i,3] = np.rad2deg(gamma_l)
        segment_data[i,8] = x_hat_trans[1]*365*1000
        segment_data[i,9] = np.sqrt(Qxhat_trans[1,1])*365*1000
        segment_data[i,10] = x_hat_normal[1]*365*1000
        segment_data[i,11] = np.sqrt(Qxhat_normal[1,1])*365*1000
        segment_data[i,12] = MDD_trans[0,0]
        segment_data[i,13] = MDD_normal[0,0]
        segment_data[i,14] = np.rad2deg(inc_asc)
        segment_data[i,15] = np.rad2deg(inc_dsc)
        segment_data[i,16] = np.rad2deg(alpha_d_asc)
        segment_data[i,17] = np.rad2deg(alpha_d_dsc)
        segment_data[i,18] = (np.std(mean_asc_inter)*1000)
        segment_data[i,19] = (np.std(mean_dsc)*1000)
        segment_data[i,20] = (std_ts_t)
        segment_data[i,21] = (std_ts_t)
```


MSE (horizontal) = 32.71
MSE (vertical) = 6.63
MSE (horizontal) = 0.49
MSE (vertical) = 0.08
MSE (horizontal) = 8.1
MSE (vertical) = 4.27
MSE (horizontal) = 1.83
MSE (vertical) = 1.64
MSE (horizontal) = 0.39
MSE (vertical) = 0.09
MSE (horizontal) = 0.45
MSE (vertical) = 0.08
MSE (horizontal) = 0.45
MSE (vertical) = 0.22
MSE (horizontal) = 4.29
MSE (vertical) = 0.75
MSE (horizontal) = 0.24
MSE (vertical) = 0.08
MSE (horizontal) = 22.65
MSE (vertical) = 12.14
MSE (horizontal) = 1.63
MSE (vertical) = 0.48
MSE (horizontal) = 7.72
MSE (vertical) = 4.48
MSE (horizontal) = 0.86
MSE (vertical) = 0.16
MSE (horizontal) = 0.89
MSE (vertical) = 0.15
MSE (horizontal) = 10.44
MSE (vertical) = 5.71
MSE (horizontal) = 0.65
MSE (vertical) = 0.26
MSE (horizontal) = 0.13
MSE (vertical) = 0.05
MSE (horizontal) = 20.45
MSE (vertical) = 5.79
MSE (horizontal) = 11.18
MSE (vertical) = 4.63
MSE (horizontal) = 0.29
MSE (vertical) = 0.09
MSE (horizontal) = 0.35
MSE (vertical) = 0.03
MSE (horizontal) = 4.97
MSE (vertical) = 3.16
MSE (horizontal) = 0.29
MSE (vertical) = 0.04
MSE (horizontal) = 4.2
MSE (vertical) = 2.09
MSE (horizontal) = 32.19
MSE (vertical) = 23.77
MSE (horizontal) = 11.12
MSE (vertical) = 6.34
MSE (horizontal) = 9.82
MSE (vertical) = 1.74

In [16]:

```
c = ['Segment', 'beta', 'gamma_t', 'gamma_l', 'nr_asc', 'nr_dsc', 'pnt_lat', 'pnt_lon', 'trans_vel [mm/yr]', 's
    'normal_vel [mm/yr]', 'std_normal_vel', 'MDD_trans [mm]', 'MDD_normal [mm]', 'inc_asc', 'inc_dsc', 'alpha_d
    'std_asc_TS [mm]', 'std_dsc_TS [mm]', 'std_trans_TS [mm]', 'std_normal_TS [mm]']

DATA = pd.DataFrame(data=segment_data, columns=c)
```

In [17]:

```
display(DATA)
```

	Segment	beta	gamma_t	gamma_l	nr_asc	nr_dsc	pnt_lat	pnt_lon	trans_vel [mm/yr]	std_trans_vel	...	MDD_trans [mm]	MDD_normal [mm]	inc_asc
0	1.0	141.0	0.000000	0.0	54.0	74.0	46.922287	7.505738	2.719714	0.155885	...	10.709217	6.814695	30.876988
1	2.0	145.0	0.000000	0.0	27.0	98.0	46.921237	7.506818	0.852715	0.116844	...	6.276108	3.884213	30.881230
2	3.0	139.0	0.000000	0.0	52.0	85.0	46.920430	7.507628	0.915307	0.185976	...	8.103751	4.265831	30.884233
3	4.0	141.0	0.000000	0.0	37.0	50.0	46.919561	7.508510	1.515831	0.151563	...	7.933716	4.241379	30.887624
4	5.0	143.0	0.000000	0.0	37.0	74.0	46.918479	7.509530	0.539919	0.132004	...	6.176441	3.891361	30.891326
5	6.0	143.0	0.000000	0.0	26.0	94.0	46.917697	7.510179	-0.103330	0.109468	...	5.407515	3.216263	30.893735
6	7.0	143.0	14.036243	0.0	19.0	36.0	46.916297	7.511514	0.354606	0.217477	...	9.180984	6.787750	30.898553
7	8.0	146.0	14.036243	0.0	9.0	50.0	46.915982	7.511682	-2.277859	0.111353	...	9.155232	5.293866	30.898669
8	9.0	144.0	18.434949	0.0	1.0	114.0	46.913321	7.514116	-0.439006	0.228182	...	10.992032	12.326181	30.907240
9	0.0	0.0	0.000000	0.0	0.0	89.0	NaN	NaN	0.000000	0.000000	...	0.000000	0.000000	0.000000
10	11.0	133.0	18.245855	0.0	10.0	142.0	46.911521	7.516091	0.414890	0.158694	...	7.829538	5.994647	30.915081
11	12.0	129.0	18.434949	0.0	41.0	142.0	46.910810	7.517182	0.215832	0.163417	...	7.512918	4.482933	30.919493
12	13.0	131.0	23.051301	0.0	25.0	120.0	46.910105	7.518292	0.956614	0.187469	...	8.883967	6.780659	30.924003
13	0.0	0.0	0.000000	0.0	0.0	49.0	NaN	NaN	0.000000	0.000000	...	0.000000	0.000000	0.000000
14	15.0	147.0	13.240520	0.0	6.0	63.0	46.908321	7.520113	0.305527	0.157513	...	6.888824	5.551142	30.930515
15	0.0	0.0	0.000000	0.0	0.0	34.0	NaN	NaN	0.000000	0.000000	...	0.000000	0.000000	0.000000
16	0.0	0.0	0.000000	0.0	0.0	78.0	NaN	NaN	0.000000	0.000000	...	0.000000	0.000000	0.000000
17	18.0	132.0	19.025606	0.0	6.0	50.0	46.905494	7.523515	0.480553	0.246806	...	10.443449	8.713493	30.944153
18	19.0	129.0	4.467159	0.0	34.0	55.0	46.905184	7.524284	1.015556	0.213551	...	9.647922	4.262502	30.947174
19	20.0	115.0	4.085617	0.0	43.0	89.0	46.904590	7.525364	0.469162	0.333638	...	14.996076	4.577030	30.951808
20	21.0	115.0	14.036243	0.0	17.0	89.0	46.903999	7.527138	-0.616746	0.338675	...	16.854226	6.541939	30.960354
21	22.0	119.0	19.230672	0.0	46.0	81.0	46.903547	7.528502	0.487590	0.303653	...	12.284671	5.506111	30.966292
22	23.0	133.0	14.743563	0.0	15.0	94.0	46.903129	7.529663	-0.115751	0.200365	...	8.775110	6.589810	30.971868
23	24.0	138.0	15.321510	0.0	51.0	91.0	46.901909	7.531129	-0.454755	0.137603	...	6.446527	4.608521	30.977626
24	25.0	152.0	3.179830	0.0	21.0	65.0	46.900766	7.532144	-0.754585	0.144978	...	6.720127	4.649488	30.981087
25	26.0	165.0	3.179830	0.0	24.0	18.0	46.899667	7.532799	0.865942	0.181036	...	8.972837	5.543369	30.982767
26	27.0	195.0	3.715289	0.0	48.0	48.0	46.898647	7.533103	0.608267	0.115308	...	5.235506	3.643915	30.982808
27	28.0	170.0	4.537773	0.0	47.0	64.0	46.897609	7.533410	0.127534	0.097692	...	4.102758	3.089681	30.982848
28	29.0	100.0	0.000000	0.0	43.0	53.0	46.896295	7.533964	4.529235	0.652028	...	83.351135	12.772201	30.983687
29	30.0	150.0	0.000000	0.0	41.0	71.0	46.895357	7.534427	4.177397	0.094580	...	11.119027	8.460883	30.984693
30	31.0	135.0	0.000000	0.0	43.0	48.0	46.894311	7.535306	2.309246	0.142026	...	9.418006	5.343502	30.987692
31	32.0	144.0	0.000000	0.0	63.0	49.0	46.893532	7.536152	-0.650660	0.151790	...	6.738647	3.822832	30.990988
32	33.0	150.0	0.000000	0.0	57.0	78.0	46.892640	7.536930	0.892161	0.111424	...	5.970807	3.786135	30.993630
33	34.0	150.0	0.000000	0.0	58.0	64.0	46.891773	7.537947	0.366279	0.111003	...	5.095326	3.405928	30.997591
34	35.0	150.0	0.000000	0.0	83.0	74.0	46.891012	7.538715	0.413740	0.104425	...	4.903910	3.275282	31.000606
35	36.0	147.0	10.124672	0.0	83.0	74.0	46.891012	7.538715	0.383844	0.109505	...	5.130468	3.260936	31.000606

36 rows × 22 columns

In [18]:

```
DATA.to_csv("segment_data.csv")
```

In [19]:

```
DATA.iloc[25,:]
```

Out[19]:

```
Segment                26.000000
beta                  165.000000
gamma_t                 3.179830
gamma_l                 0.000000
nr_asc                 24.000000
nr_dsc                 18.000000
pnt_lat                46.899667
pnt_lon                 7.532799
trans_vel [mm/yr]       0.865942
std_trans_vel           0.181036
normal_vel [mm/yr]     -0.657107
std_normal_vel          0.119108
MDD_trans [mm]          8.972837
MDD_normal [mm]         5.543369
inc_asc                30.982767
inc_dsc                43.616482
alpha_d_asc           261.005500
alpha_d_dsc            98.486500
std_asc_TS [mm]         1.514651
std_dsc_TS [mm]         3.051023
std_trans_TS [mm]       2.873512
std_normal_TS [mm]      2.873512
Name: 25, dtype: float64
```

In [ ]:

2.5 P2 - User of an existing InSAR product - Decomposition

Notebook to estimate the displacement rates in the transversal and normal direction for Veendam

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
from pandas import read_csv
import pandas as pd
from matplotlib import cm

from pathlib import Path
from math import *
from functions import *
from functions_perspective_1 import *

import scipy.stats
```

In [2]:
```
def iteration_estimate_x(x0, y, Qyy, epsilon, inc, alpha_d):
    """
    A function to estimate x based on an initial guess and the linearized method

    Input:
        1. x0: initial guess vector: [dt, dn, beta, gamma_t, gamma_l]
    """

    # Initial guess
    x = x0

    # max number of iterations
    N = 1000
    x_trans = np.zeros(N)
    x_normal = np.zeros(N)
    x_beta = np.zeros(N)
    x_gamma_t = np.zeros(N)
    x_gamma_l = np.zeros(N)

    for i in range(N):

        xg = np.squeeze(np.asarray(x))

        x_trans[i] = xg[0]
        x_normal[i] = xg[1]
        x_beta[i] = xg[2]
        x_gamma_t[i] = xg[3]
        x_gamma_l[i] = xg[4]

        # Observations for initial guess
        y_x0 = TLN_forward_model(inc, alpha_d, xg[2], xg[3], xg[4], xg[0], xg[1])

        delta_y = y - y_x0

        # fill in Jacobian (with with the values for x)
        jacobian = TLN_jacobian(inc, alpha_d, xg[2], xg[3], xg[4], xg[0], xg[1])

        # BLUE for the Jacobian and delta_y
        d_xhat, Qx_hat = BLUE(jacobian, delta_y, Qyy)


        x = x+d_xhat

        if np.sqrt(d_xhat[0]**2+d_xhat[1]**2+d_xhat[2]**2+d_xhat[3]**2+d_xhat[4]**2)<=epsilon:
            #print ('The number of iterations was', i)
            break


    x_trans = x_trans[0:i+1]
    x_normal = x_normal[0:i+1]
    x_beta = x_beta[0:i+1]
    x_gamma_t = x_gamma_t[0:i+1]
    x_gamma_l = x_gamma_l[0:i+1]

    return x, x_trans, x_normal, x_beta, x_gamma_t, x_gamma_l, Qx_hat
```

In [3]:
```
def y_pseudo(y_los, beta_pseudo, gamma_t_pseudo, gamma_l_pseudo):
    """
    Function to compute the y vector with pseudo observations for the angles


    """

    y = np.zeros((len(y_los)+3, 1))
    y[0:len(y_los)] = y_los
    y[-1] = gamma_l_pseudo
    y[-2] = gamma_t_pseudo
    y[-3] = beta_pseudo

    return y
```

In [4]:
```
def x0_TLN(dt0, dn0, beta0, gamma_t0, gamma_l0):
    """
    Function to compute the initial guess vector for x
    """

    x = np.matrix([[dt0], [dn0], [beta0], [gamma_t0], [gamma_l0]])

    return x
```

Load the data

In [5]:
```
period1 = pd.read_csv('period1.csv',sep=';')
period2 = pd.read_csv('period2.csv',sep=';')
period3 = pd.read_csv('period3.csv',sep=';')

display(period1.head(5))
```

	Unnamed: 0	Segment	start_asc	vel_asc	vel_asc_regr	std_asc_rate	std_asc_ts	start_dsc	vel_dsc	vel_dsc_regr	std_dsc_rate	std_dsc_ts
0	0	11	-0.001899	-9.647044	-9.647044	0.582410	3.262513	-0.002317	-15.127512	-15.127512	0.480442	2.778466
1	1	12	-0.000402	-9.405862	-9.405862	0.546190	3.059619	-0.003581	-14.043741	-14.043741	0.565478	3.270238
2	2	13	-0.001898	-10.743008	-10.743008	0.591976	3.316099	-0.004180	-13.690156	-13.690156	0.619166	3.580721
3	3	14	-0.013690	-9.710099	-9.710099	1.059777	5.936600	NaN	NaN	NaN	NaN	NaN
4	4	15	0.000376	-8.054822	-8.054822	0.495916	2.777994	-0.005606	-18.667913	-18.667913	0.533987	3.088120

In [6]:
```
geometry = pd.read_csv('beta en inc angle.csv',sep=';')
display(geometry.head(5))
```

	Unnamed: 0	Segment	beta	oost1_t139	oost2_t15
0	0	11	105	44.1903	36.2944
1	1	12	135	44.1903	36.2944
2	2	13	165	44.1903	36.2944
3	3	14	195	44.1903	36.2944
4	4	15	225	44.1903	36.2944

Define viewing geometry and TLN orientation per RUM

In [7]:
```
# Segements
s = np.array(period1.Segment)

# TLN frame orientation
beta = np.array(geometry.beta)*2*np.pi/360
gamma_t = np.zeros(len(s))
gamma_l = np.zeros(len(s))

# Viewing geometry
inc_dsc = np.array(geometry.oost1_t139)*2*np.pi/360
inc_asc = np.array(geometry.oost2_t15)*2*np.pi/360

heading_asc = np.ones(len(s))*350+1
heading_dsc = np.ones(len(s))*190-2

alpha_d_asc = (heading_asc-90)*2*np.pi/360
alpha_d_dsc = (heading_dsc-90)*2*np.pi/360
```

In [8]:
```
# Uentainty for the viewing geometry
std_beta_deg = 10
std_gamma_t_deg = 2
std_gamma_l_deg = 2

std_beta = std_beta_deg*2*np.pi/360
std_gamma_t = std_gamma_t_deg*2*np.pi/360
std_gamma_l = std_gamma_l_deg*2*np.pi/360
```

Convert projections to LoS displacements

In [9]:
```
proj_asc1 = np.array(period1.vel_asc)
proj_asc2 = np.array(period2.vel_asc)
proj_asc3 = np.array(period3.vel_asc)

proj_dsc1 = np.array(period1.vel_dsc)
proj_dsc2 = np.array(period2.vel_dsc)
proj_dsc3 = np.array(period3.vel_dsc)

vel_asc1 = proj_asc1 * np.cos(inc_asc)
vel_asc2 = proj_asc2 * np.cos(inc_asc)
vel_asc3 = proj_asc3 * np.cos(inc_asc)

vel_dsc1 = proj_dsc1 * np.cos(inc_dsc)
vel_dsc2 = proj_dsc2 * np.cos(inc_dsc)
vel_dsc3 = proj_dsc3 * np.cos(inc_dsc)
```

In [10]:
```
proj_asc1_std = np.array(period1.std_asc_rate)
proj_asc2_std = np.array(period2.std_asc_rate)
proj_asc3_std = np.array(period3.std_asc_rate)

proj_dsc1_std = np.array(period1.std_dsc_rate)
proj_dsc2_std = np.array(period2.std_dsc_rate)
proj_dsc3_std = np.array(period3.std_dsc_rate)

vel_asc1_std = proj_asc1_std * np.cos(inc_asc)
vel_asc2_std = proj_asc2_std * np.cos(inc_asc)
vel_asc3_std = proj_asc3_std * np.cos(inc_asc)

vel_dsc1_std = proj_dsc1_std * np.cos(inc_dsc)
vel_dsc2_std = proj_dsc2_std * np.cos(inc_dsc)
vel_dsc3_std = proj_dsc3_std * np.cos(inc_dsc)
```

Test the decomposition for 1 RUM

In [11]:
```
i = 11 #i = 14 --> segment nr 24

dt0 = 5
dn0 = 10

y_los1 = np.matrix([[vel_asc1[i]], [vel_dsc1[i]]])

y1 = y_pseudo(y_los1, beta[i], gamma_t[i], gamma_l[i])
x01 = x0_TLN(dt0, dn0, beta[i], gamma_t[i], gamma_l[i])

Qyy1 = np.zeros((5,5))
Qyy1[0,0] = vel_asc1_std[i]**2
Qyy1[1,1] = vel_dsc1_std[i]**2
Qyy1[2,2] = std_beta**2
Qyy1[3,3] = std_gamma_t**2
Qyy1[4,4] = std_gamma_l**2

inc = np.array([inc_asc[i], inc_dsc[i]])
alpha_d = np.array([alpha_d_asc[i], alpha_d_dsc[i]])


x, x_trans, x_normal, x_beta, x_gamma_t, x_gamma_l, Qx_hat = iteration_estimate_x(x01, y1, Qyy1, 10**(-5), inc,

x_trans1 = x[0,0]

print (Qx_hat)
```

```
[[ 1.14273397e+02 -1.44096126e+01 -1.84085739e+00  8.74328868e-03
   3.42436263e-02]
 [-1.44096126e+01  2.23033360e+00  2.35170884e-01  1.88007514e-02
  -4.37464842e-03]
 [-1.84085739e+00  2.35170884e-01  3.04617420e-02 -1.13120822e-17
   1.12447315e-17]
 [ 8.74328868e-03  1.88007514e-02 -9.21582023e-18  1.21846968e-03
   2.77023813e-20]
 [ 3.42436263e-02 -4.37464842e-03  1.79637997e-18  1.25906089e-19
   1.21846968e-03]]
```

Estimate displacements for all RUMs

In [13]:
```
result_period1 = np.zeros((len(s), 5))
result_period2 = np.zeros((len(s), 5))
result_period3 = np.zeros((len(s), 5))

dt0 = 5
dn0 = 10

for i in range (len(s)):

    #Determine LoS observations per segment for the three time periods
    y_los1 = np.matrix([[vel_asc1[i]], [vel_dsc1[i]]])
    y_los2 = np.matrix([[vel_asc2[i]], [vel_dsc2[i]]])
    y_los3 = np.matrix([[vel_asc3[i]], [vel_dsc3[i]]])


    # Add pseudo observations to the y vector (angles for beta, gamma_t and gamma_l)
    y1 = y_pseudo(y_los1, beta[i], gamma_t[i], gamma_l[i])
    y2 = y_pseudo(y_los2, beta[i], gamma_t[i], gamma_l[i])
    y3 = y_pseudo(y_los3, beta[i], gamma_t[i], gamma_l[i])


    # Come up with first estimate for the unknown parameters x
    x0 = x0_TLN(dt0, dn0, beta[i], gamma_t[i], gamma_l[i])

    # Define Qyy matrix for the three time periods
    Qyy1 = np.zeros((5,5))
    Qyy1[0,0] = vel_asc1_std[i]**2
    Qyy1[1,1] = vel_dsc1_std[i]**2
    Qyy1[2,2] = std_beta**2
    Qyy1[3,3] = std_gamma_t**2
    Qyy1[4,4] = std_gamma_l**2

    Qyy2 = np.zeros((5,5))
    Qyy2[0,0] = vel_asc2_std[i]**2
    Qyy2[1,1] = vel_dsc2_std[i]**2
    Qyy2[2,2] = std_beta**2
    Qyy2[3,3] = std_gamma_t**2
    Qyy2[4,4] = std_gamma_l**2

    Qyy3 = np.zeros((5,5))
    Qyy3[0,0] = vel_asc3_std[i]**2
    Qyy3[1,1] = vel_dsc3_std[i]**2
    Qyy3[2,2] = std_beta**2
    Qyy3[3,3] = std_gamma_t**2
    Qyy3[4,4] = std_gamma_l**2

    # define incidence and alpha_d array for the segment
    inc = np.array([inc_asc[i], inc_dsc[i]])
    alpha_d = np.array([alpha_d_asc[i], alpha_d_dsc[i]])

    # Compute the estimates for the unknown parameters
    x1, x_trans, x_normal, x_beta, x_gamma_t, x_gamma_l, Qx_hat1 = iteration_estimate_x(x0, y1, Qyy1, 10**(-5),
    x2, x_trans, x_normal, x_beta, x_gamma_t, x_gamma_l, Qx_hat2 = iteration_estimate_x(x0, y2, Qyy2, 10**(-5),
    x3, x_trans, x_normal, x_beta, x_gamma_t, x_gamma_l, Qx_hat3 = iteration_estimate_x(x0, y3, Qyy3, 10**(-5),

    # Fill je result matrices with the final results

    result_period1[i,0] = s[i] #segment number
    result_period2[i,0] = s[i]
    result_period3[i,0] = s[i]

    result_period1[i,1] = x1[0,0] #transversal estimate
    result_period2[i,1] = x2[0,0]
    result_period3[i,1] = x3[0,0]

    result_period1[i,2] = np.sqrt(Qx_hat1[0,0]) #transversal standard deviation
    result_period2[i,2] = np.sqrt(Qx_hat2[0,0])
    result_period3[i,2] = np.sqrt(Qx_hat3[0,0])

    result_period1[i,3] = x1[1,0] #normal estimate
    result_period2[i,3] = x2[1,0]
    result_period3[i,3] = x3[1,0]

    result_period1[i,4] = np.sqrt(Qx_hat1[1,1]) #normal standard deviation
    result_period2[i,4] = np.sqrt(Qx_hat2[1,1])
    result_period3[i,4] = np.sqrt(Qx_hat3[1,1])
```

remove nan values

In [14]:
```
result_period1 = np.nan_to_num(result_period1)
result_period2 = np.nan_to_num(result_period2)
result_period3 = np.nan_to_num(result_period3)
```

In [15]:
```
coordinates = pd.read_csv('coordinaten_segmenten.csv',sep=',')
display(coordinates.head())
```

	fid	id	layer	path	xcoord	ycoord
0	1	27	split_ring1	D:/Documents/Civiele_techniek/GRS/Afstuderen/D...	251357.156318	569644.286798
1	2	21	split_ring1	D:/Documents/Civiele_techniek/GRS/Afstuderen/D...	252370.386705	572987.865455
2	3	211	split_ring1	D:/Documents/Civiele_techniek/GRS/Afstuderen/D...	250601.018969	572516.426808
3	4	26	split_ring1	D:/Documents/Civiele_techniek/GRS/Afstuderen/D...	252388.626839	569646.954892
4	5	25	split_ring1	D:/Documents/Civiele_techniek/GRS/Afstuderen/D...	253212.158127	570180.938953

In [16]:
```
rd_x = np.zeros(len(s))
rd_y = np.zeros(len(s))

for i in range (len(s)):
    rd_x[i] = coordinates.xcoord[coordinates.id==s[i]]
    rd_y[i] = coordinates.ycoord[coordinates.id==s[i]]
```

In [17]:
```
print (np.min(result_period3[:,1]))
print (np.max(result_period3[:,1]))
```

```
-9.079086515065374
44.919064870460346
```

Remove non value RUMs

In [18]:
```
#print (np.around(result_period1),2)
```

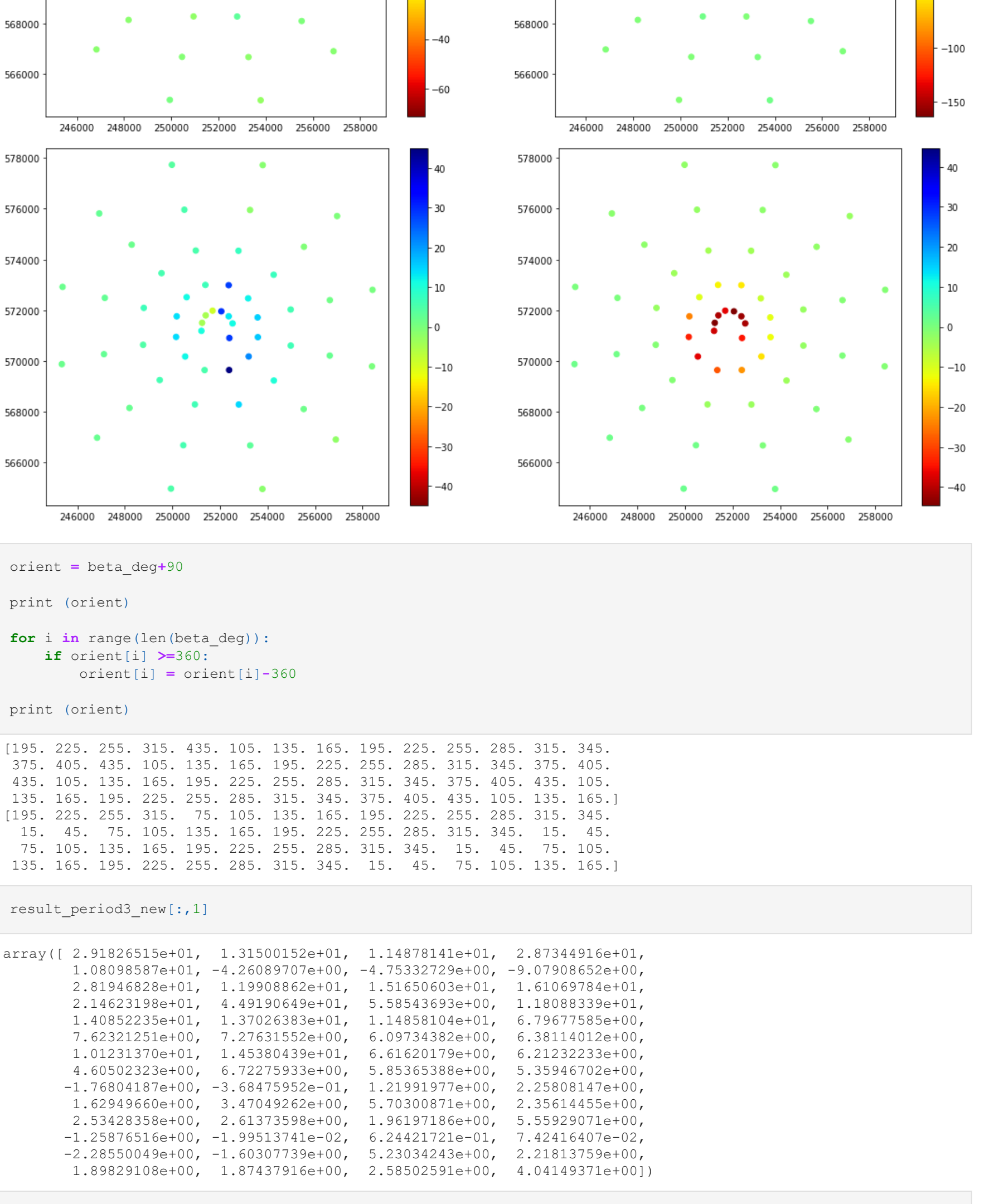
In [19]:
```
result_period1_new = np.delete(result_period1,[3,5,6],0)
result_period2_new = np.delete(result_period2,[3,5,6],0)
result_period3_new = np.delete(result_period3,[3,5,6],0)

rd_x_new = np.delete(rd_x,[3,5,6])
rd_y_new = np.delete(rd_y,[3,5,6])

beta_d = np.zeros((len(beta),1))
beta_d[:,0] = beta*360/(2*np.pi)
beta_deg = np.delete(beta_d, [3,5,6])
```

In [20]:
```
print (len(beta_deg))
```

```
56
```

In [21]:
```
c_max = -np.min(result_period3[:,1])
c_min = np.min(result_period3[:,1])

plt.figure(figsize=(18,7))
plt.subplot(121)
cm = plt.cm.get_cmap('jet_r')
sc = plt.scatter(rd_x_new, rd_y_new, c=result_period1_new[:,1],
                 vmin=-np.max(result_period1_new[:,1]), vmax=np.max(result_period1_new[:,1]), s=35, cmap=cm)
plt.colorbar(sc)

plt.subplot(122)
cm = plt.cm.get_cmap('jet_r')
sc = plt.scatter(rd_x_new, rd_y_new, c=result_period1_new[:,3],
                 vmin=np.min(result_period1_new[:,3]), vmax=-np.min(result_period1_new[:,3]), s=35, cmap=cm)
plt.colorbar(sc)

plt.figure(figsize=(18,7))
plt.subplot(121)
cm = plt.cm.get_cmap('jet_r')
sc = plt.scatter(rd_x_new, rd_y_new, c=result_period2_new[:,1],
                 vmin=-np.max(result_period2_new[:,1]), vmax=np.max(result_period2_new[:,1]), s=35, cmap=cm)
plt.colorbar(sc)

plt.subplot(122)
cm = plt.cm.get_cmap('jet_r')
sc = plt.scatter(rd_x_new, rd_y_new, c=result_period2_new[:,3],
                 vmin=np.min(result_period2_new[:,3]), vmax=-np.min(result_period2_new[:,3]), s=35, cmap=cm)
plt.colorbar(sc)

plt.figure(figsize=(18,7))
plt.subplot(121)
cm = plt.cm.get_cmap('jet_r')
sc = plt.scatter(rd_x_new, rd_y_new, c=result_period3_new[:,1],
                 vmin=-np.max(result_period3_new[:,1]), vmax=np.max(result_period3_new[:,1]), s=35, cmap=cm)
plt.colorbar(sc)

plt.subplot(122)
cm = plt.cm.get_cmap('jet_r')
sc = plt.scatter(rd_x_new, rd_y_new, c=result_period3_new[:,3],
                 vmin=np.min(result_period3_new[:,3]), vmax=-np.min(result_period3_new[:,3]), s=35, cmap=cm)
plt.colorbar(sc)
```

Out[21]: <matplotlib.colorbar.Colorbar at 0x256071264c0>



In [22]:
```
orient = beta_deg+90

print (orient)

for i in range(len(beta_deg)):
    if orient[i] >=360:
        orient[i] = orient[i]-360

print (orient)
```

```
[195. 225. 255. 315. 435. 105. 135. 165. 195. 225. 255. 285. 315. 345.
 375. 405. 435. 105. 135. 165. 195. 225. 255. 285. 315. 345. 375. 405.
 435. 105. 135. 165. 195. 225. 255. 285. 315. 345. 375. 405. 435. 105.
 135. 165. 195. 225. 255. 285. 315. 345. 375. 405. 435. 105. 135. 165.]
[195. 225. 255. 315.  75. 105. 135. 165. 195. 225. 255. 285. 315. 345.
  15.  45.  75. 105. 135. 165. 195. 225. 255. 285. 315. 345.  15.  45.
  75. 105. 135. 165. 195. 225. 255. 285. 315. 345.  15.  45.  75. 105.
 135. 165. 195. 225. 255. 285. 315. 345.  15.  45.  75. 105. 135. 165.]
```

In [23]:
```
result_period3_new[:,1]
```

Out[23]:
```
array([ 2.91826515e+01,  1.31500152e+01,  1.14878141e+01,  2.87344916e+01,
        1.08098587e+01, -4.26089707e+00, -4.75332729e+00, -9.07908652e+00,
        2.81946828e+01,  1.19908862e+01,  1.51650603e+01,  1.61069784e+01,
        2.14623198e+01,  4.49190649e+01,  5.58543693e+00,  1.18088339e+01,
        1.40852235e+01,  1.37026383e+01,  1.14858104e+01,  6.79677585e+00,
        7.62321251e+00,  7.27631552e+00,  6.09734382e+00,  6.38114012e+00,
        1.01231370e+01,  1.45380439e+01,  6.61620179e+00,  6.21232233e+00,
        4.60502323e+00,  6.72275933e+00,  5.85365388e+00,  5.35946702e+00,
       -1.76804187e+00, -3.68475952e-01,  1.21991977e+00,  2.25808147e+00,
        1.62949660e+00,  3.47049262e+00,  5.70300871e+00,  2.35614455e+00,
        2.53428358e+00,  2.61373598e+00,  1.96197186e+00,  5.55929071e+00,
       -1.25876516e+00, -1.99513741e-02,  6.24421721e-01,  7.42416407e-02,
       -2.28550049e+00, -1.60307739e+00,  5.23034243e+00,  2.21813759e+00,
        1.89829108e+00,  1.87437916e+00,  2.58502591e+00,  4.04149371e+00])
```

In [24]:
```
orient
```

Out[24]:
```
array([195., 225., 255., 315.,  75., 105., 135., 165., 195., 225., 255.,
       285., 315., 345.,  15.,  45.,  75., 105., 135., 165., 195., 225.,
       255., 285., 315., 345.,  15.,  45.,  75., 105., 135., 165., 195.,
       225., 255., 285., 315., 345.,  15.,  45.,  75., 105., 135., 165.,
       195., 225., 255., 285., 315., 345.,  15.,  45.,  75., 105., 135.,
       165.])
```

In [25]:
```
result_period3_new[:,1]
```

Out[25]:
```
array([ 2.91826515e+01,  1.31500152e+01,  1.14878141e+01,  2.87344916e+01,
        1.08098587e+01, -4.26089707e+00, -4.75332729e+00, -9.07908652e+00,
        2.81946828e+01,  1.19908862e+01,  1.51650603e+01,  1.61069784e+01,
        2.14623198e+01,  4.49190649e+01,  5.58543693e+00,  1.18088339e+01,
        1.40852235e+01,  1.37026383e+01,  1.14858104e+01,  6.79677585e+00,
        7.62321251e+00,  7.27631552e+00,  6.09734382e+00,  6.38114012e+00,
        1.01231370e+01,  1.45380439e+01,  6.61620179e+00,  6.21232233e+00,
        4.60502323e+00,  6.72275933e+00,  5.85365388e+00,  5.35946702e+00,
       -1.76804187e+00, -3.68475952e-01,  1.21991977e+00,  2.25808147e+00,
        1.62949660e+00,  3.47049262e+00,  5.70300871e+00,  2.35614455e+00,
        2.53428358e+00,  2.61373598e+00,  1.96197186e+00,  5.55929071e+00,
       -1.25876516e+00, -1.99513741e-02,  6.24421721e-01,  7.42416407e-02,
       -2.28550049e+00, -1.60307739e+00,  5.23034243e+00,  2.21813759e+00,
        1.89829108e+00,  1.87437916e+00,  2.58502591e+00,  4.04149371e+00])
```

In [26]:
```
# Adding coordinates for the std circles
x_circle = rd_x_new + np.sin(np.deg2rad(orient))*result_period3_new[:,1]*20
y_circle = rd_y_new + np.cos(np.deg2rad(orient))*result_period3_new[:,1]*20
```

In [27]:
```
result_period1_new[:,2]
```

Out[27]:
```
array([ 9.36860268,  1.13288099,  0.69076385,  1.71404239,  0.69118328,
        0.80055669,  0.86569983,  5.43809255, 10.68987358,  0.9467276 ,
        0.50726172,  0.38592627,  1.08168467,  7.8936481 ,  2.945537  ,
        1.07167866,  0.65412169,  0.54019981,  0.89592433,  1.67284204,
        2.24105342,  0.56661325,  0.21090559,  0.14703623,  0.47205993,
        2.05637723,  1.75226942,  0.37602174,  0.23904882,  0.22844971,
        0.45631757,  1.03736031,  0.67421091,  0.17537874,  0.10514494,
        0.10574877,  0.19640785,  0.40049154,  0.6442232 ,  0.20279655,
        0.19270376,  0.11760767,  0.15466839,  0.40377361,  0.59509215,
        0.25689739,  0.11058348,  0.13655547,  0.48130683,  1.17390632,
        1.86901793,  0.22357961,  0.14819718,  0.15080712,  0.14139751,
        1.65705573])
```

In [28]:
```
xcoor = np.zeros((len(rd_x_new),1))
xcoor[:,0] = rd_x_new

ycoor = np.zeros((len(rd_y_new),1))
ycoor[:,0] = rd_y_new

beta_deg_df = np.zeros((len(rd_y_new),1))
beta_deg_df[:,0] = beta_deg

vec_orientation = np.zeros((len(rd_y_new),1))
vec_orientation[:,0] = orient

xc = np.zeros((len(x_circle),1))
xc[:,0] = x_circle

yc = np.zeros((len(y_circle),1))
yc[:,0] = y_circle

std = np.zeros((len(y_circle),1))
std[:,0] = (result_period3_new[:,2])

data_period1 = np.hstack((result_period1_new, xcoor, ycoor, beta_deg_df, vec_orientation))
data_period2 = np.hstack((result_period2_new, xcoor, ycoor,beta_deg_df, vec_orientation))
data_period3 = np.hstack((result_period3_new, xcoor, ycoor,beta_deg_df, vec_orientation))


std_data = np.hstack((xc, yc, vec_orientation, std))
```

In [30]:
```
c = ['Segment', 'trans_vel', 'std_trans', 'normal_vel', 'std_normal', 'rd_x', 'rd_y', 'Beta', 'Orientation']

data_period1 = pd.DataFrame(data=data_period1, columns=c)
data_period2 = pd.DataFrame(data=data_period2, columns=c)
data_period3 = pd.DataFrame(data=data_period3, columns=c)
```

In [31]:
```
display(data_period3.head(20))
```

	Segment	trans_vel	std_trans	normal_vel	std_normal	rd_x	rd_y	Beta	Orientation
0	11.0	29.182651	21.058246	-44.540885	2.898916	252058.389281	571958.494964	105.0	195.0
1	12.0	13.150015	3.266396	-42.451797	0.816971	252371.075110	571763.127126	135.0	225.0
2	13.0	11.487814	1.757813	-41.096108	0.779575	252532.638979	571482.527359	165.0	255.0
3	15.0	28.734492	5.225396	-34.504299	1.409237	252398.644311	570908.344549	225.0	315.0
4	19.0	10.809859	1.584228	-38.637041	0.645186	251223.068496	571189.342541	345.0	75.0
5	110.0	-4.260897	1.750790	-44.561418	0.627192	251251.518237	571504.754619	15.0	105.0
6	111.0	-4.753327	2.301651	-41.019088	0.617460	251402.120643	571799.846290	45.0	135.0
7	112.0	-9.079087	7.631613	-36.757607	1.141793	251691.627908	571983.620334	75.0	165.0
8	21.0	28.194683	19.448520	-14.369971	2.685325	252370.386705	572987.865455	105.0	195.0
9	22.0	11.990886	2.235533	-8.163300	0.632337	253189.222984	572470.517369	135.0	225.0
10	23.0	15.165060	0.971597	-11.398111	0.736875	253587.509365	571718.682046	165.0	255.0
11	24.0	16.106978	0.924652	-12.301707	0.740453	253598.008711	570941.292227	195.0	285.0
12	25.0	21.462320	3.761478	-15.042053	1.039680	253212.158127	570180.938953	225.0	315.0
13	26.0	44.919065	28.104891	-22.405450	3.952050	252388.626839	569646.954892	255.0	345.0
14	27.0	5.585437	5.732581	-28.387026	0.795665	251357.156318	569644.286798	285.0	15.0
15	28.0	11.808834	2.869281	-36.569188	0.708624	250539.659662	570179.655398	315.0	45.0
16	29.0	14.085223	1.490283	-33.643998	0.734731	250157.258451	570950.815424	345.0	75.0
17	210.0	13.702638	1.142746	-24.031989	0.681269	250182.478034	571761.708795	15.0	105.0
18	211.0	11.485810	2.101740	-10.753199	0.642985	250601.018969	572516.426808	45.0	135.0
19	212.0	6.796776	4.819294	-13.527919	0.755582	251389.274794	572997.407686	75.0	165.0

In [32]:
```
c1 = ['rd_x', 'rd_y', 'Orientation', 'std']
coor_orient = pd.DataFrame(data=std_data, columns=c1)
display(coor_orient.head(30))
```

	rd_x	rd_y	Orientation	std
0	251907.328761	571394.729428	195.0	21.058246
1	252185.105812	571577.157829	225.0	3.266396
2	252310.711452	571423.062057	255.0	1.757813
3	251992.277234	571314.711627	315.0	5.225396
4	251431.898929	571245.298487	75.0	1.584228
5	251169.204026	571526.810645	105.0	1.750790
6	251334.898444	571867.068490	135.0	2.301651
7	251644.631098	572159.014817	165.0	7.631613
8	252224.440288	572443.186010	195.0	19.448520
9	253019.646245	572300.940630	225.0	2.235533
10	253294.542897	571640.181917	255.0	0.971597
11	253286.845784	571024.668082	285.0	0.924652
12	252908.635089	570484.461991	315.0	3.761478
13	252156.108650	570514.724589	345.0	28.104891
14	251386.068667	569752.189154	15.0	5.732581
15	250706.661793	570346.657529	45.0	2.869281
16	250429.364074	571023.725905	75.0	1.490283
17	250447.192678	571690.778720	105.0	1.142746
18	250763.452857	572353.992920	135.0	2.101740
19	251424.457495	572866.104060	165.0	4.819294
20	252741.777185	574195.786437	195.0	5.358348
21	254160.978818	573296.715551	225.0	1.373426
22	254860.414312	571998.450934	255.0	0.371898
23	254858.110220	570642.085618	285.0	0.352633
24	254128.436166	569376.753041	315.0	1.750253
25	252718.888230	568569.096372	345.0	9.073052
26	250981.531143	568418.993760	15.0	4.656920
27	249555.913031	569340.917093	45.0	1.171994
28	248858.112941	570663.956204	75.0	0.348048
29	248928.179337	572059.512014	105.0	0.433518

In [33]:
```
data_period1.to_csv("data_period1.csv")
data_period2.to_csv("data_period2.csv")
data_period3.to_csv("data_period3.csv")

coor_orient.to_csv("orient_coor.csv")
```

In [ ]:

In [ ]:

2.6 P3 - Sentinel-1 viewing geometry

Notebook to simulate the viewing geometry of Sentinel1 at a particular location

In [ ]:
```
%%time

from pathlib import Path

import numpy as np
import matplotlib.pyplot as plt
import matplotlib.colors as colors
from mpl_toolkits.axes_grid1 import make_axes_locatable
import cartopy.crs as ccrs
import cartopy.feature as cfeature

from drama.performance.sar import SARModeFromCfg
from drama.io import cfg
from drama.mission.timeline import LatLonTimeline
```

In [2]:
```
par_file = Path(r"C:\Users\wiets\Python\Afstuderen\Python - Discplacement vector decomposition\drama-fix-keple
mode = SARModeFromCfg(cfg.ConfigFile(par_file), "IWS")
```

In [3]:
```
n_orbits_cycle = 175
lon_repeat_cycle = 360 / n_orbits_cycle

# The location for station ATAP is: lat = 39.125 and lon = 40.515

min_lon = 40.5
max_lon = 40.53
min_lat = 39.110
max_lat = 39.140

lat_grid, lon_grid = np.mgrid[min_lat:max_lat:0.015, min_lon:max_lon:0.015]
lat = lat_grid[:, 0]
lon = lon_grid[0, :]
```

In [4]:
```
print (lon_grid)
print (lat_grid)
```

```
[[40.5   40.515 40.53 ]
 [40.5   40.515 40.53 ]]
[[39.11  39.11  39.11 ]
 [39.125 39.125 39.125]
 [39.14  39.14  39.14 ]]
```

In [5]:
```
%%time
orbit_resolution = 0.03
timeline = LatLonTimeline(
    par_file, np.ravel(lat_grid), np.ravel(lon_grid), inc_angle_range=(mode.incs[0, 0], mode.incs[-1, 1]), dlat
)
```

```
 Parameters:
 -----------
 Inclination:     98.15875661071759 deg
 Semi major:      7070965.02426319 m
 Above Ground:    692828.0242631901 m
 Eccentricity:    0.001131982441389299
 Ascend. Node:    45.82 deg
 Arg. of Perigee: 83.19 deg
 Starttime:       0
 Duration:        12426.485259242287 days
 Time increment:  1.0
 Offset Time:     0.0
Interpolating track
Processing          : [##############################] 100%Wall time: 4min 39s
```

In [8]:
```
# print the incidence angles and heading angles for the location
inc_asc = timeline.asc_acqs[4].theta_i
alpha_d_asc = timeline.asc_acqs[4].northing + 2*np.pi

print (np.rad2deg(inc_asc))
print (np.rad2deg(alpha_d_asc))
```

```
[31.75325529 42.70357049]
[259.04471839 260.38049674]
```

In [10]:
```
# print the incidence angles and heading angles for the location
inc_dsc = timeline.desc_acqs[4].theta_i
alpha_d_dsc = timeline.desc_acqs[4].northing

print (np.rad2deg(inc_dsc))
print (np.rad2deg(alpha_d_dsc))
```

```
[34.80212177 45.10891672]
[100.58557978  99.26703387]
```

In [ ]:

2.7 P3 - Optimal viewing geometry

Notebook to estimate the optimal viewing geometry of a new satellite mission

In [ ]:
```
%%time

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
import matplotlib.colors
from scipy.linalg import null_space

from functions_strap_down_method_nodrama import *
from functions import *

from scipy.linalg import null_space
```

In [2]:
```
def best_geometry(inc_sat, alpha_d_sat, std_sat, inc_new, alpha_d_new, std_new, TLN_frame, def_signal):
    """
    Input:
        Inc_sat: array with incidence angles of the available satellites [radians]
        alpha_d_sat: array with alpha_d values of the available satellites [radians]
        std_sat: array with std_values of the available satellites
        inc_new: array with possible incidence angles [radians]
        alpha_d_new: array with possible values for alpha_d [radians]
        std_new: value for the standard deviation for the new satellite
        def_signal: 3x1 vector with estimated displacement signal: [d_t, d_l, d_n]^T

    Output:
        std_trans0: values for std transversal computed at method without TLN frame uncertainty for all inc and
        std_normal0: values for std normal computed at method without TLN frame uncertainty for all inc and alp

        std_trans: values for std transversal computed at method WITH TLN frame uncertainty for all inc and alp
        std_normal: values for std normal computed at method WITH TLN frame uncertainty for all inc and alpha_d

        inc_opt_t: best value for the incidence angle [radians] (for the transversal comp)
        alpha_d_opt_t: best value for the alpha_d [radians] (for the normal comp)
    """

    N = len(inc_new)
    M = len(alpha_d_new)

    #Uncertainty radar signal (0 method)
    std_los = np.append(std_sat, std_new)
    Qyy0 = np.zeros((len(std_sat)+1, len(std_sat)+1))
    np.fill_diagonal(Qyy0,std_los)

    # Create zero matrices
    std_trans = np.zeros((M,N))
    std_normal = np.zeros((M,N))
    std_trans0 = np.zeros((M,N))
    std_normal0 = np.zeros((M,N))

    for j in range(M):
        for i in range(N):
            inc2 = np.deg2rad(inc_new[i])
            alpha_d2 = np.deg2rad(alpha_d_new[j])

            inc = np.append(inc_sat, inc2)
            alpha_d = np.append(alpha_d_sat, alpha_d2)


            # Compute STD_trans and normal without taking uncertainty TLN frame into account
            A_proj = projection_matrix(inc, alpha_d, TLN_frame[0], TLN_frame[1], TLN_frame[2])
            los_obs = A_proj@def_signal

            A2 = np.delete(A_proj, 1, 1)
            x_hat0, Qx_hat0 = BLUE(A2, los_obs, Qyy0)
            std_trans0[j,i] = np.sqrt(Qx_hat0[0,0])
            std_normal0[j,i] = np.sqrt(Qx_hat0[1,1])

            # Compute STD_trans and normal WITH taking uncertainty TLN frame into account
            # 1. Compute 'expected y vector (needed for linearization)'
            y = TLN_forward_model(inc, alpha_d, TLN_frame[0], TLN_frame[1], TLN_frame[2], def_signal[0,0], def_
            # 2.Compute the Qyy matrix for TLN method with uncertainty TLN frame
            Qyy = TLN_Qyy(std_los, TLN_frame[3], TLN_frame[4], TLN_frame[5])

            # 3. Compute Qx_hat for the TLN jacobian matrix
            Qx_hat = TLN_Qxhat(inc, alpha_d, TLN_frame[0], TLN_frame[1], TLN_frame[2], def_signal[0,0], def_sig

            std_trans[j,i] = np.sqrt(Qx_hat[0,0])
            std_normal[j,i] = np.sqrt(Qx_hat[1,1])

    best_trans = np.where(std_trans == np.amin(std_trans))
    alpha_d_opt_t = alpha_d_new[best_trans[0]]
    inc_opt_t = inc_new[best_trans[1]]

    best_normal = np.where(std_normal == np.amin(std_normal))
    alpha_d_opt_n = alpha_d_new[best_normal[0]]
    inc_opt_n = inc_new[best_normal[1]]

    return std_trans, std_normal, alpha_d_opt_t, inc_opt_t, alpha_d_opt_n, inc_opt_n
```

In [3]:
```
n = 100
m = 200

inc_sat = np.array([np.deg2rad(31.75), np.deg2rad(34.8), np.deg2rad(42.7), np.deg2rad(45.11)])
alpha_d_sat = np.array([np.deg2rad(259.05), np.deg2rad(100.59), np.deg2rad(260.38), np.deg2rad(99.27)])

inc_sat = np.array([np.deg2rad(31.75)])
alpha_d_sat = np.array([np.deg2rad(259.05)])

std_sat = np.array([1])

# inc_sat = np.array([np.deg2rad(31)])
# alpha_d_sat = np.array([np.deg2rad(260)])

# std_sat = np.array([1])

inc_new = np.linspace(10,80, n)
alpha_d_new = np.linspace(0,360,m)
std_new = 1

# TLN frame orientation
east = 18.00 #mm/yr
north = 5.45 #mm/yr
beta =  np.rad2deg(np.arctan(north/east) + np.pi)
gamma_t = 0
gamma_l = 0
beta_r, gamma_t_r, gamma_l_r = np.deg2rad(beta), np.deg2rad(gamma_t), np.deg2rad(gamma_l)

# Uncertainty frame
std_beta_r = np.deg2rad(7)
std_gamma_t_r = np.deg2rad(5)
std_gamma_l_r = np.deg2rad(5)

TLN_frame = np.array([beta_r, gamma_t_r, gamma_l_r, std_beta_r, std_gamma_t_r, std_gamma_l_r])

#estimated displacement signal
d_t = 1
d_l = 0
d_n = 1
def_signal = np.matrix([[d_t], [d_l], [d_n]])

# Run the optimal function
std_trans, std_normal, alpha_d_opt_t, inc_opt_t, alpha_d_opt_n, inc_opt_n = best_geometry(inc_sat, alpha_d_sat,
                                                                                          std_sat, inc_new, alp
                                                                                          std_new, TLN_frame, d
```

In [4]:
```
inc_plot = np.linspace(np.min(inc_new), np.max(inc_new),5)
azimuth_EDP_plot = np.linspace(np.min(alpha_d_new), np.max(alpha_d_new),10)
```

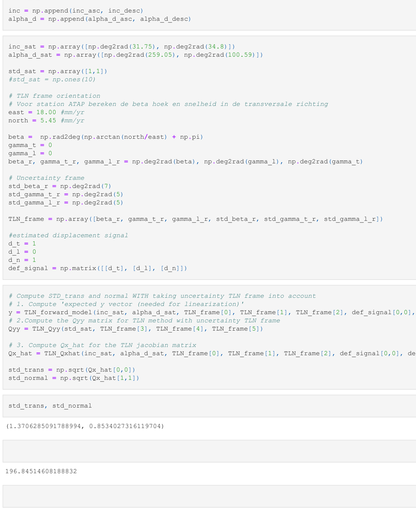
In [5]:
```
np.min(std_trans), np.min(std_normal)
```

Out[5]:
```
(0.9465721247785467, 0.7969496917151403)
```

In [6]:
```
alpha_d_opt_t, inc_opt_t, alpha_d_opt_n, inc_opt_n
```

Out[6]:
```
(array([106.73366834]),
 array([77.17171717]),
 array([106.73366834]),
 array([17.77777778]))
```

In [8]:
```
fig = plt.figure(figsize = (15,8))
ax = fig.add_subplot(121)
std_t = ax.matshow(((std_trans)), cmap='jet', vmin = 0, vmax = 3, aspect = 1)
cb = fig.colorbar(std_t)
cb.set_label(label = 'SD for the transversal direction', size=16)
#cb.ax.tick_params(labelsize='large')
xaxis = np.linspace(0, len(inc_new), len(inc_plot))
yaxis = np.linspace(0, len(alpha_d_new), len(azimuth_EDP_plot))
ax.set_xticks(yaxis)
ax.set_xticks(xaxis)
ax.set_xticklabels(azimuth_EDP_plot)
ax.set_xticklabels(inc_plot)
plt.title(r'$\theta$ [degrees]', fontsize = 13)
plt.ylabel(r'$\alpha_d$ [degrees]', fontsize = 13)

ax = fig.add_subplot(122)
std_n = ax.matshow(((std_normal)), cmap='jet', vmin = 0, vmax = 3, aspect = 1)
cb = fig.colorbar(std_n)
cb.set_label(label = 'SD for the normal direction', size=16)
#cb.ax.tick_params(labelsize='large')
xaxis = np.linspace(0, len(inc_new), len(inc_plot))
yaxis = np.linspace(0, len(alpha_d_new), len(azimuth_EDP_plot))
ax.set_xticks(yaxis)
ax.set_xticks(xaxis)
ax.set_xticklabels(azimuth_EDP_plot)
ax.set_xticklabels(inc_plot)
plt.title(r'$\theta$ [degrees]', fontsize = 13)
plt.ylabel(r'$\alpha_d$ [degrees]', fontsize = 13)
fig.savefig('NAF_optimal_2sat.png', dpi=fig.dpi)
```



In [29]:
```

```

Out[29]:
```
(array([106.73366834]),
 array([80.]),
 array([104.92462312]),
 array([11.41414141]))
```

Bereken eerst wat we nu al kunnen

Dus

1. 1 keer asc en 1 dsc
2. 1 asc en 2 dsc
3. 2 asc en 2 dsc
4. Hypothetich geval, wat als we 5 asc en 5 dsc

In [72]:
```
inc_asc = np.linspace(31.75,42.7,5)
alpha_d_asc = np.linspace(259.05,260.38,5)
inc_desc = np.linspace(34.8, 45.11,5)
alpha_d_desc = np.linspace(100.59,99.27,5)

inc = np.append(inc_asc, inc_desc)
alpha_d = np.append(alpha_d_asc, alpha_d_desc)
```

In [17]:
```
inc_sat = np.array([np.deg2rad(31.75), np.deg2rad(34.8)])
alpha_d_sat = np.array([np.deg2rad(259.05), np.deg2rad(100.59)])

std_sat = np.array([1,1])
#std_sat = np.ones(10)

# TLN frame orientation
# Voor station ATAP bereken de beta hoek en snelheid in de transversale richting
east = 18.00 #mm/yr
north = 5.45 #mm/yr

beta =  np.rad2deg(np.arctan(north/east) + np.pi)
gamma_t = 0
gamma_l = 0
beta_r, gamma_t_r, gamma_l_r = np.deg2rad(beta), np.deg2rad(gamma_t), np.deg2rad(gamma_l)

# Uncertainty frame
std_beta_r = np.deg2rad(7)
std_gamma_t_r = np.deg2rad(5)
std_gamma_l_r = np.deg2rad(5)

TLN_frame = np.array([beta_r, gamma_t_r, gamma_l_r, std_beta_r, std_gamma_t_r, std_gamma_l_r])

#estimated displacement signal
d_t = 1
d_l = 0
d_n = 1
def_signal = np.matrix([[d_t], [d_l], [d_n]])
```

In [20]:
```
# Compute STD_trans and normal WITH taking uncertainty TLN frame into account
# 1. Compute 'expected y vector (needed for linearization)'
y = TLN_forward_model(inc_sat, alpha_d_sat, TLN_frame[0], TLN_frame[1], TLN_frame[2], def_signal[0,0], def_sign
# 2.Compute the Qyy matrix for TLN method with uncertainty TLN frame
Qyy = TLN_Qyy(std_sat, TLN_frame[3], TLN_frame[4], TLN_frame[5])

# 3. Compute Qx_hat for the TLN jacobian matrix
Qx_hat = TLN_Qxhat(inc_sat, alpha_d_sat, TLN_frame[0], TLN_frame[1], TLN_frame[2], def_signal[0,0], def_signal[

std_trans = np.sqrt(Qx_hat[0,0])
std_normal = np.sqrt(Qx_hat[1,1])
```

In [21]:
```
std_trans, std_normal
```

Out[21]:
```
(1.3706285091788994, 0.8534027316119704)
```

In [9]:
```

```

Out[9]:
```
196.84514608188832
```

In [ ]:
```

```

2.8 Monte Carlo simulations

Notebook for the monte carlo simulations

In [ ]:
```
%matplotlib notebook

import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import axes3d
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import cm
import matplotlib.colors

import scipy.stats as st
from scipy.stats import norm
from scipy import stats

from math import *
from basic_geometric_functions import *
from functional_models import *
from scipy.linalg import null_space
import numpy as np
```

Monte Carlo analysis

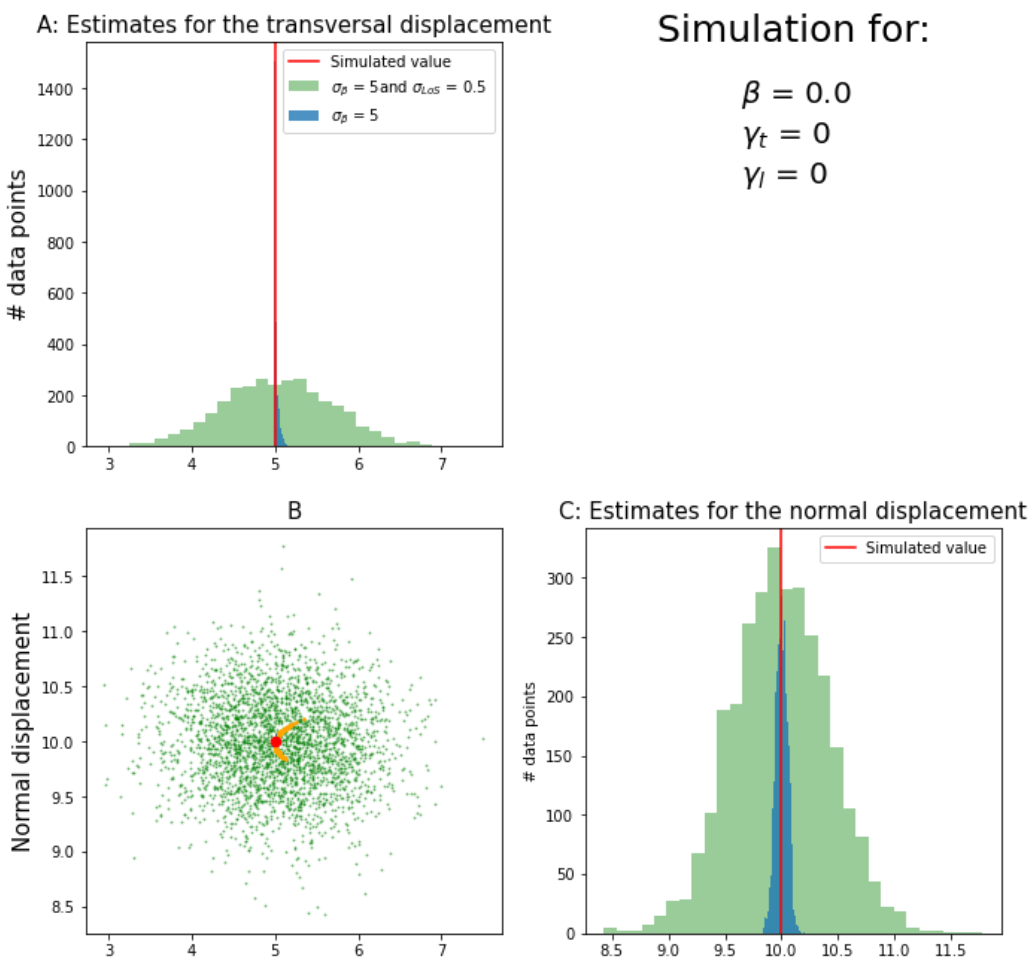
Notebook for the monte carlo analysis

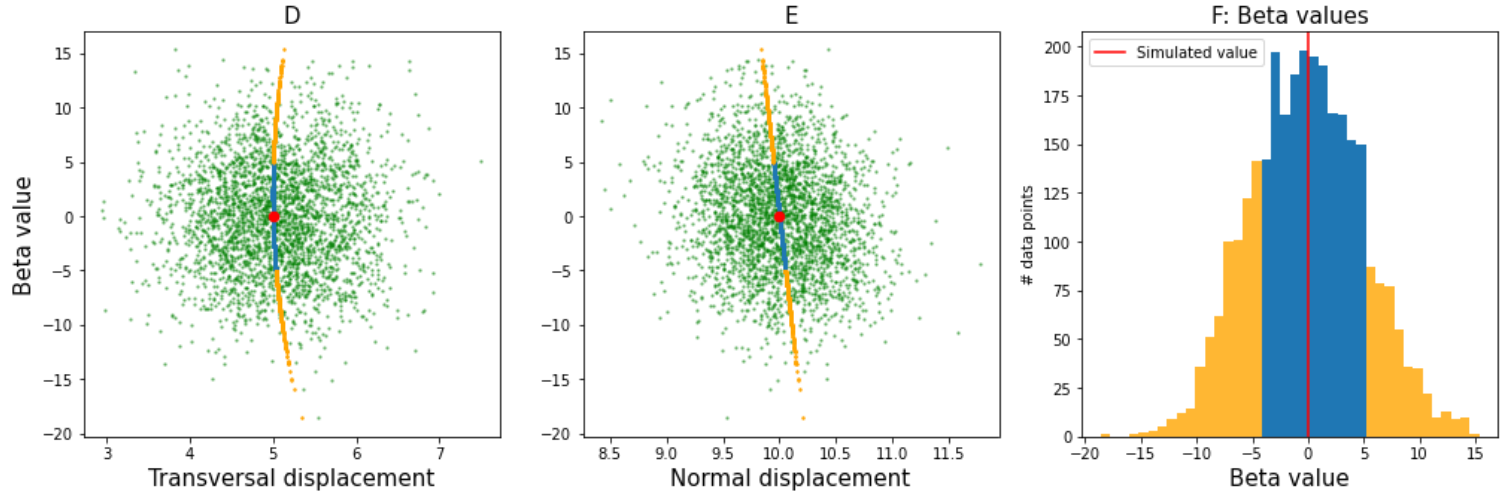
In [2]:
```
# The orientation
#beta = 10
gamma_t = 0
gamma_l = 0

# The simulated signal
d_t = 5
d_l = 0
d_n =10
d_tln = np.matrix([[d_t], [d_l], [d_n]])

# The satellite characteristics
inc = np.array([30, 35])
alpha_d = np.array([350-90, 192-90])
std_los = np.array([0.0001, 0.0001])
std_los_ruis = np.array([0.5, 0.5])
#std_los_ruis = np.array([0.0001, 0.0001])


# The uncertainty for beta
std_beta = 5

N = 3000

b = 40

beta_range = np.linspace(0,360,37)
print (beta_range)

#beta_range = np.linspace(0,20, 3)
#print (beta_range)

# The Qyymatrix
Qyy = np.zeros((2,2))
np.fill_diagonal(Qyy, (std_los)**2) # Add std's as variances in the Qyy matrix

inca = inc[0]
incd = inc[1]
alpha_d1 = alpha_d[0]
alpha_d2 = alpha_d[1]
```

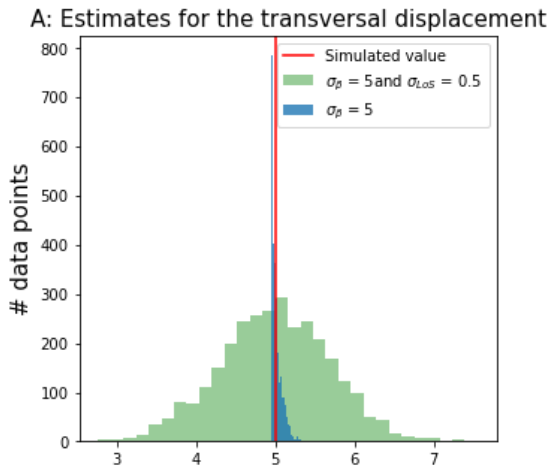
```
[  0.  10.  20.  30.  40.  50.  60.  70.  80.  90. 100. 110. 120. 130.
 140. 150. 160. 170. 180. 190. 200. 210. 220. 230. 240. 250. 260. 270.
 280. 290. 300. 310. 320. 330. 340. 350. 360.]
```

In [11]:
```
for k in range(len(beta_range)):
#for k in range(1):


    beta = beta_range[k]
    print (beta)
    # Define the projection matrix A
    A_proj = projection_matrix(inc, alpha_d, beta_range[k], gamma_l, gamma_t)
    y_los = A_proj@d_tln
    null = null_space(A_proj)

    upper = beta_range[k]+std_beta
    lower = beta_range[k]-std_beta

    beta_values = np.random.normal(beta_range[k], std_beta, N)
    asc_obs = np.random.normal(np.squeeze(np.asarray(y_los))[0], std_los[0], N)
    dsc_obs = np.random.normal(np.squeeze(np.asarray(y_los))[1], std_los[1], N)

    asc_obs_r = np.random.normal(np.squeeze(np.asarray(y_los))[0], std_los_ruis[0], N)
    dsc_obs_r = np.random.normal(np.squeeze(np.asarray(y_los))[1], std_los_ruis[1], N)


    sol_trans = np.zeros(N)
    sol_normal = np.zeros(N)
    sol_trans_r = np.zeros(N)
    sol_normal_r = np.zeros(N)
    sol_trans_obs = np.zeros(N)
    sol_normal_obs = np.zeros(N)

    y_los_r = np.zeros((2,1))

    A_proj_obs = projection_matrix(inc, alpha_d, beta_range[k], gamma_l, gamma_t)
    A_new_obs = np.delete(A_proj_obs,1,1)


    for i in range(N):
        y_los[0,0] = asc_obs[i]
        y_los[1,0] = dsc_obs[i]

        y_los_r[0,0] = asc_obs_r[i]
        y_los_r[1,0] = dsc_obs_r[i]

        # Computing estimates only 'beta noise'
        A_proj = projection_matrix(inc, alpha_d, beta_values[i], gamma_l, gamma_t)
        A_new = np.delete(A_proj,1,1)
        x_hat, Qx_hat = BLUE(A_new, y_los, Qyy)
        x_hat = np.squeeze(np.asarray(x_hat))
        sol_trans[i] = x_hat[0]
        sol_normal[i] = x_hat[1]

        Qx_hat = np.squeeze(np.asarray(Qx_hat))

        # Compute estimates for 'beta and obsveration noise'
        x_hat_r, Qx_hat = BLUE(A_new, y_los_r, Qyy)
        x_hat_r = np.squeeze(np.asarray(x_hat_r))
        sol_trans_r[i] = x_hat_r[0]
        sol_normal_r[i] = x_hat_r[1]

        # Compute estimates only LoS noise (one beta value)
        x_hat_obs, Qx_hat = BLUE(A_new_obs, y_los_r, Qyy)
        x_hat_obs = np.squeeze(np.asarray(x_hat_obs))
        sol_trans_obs[i] = x_hat_obs[0]
        sol_normal_obs[i] = x_hat_obs[1]


    size = 2

    fig = plt.figure(figsize = (17,17))

    plt.subplot(331)
    hist_trans = plt.hist(sol_trans_r, bins=30, alpha = 0.4, color='green', label = r'$\sigma_{\beta}$ = '+ str
    #hist_trans = plt.hist(sol_trans_obs, bins=30, alpha = 0.4, color='red', label = r'$\sigma_{LoS}$ = '+ str
    hist_trans = plt.hist(sol_trans, bins=30, alpha = 0.8, label = r'$\sigma_{\beta}$ = '+ str(std_beta))
    #plt.xlabel('Transversal displacement', fontsize = 15)
    plt.ylabel('# data points', fontsize = 15)
    plt.title('A: Estimates for the transversal displacement', fontsize = 15)
    # print('number of data points in each bin:', hist_beta[0])
    # print('limits of the bins:', hist_beta[1])
    plt.axvline(d_t, color='r', label='Simulated value')
    plt.legend()

    plt.subplot(332)
    plt.text(0.35, 2, 'Simulation for:', fontsize = 25)
    plt.text(0.75, 1.7, r'$\beta$ = '+ str(beta),fontsize = 20)
    plt.text(0.75, 1.5, r'$\gamma_t$ = '+ str(gamma_t),fontsize = 20)
    plt.text(0.75, 1.3, r'$\gamma_l$ = '+ str(gamma_l),fontsize = 20)
    #plt.text(1, 1.1, r'Incidence angles: $\theta_a$ = '+ str(inca)+', $\theta_d$ = '+ str(incd))
    #plt.text(1, 0.9, r'Azimuth zero-Doppler: $\alpha$ = '+ str(alpha_d1)+', $\alpha$ = '+ str(alpha_d1))
    plt.ylim([0,2])
    plt.xlim([0,2])
    plt.axis('off')

#     plt.subplot(333)
#     plt.arrow(0,0,np.squeeze(np.asarray(null))[0], np.squeeze(np.asarray(null))[1], width = 0.05)
#     plt.axis('equal')
#     plt.xlim(-1.3,2.3)
#     plt.ylim(-1.3,2.3)
#     plt.ylabel('Longitudinal direction')
#     plt.xlabel('Transversal direction')
#     plt.text(1, 1.5,' T: '+str(np.around(null[0],3))+'\n L: '+str(np.around(null[1],3))+'\n N: ' +str(np.arou
#     plt.title('B: Direction of the null-space (top view)')



    plt.subplot(334)
    plt.scatter(sol_trans_r, sol_normal_r, s=size*0.5, alpha = 0.4, color='green')
    #plt.scatter(sol_trans_obs, sol_normal_obs, s=size*0.5, alpha = 0.4, color='red')

    plt.scatter(sol_trans[(beta_values>=lower)&(beta_values<=upper)], sol_normal[(beta_values>=lower)&(beta_val
    plt.scatter(sol_trans[(beta_values>upper)], sol_normal[(beta_values>upper)], s=size, color='orange')
    plt.scatter(sol_trans[(beta_values<lower)], sol_normal[(beta_values<lower)], s=size, color='orange')
    plt.scatter(d_t, d_n, s=size+40, color='red')
    #plt.xlabel('Transversal displacement', fontsize = 15)
    plt.ylabel('Normal displacement', fontsize = 15)
    plt.title('B', fontsize = 15)

    plt.subplot(335)
    hist_trans = plt.hist(sol_normal_r, bins=30, alpha = 0.4, color='green')
    #hist_trans = plt.hist(sol_normal_obs, bins=30, alpha = 0.4, color='red')

    hist_trans = plt.hist(sol_normal, bins=30, alpha = 0.8)
    #plt.xlabel('Normal displacement')
    plt.ylabel('# data points')
    plt.title('C: Estimates for the normal displacement', fontsize = 15)
    plt.axvline(d_n, color='r', label='Simulated value')
    plt.legend()
    # print('number of data points in each bin:', hist_beta[0])
    # print('limits of the bins:', hist_beta[1])


    plt.subplot(337)
    plt.scatter(sol_trans_r, beta_values, s=size, alpha = 0.4, color='green')
    #plt.scatter(sol_trans_obs, beta, s=size, alpha = 0.5, color='red')

    plt.scatter(sol_trans[(beta_values>=lower)&(beta_values<=upper)], beta_values[(beta_values>=lower)&(beta_va
    plt.scatter(sol_trans[(beta_values>upper)], beta_values[(beta_values>upper)], s=size, color='orange')
    plt.scatter(sol_trans[(beta_values<lower)], beta_values[(beta_values<lower)], s=size, color='orange')
    plt.scatter(d_t, beta, s=size+40, color='red')
    plt.xlabel('Transversal displacement', fontsize = 15)
    plt.ylabel('Beta value', fontsize = 15)
    plt.title('D', fontsize = 15)

    plt.subplot(338)
    plt.scatter(sol_normal_r, beta_values, s=size, alpha = 0.4, color='green')
    #plt.scatter(sol_normal_obs, beta, s=size, alpha = 0.5, color='red')

    plt.scatter(sol_normal[(beta_values>=lower)&(beta_values<=upper)], beta_values[(beta_values>=lower)&(beta_v
    plt.scatter(sol_normal[(beta_values>upper)], beta_values[(beta_values>upper)], s=size, color='orange')
    plt.scatter(sol_normal[(beta_values<lower)], beta_values[(beta_values<lower)], s=size, color='orange')
    plt.scatter(d_n, beta, s=size+40, color='red')
    plt.xlabel('Normal displacement', fontsize = 15)
    #plt.ylabel('Beta value')
    plt.title('E', fontsize = 15)

    plt.subplot(339)
    n, bins, patches = plt.hist(beta_values, bins=b)
    for i in range(b):
        if bins[i]<lower:
            patches[i].set_facecolor('orange')
            patches[i].set_alpha(0.8)
        if bins[i]>upper:
            patches[i].set_facecolor('orange')
            patches[i].set_alpha(0.8)
    plt.xlabel('Beta value', fontsize = 15)
    plt.ylabel('# data points')
    plt.title('F: Beta values', fontsize = 15)
    plt.axvline(beta, color='r', label='Simulated value')
    plt.legend()
    fig.savefig("Distribution Normal and Transversal displacement, beta = "+ str(beta_range[k]) +" (null space,
```

```
0.0
10.0
20.0
30.0
40.0
50.0
60.0
70.0
80.0
90.0
100.0
110.0
120.0
130.0
140.0
150.0
160.0
170.0
180.0
190.0
200.0
<ipython-input-11-124123e98dba>:68: RuntimeWarning: More than 20 figures have been opened. Figures created thro
ugh the pyplot interface (`matplotlib.pyplot.figure`) are retained until explicitly closed and may consume too
much memory. (To control this warning, see the rcParam `figure.max_open_warning`).
  fig = plt.figure(figsize = (17,17))
210.0
220.0
230.0
240.0
250.0
260.0
270.0
280.0
290.0
300.0
310.0
320.0
330.0
340.0
350.0
360.0
```

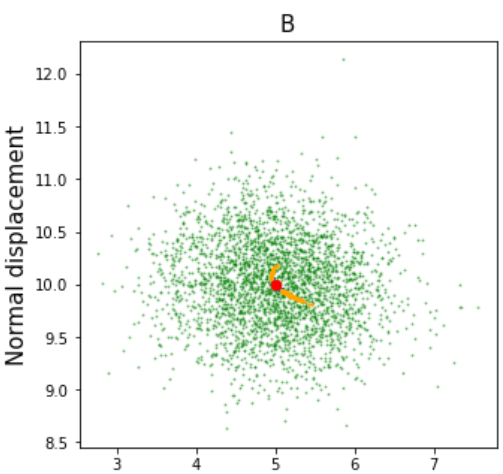


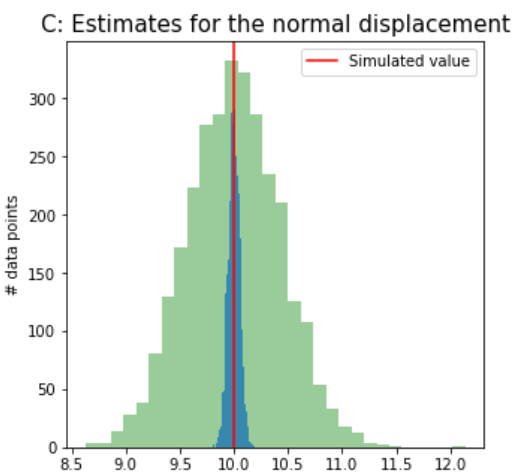


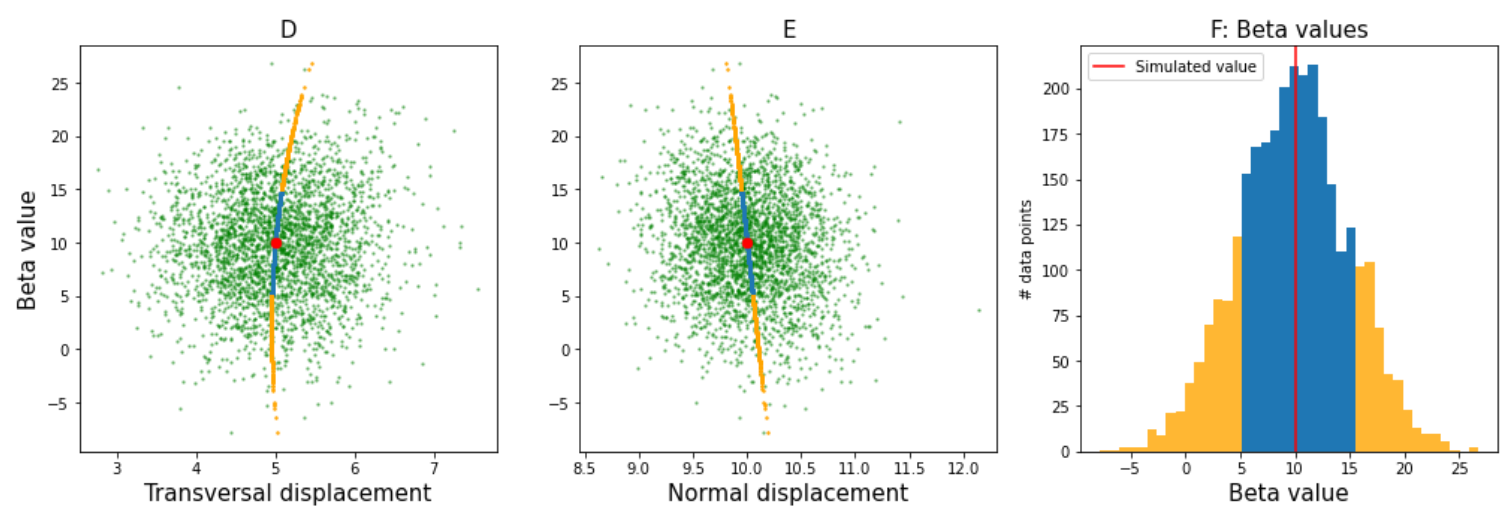


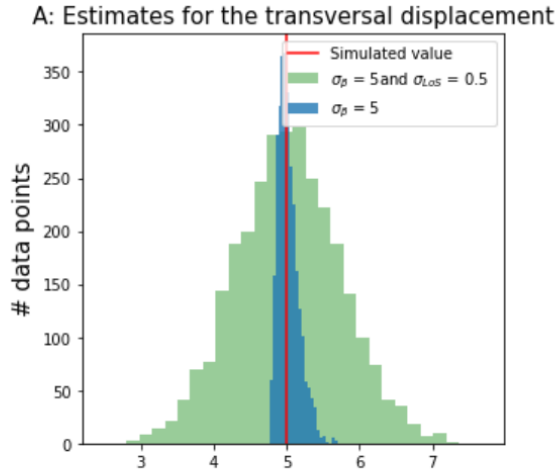
Simulation for:

$\beta = 10.0$
$\gamma_t = 0$
$\gamma_l = 0$



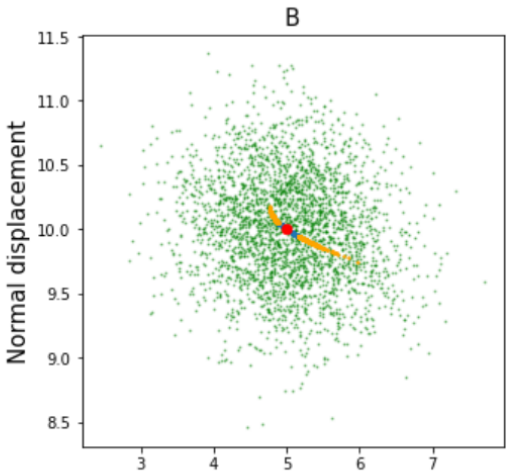


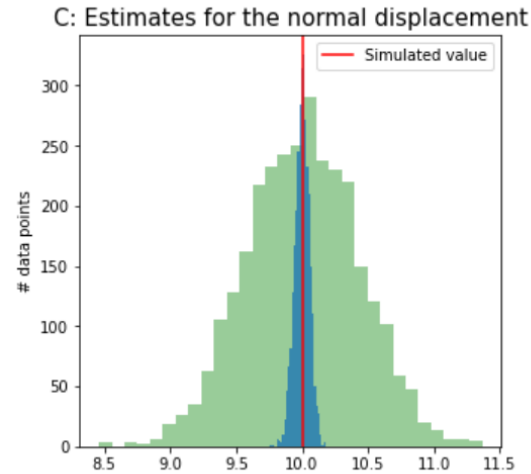


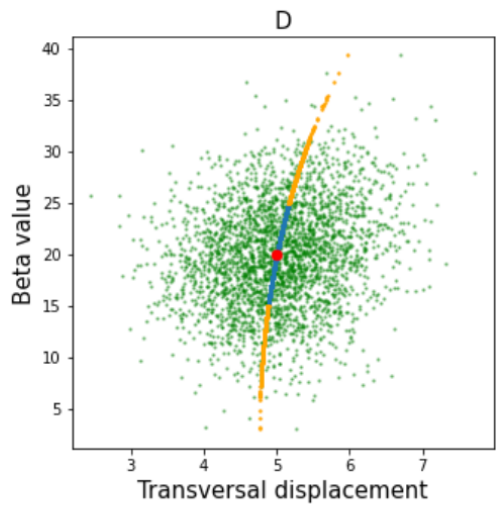


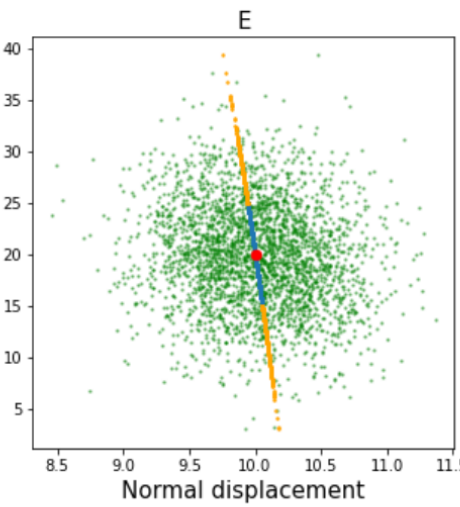
Simulation for:

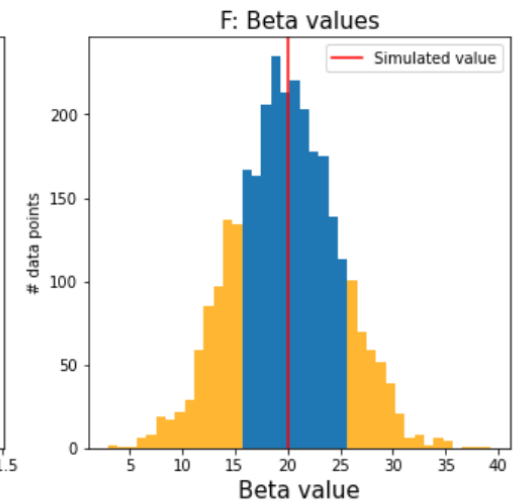
$\beta = 20.0$
$\gamma_t = 0$
$\gamma_l = 0$

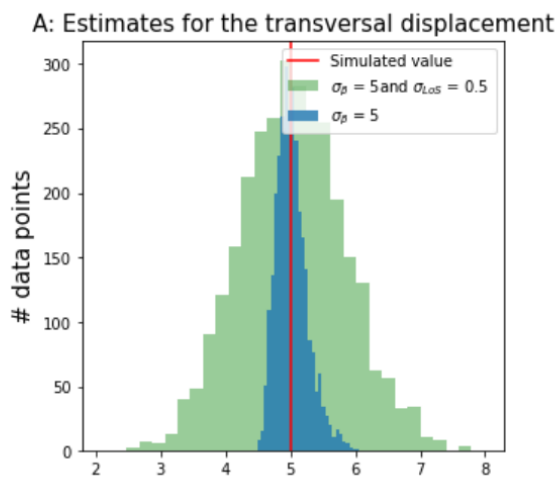






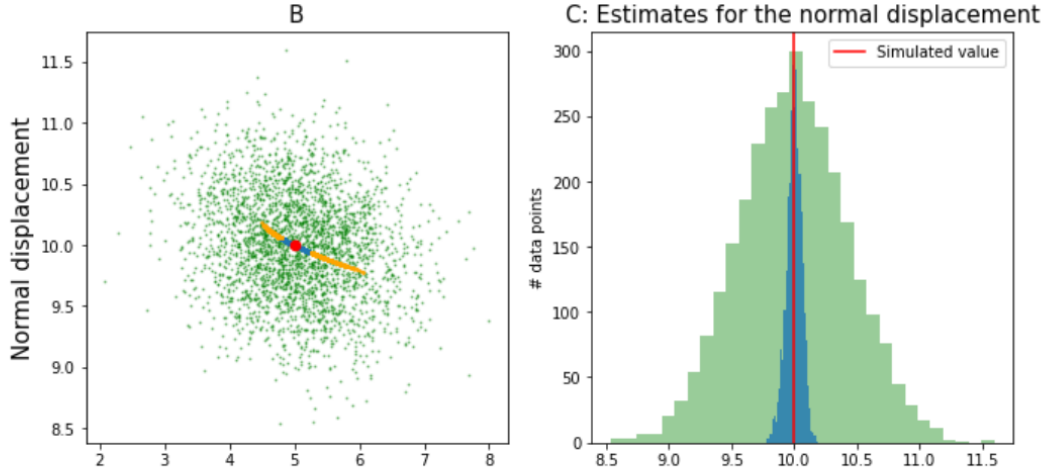


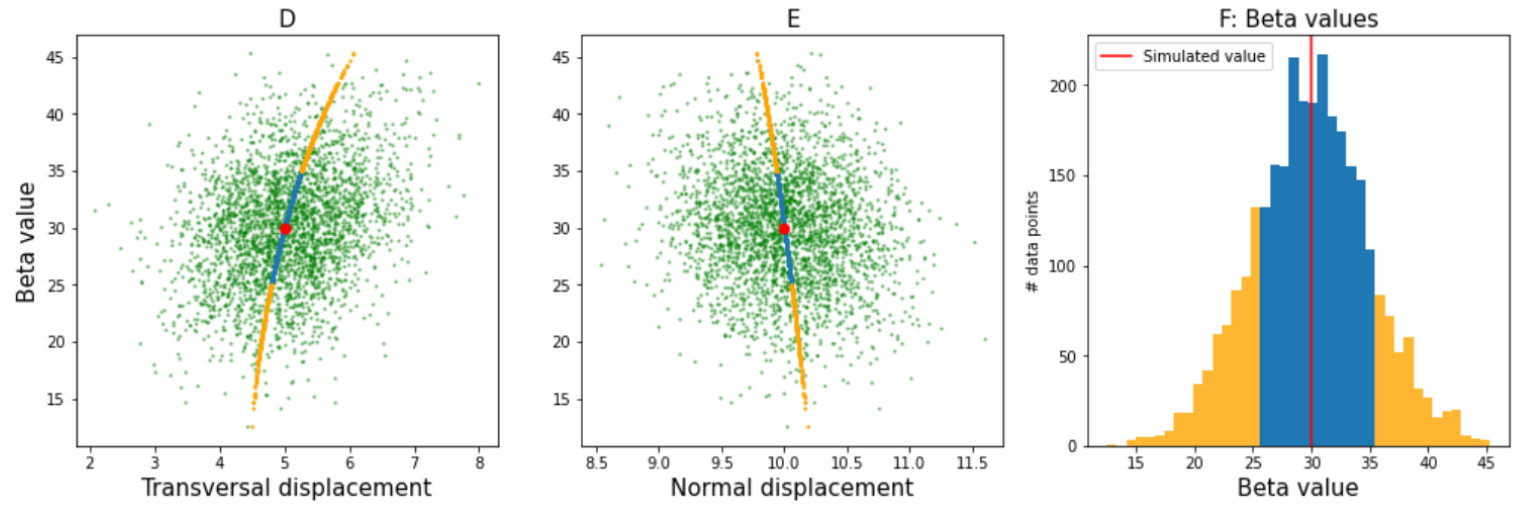


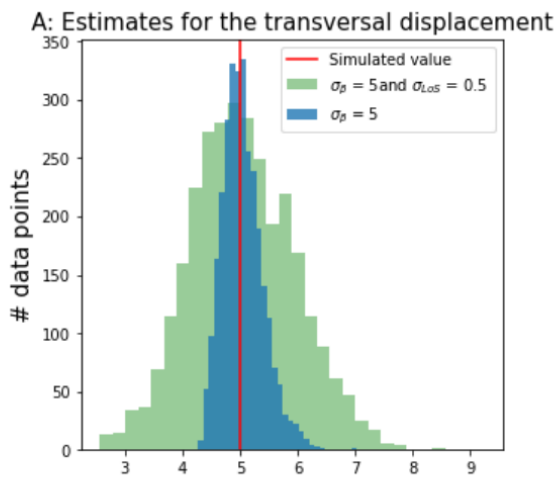


Simulation for:

$\beta = 30.0$
$\gamma_t = 0$
$\gamma_l = 0$





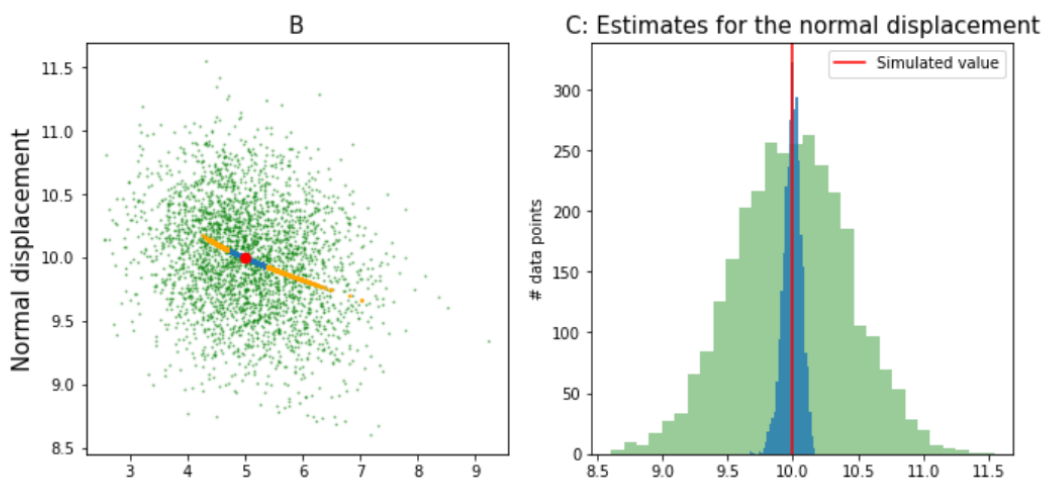


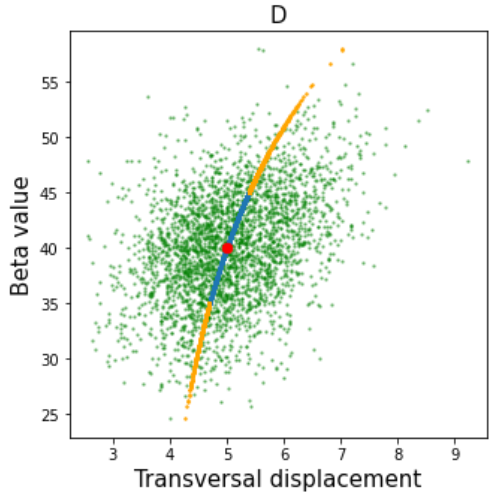
Simulation for:

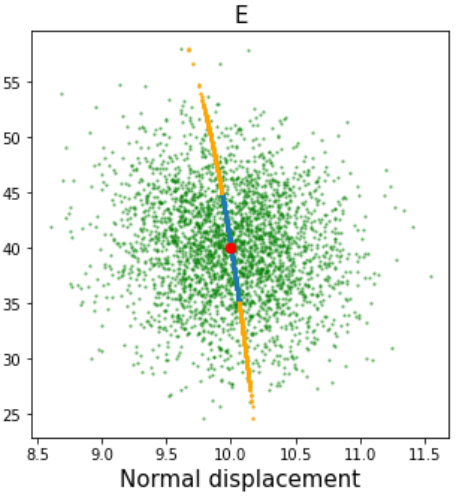
$\beta = 40.0$

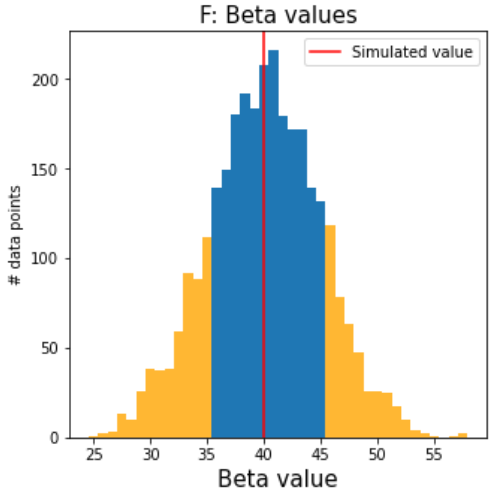
$\gamma_t = 0$

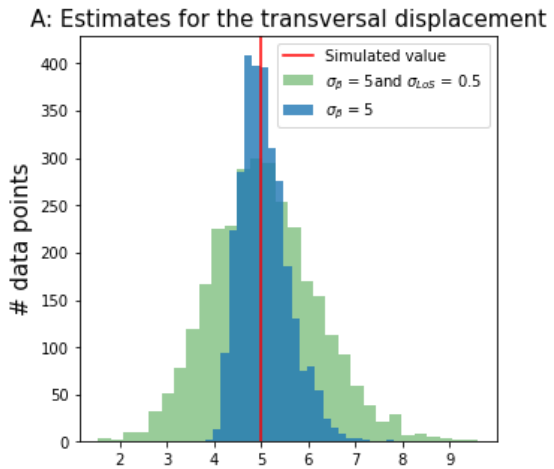
$\gamma_l = 0$





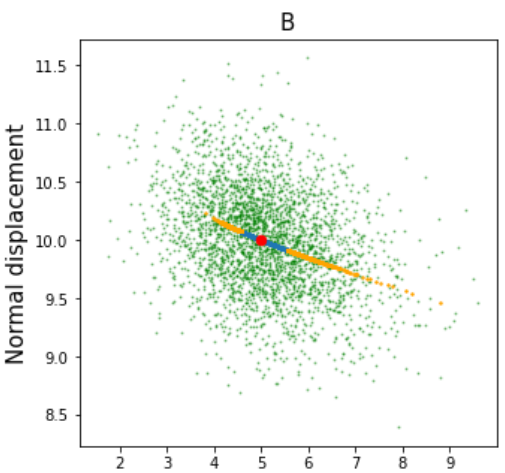


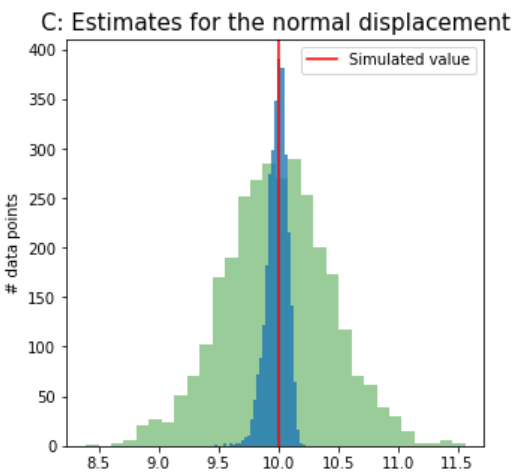


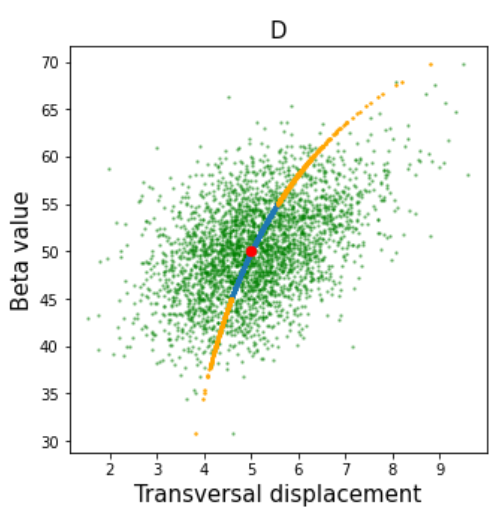


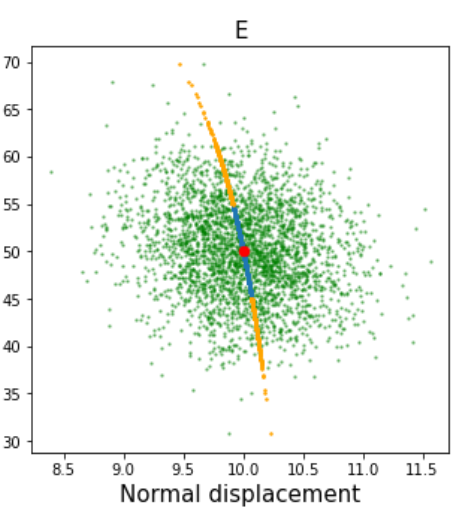
Simulation for:

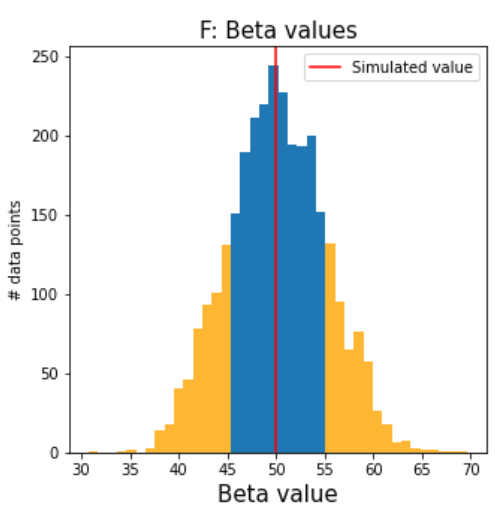
$\beta = 50.0$
$\gamma_t = 0$
$\gamma_l = 0$

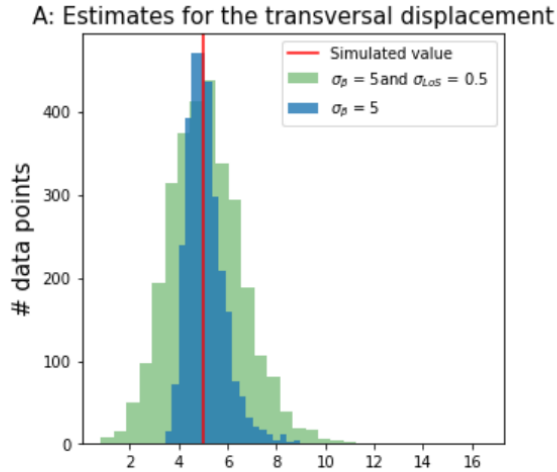






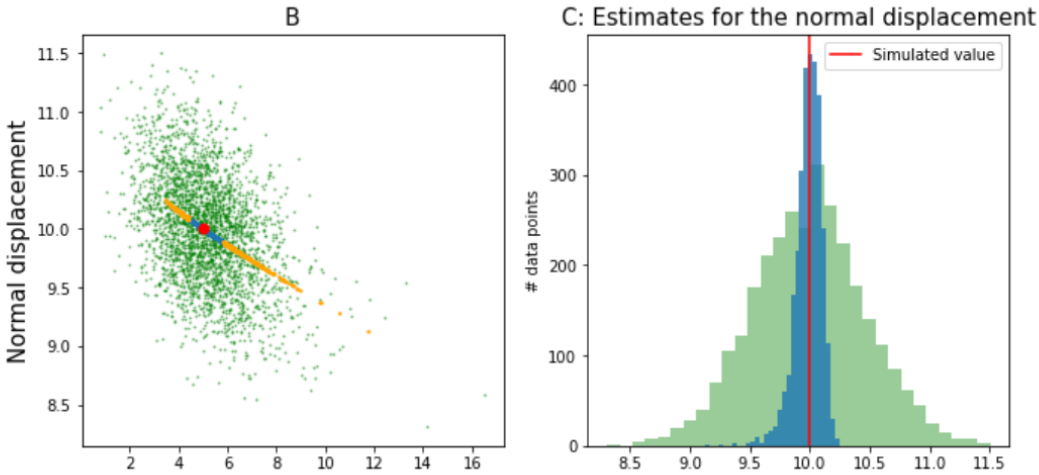


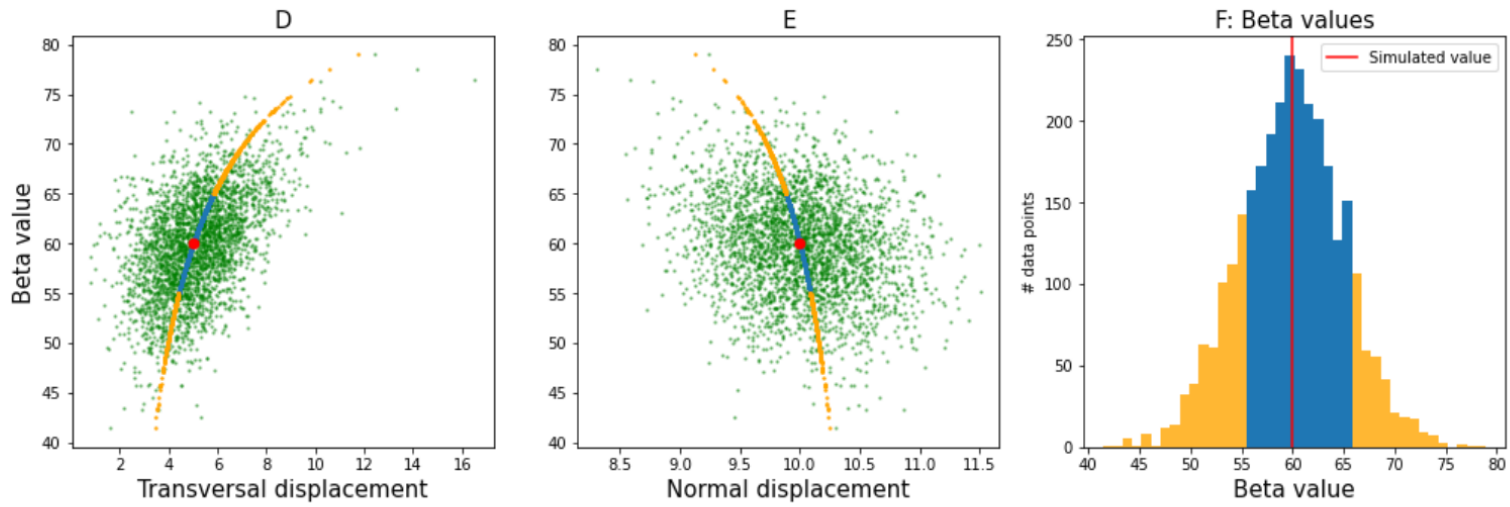


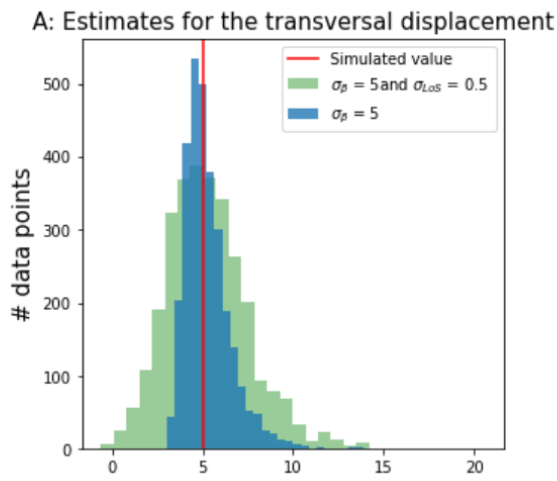


Simulation for:

$\beta = 60.0$
$\gamma_t = 0$
$\gamma_l = 0$

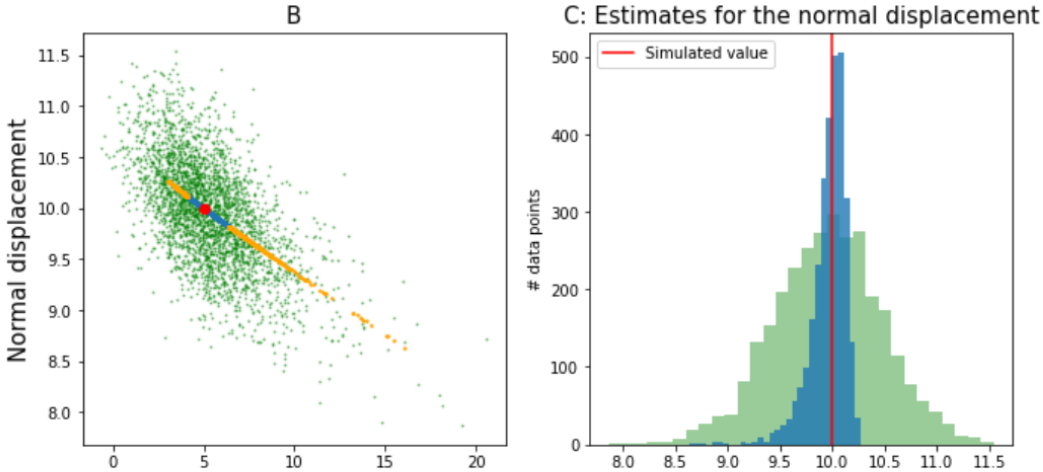


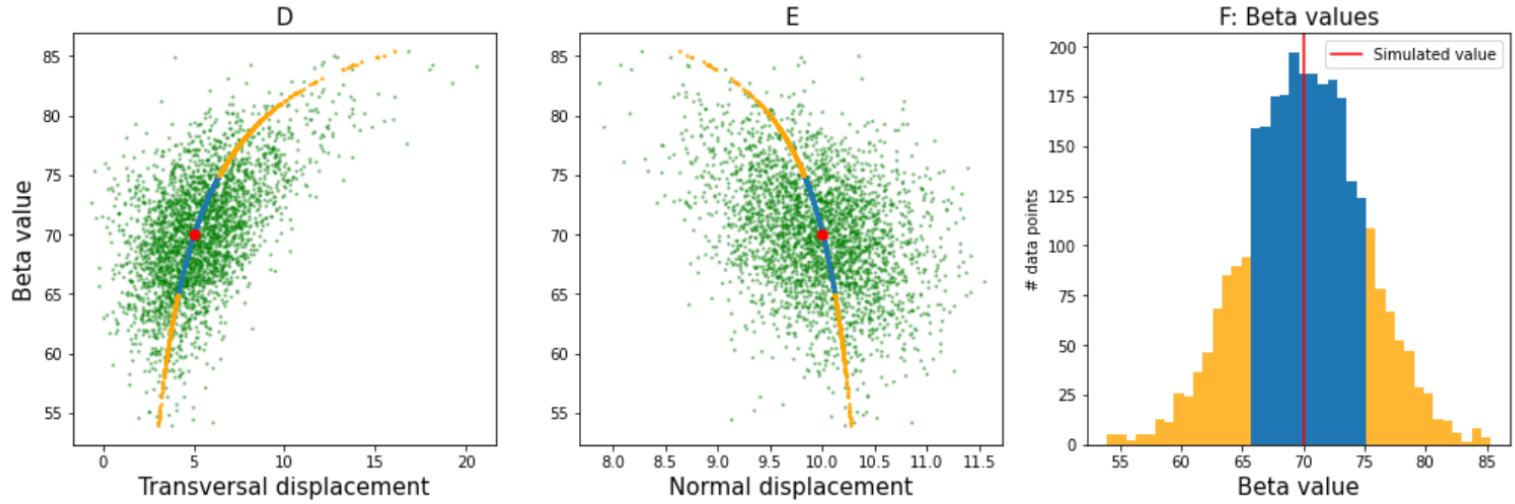


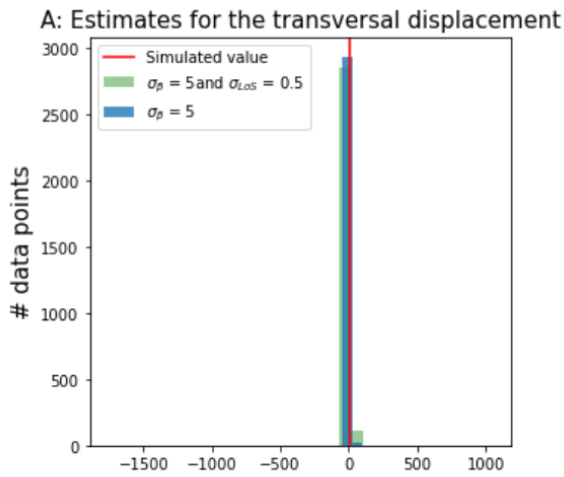


Simulation for:

$\beta = 70.0$
$\gamma_t = 0$
$\gamma_l = 0$





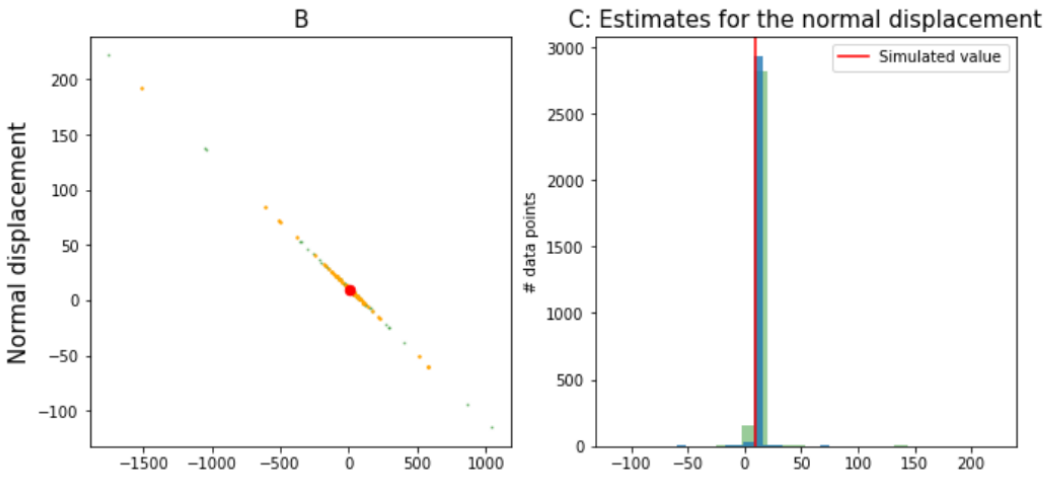


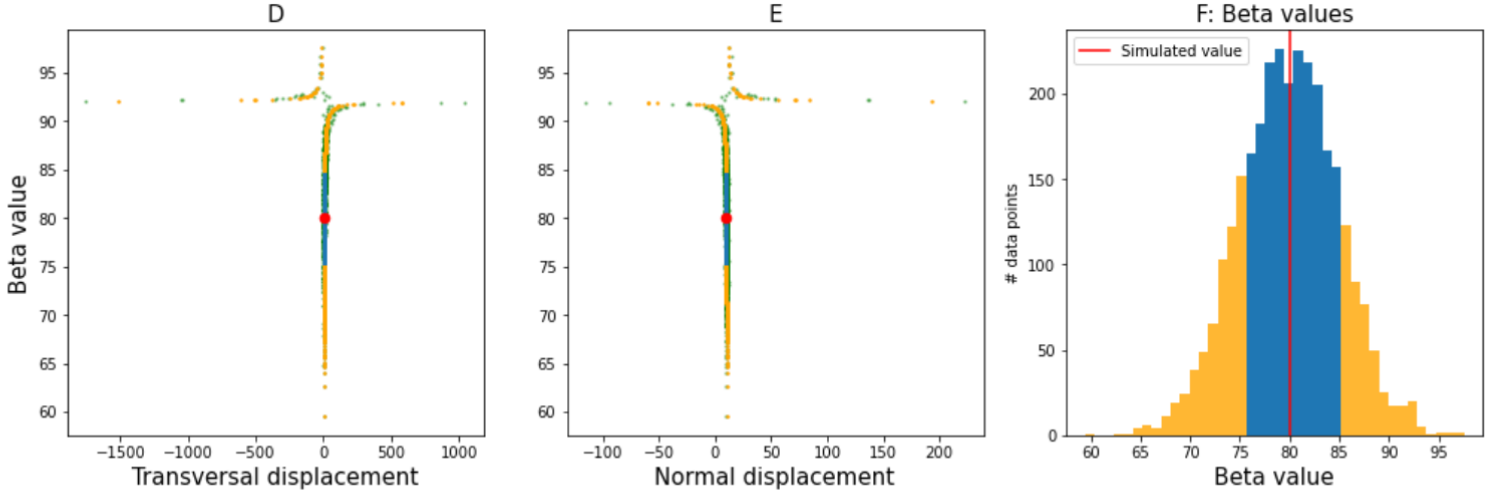
Simulation for:

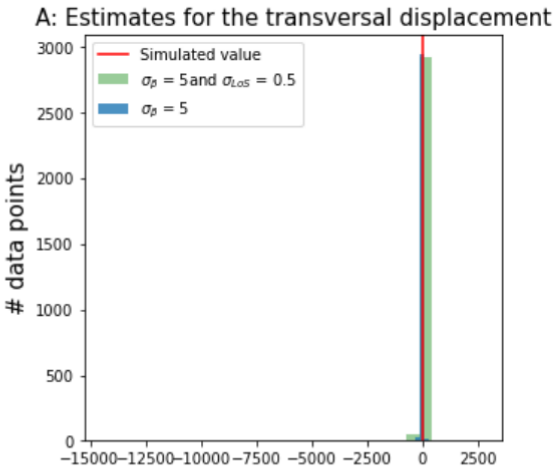
$\beta = 80.0$

$\gamma_t = 0$

$\gamma_l = 0$





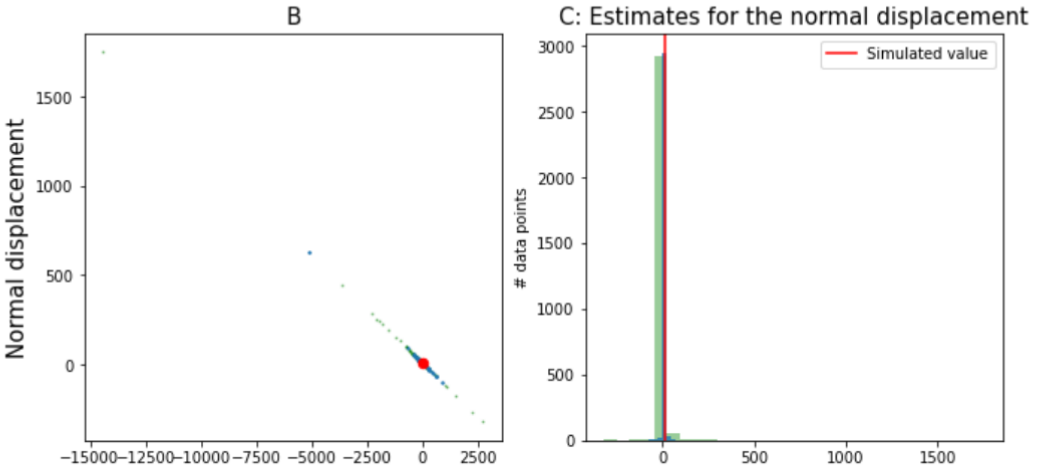


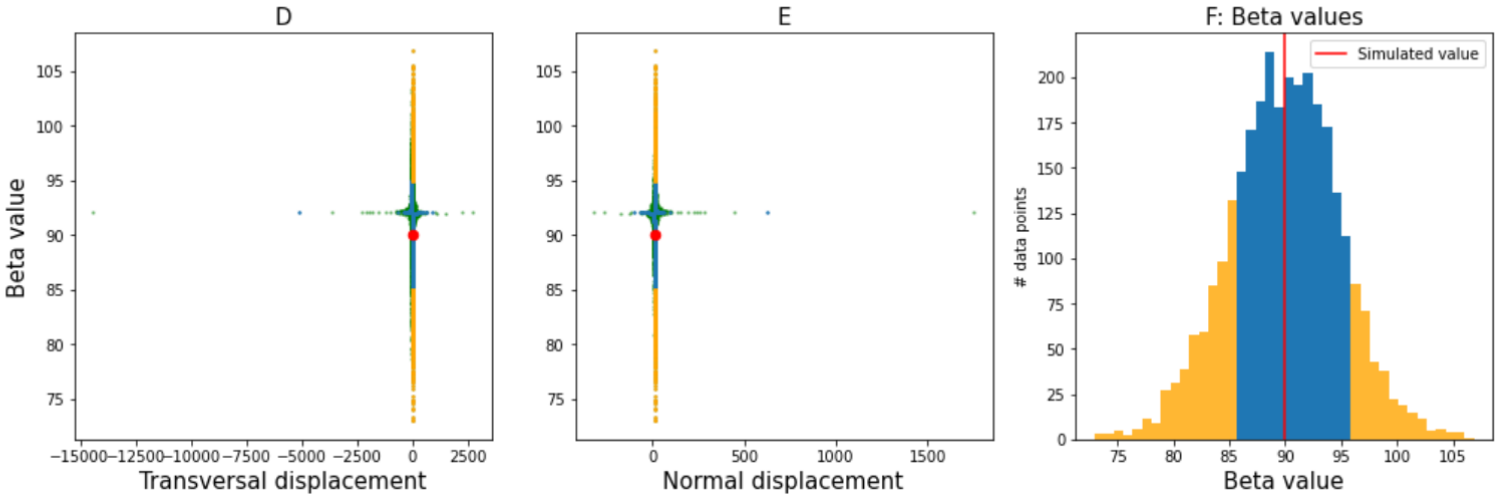
Simulation for:

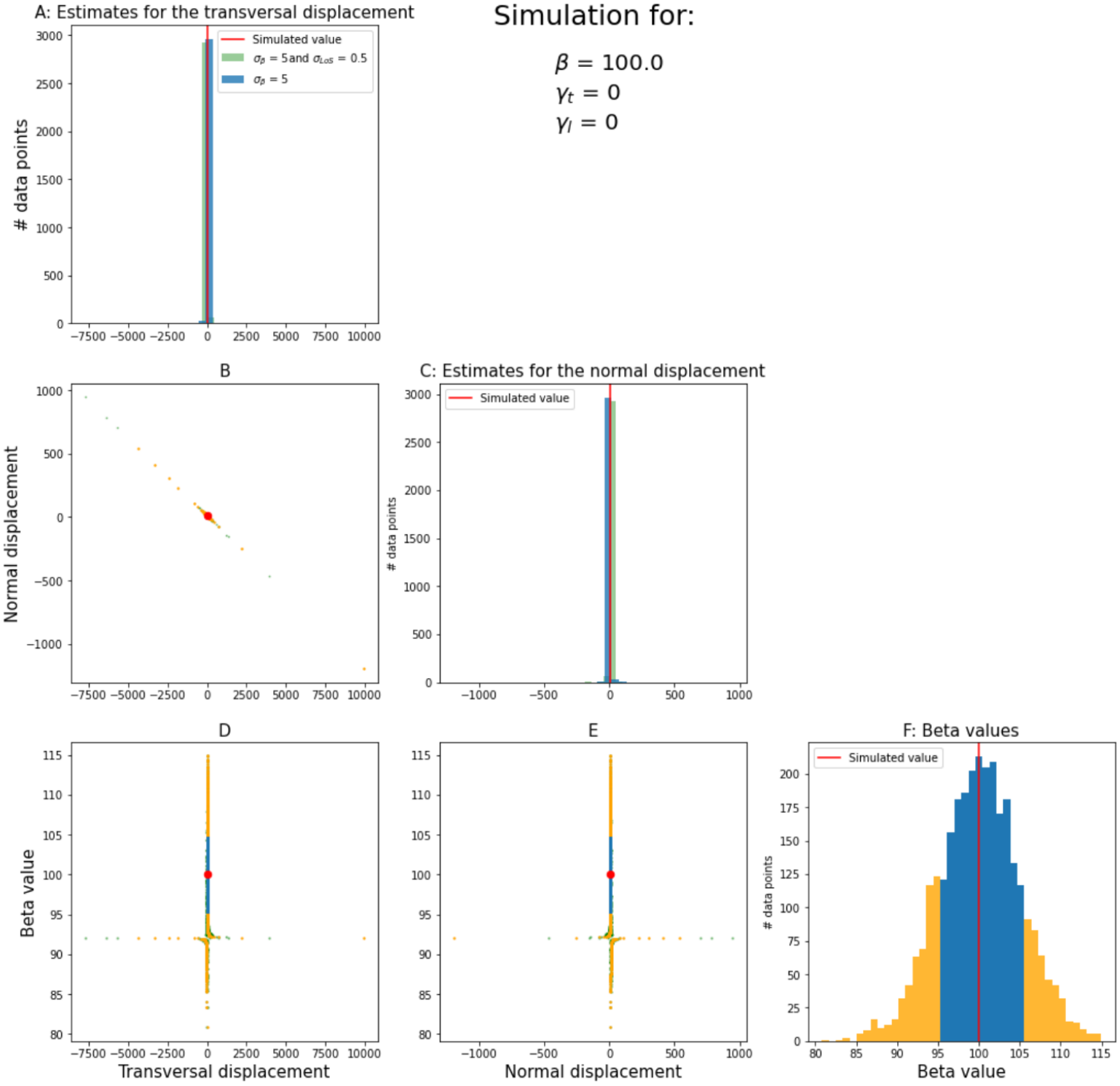
$\beta = 90.0$

$\gamma_t = 0$

$\gamma_l = 0$

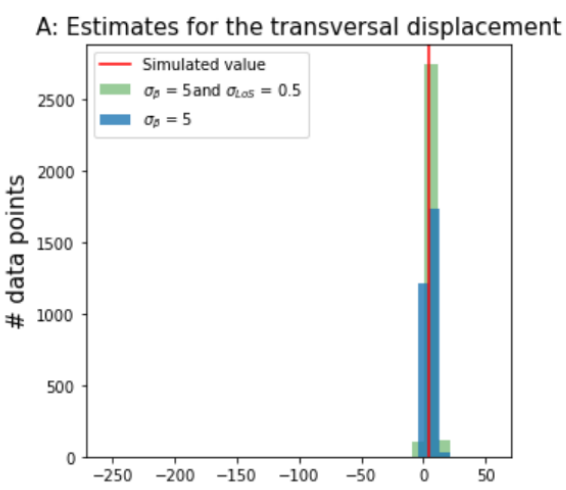






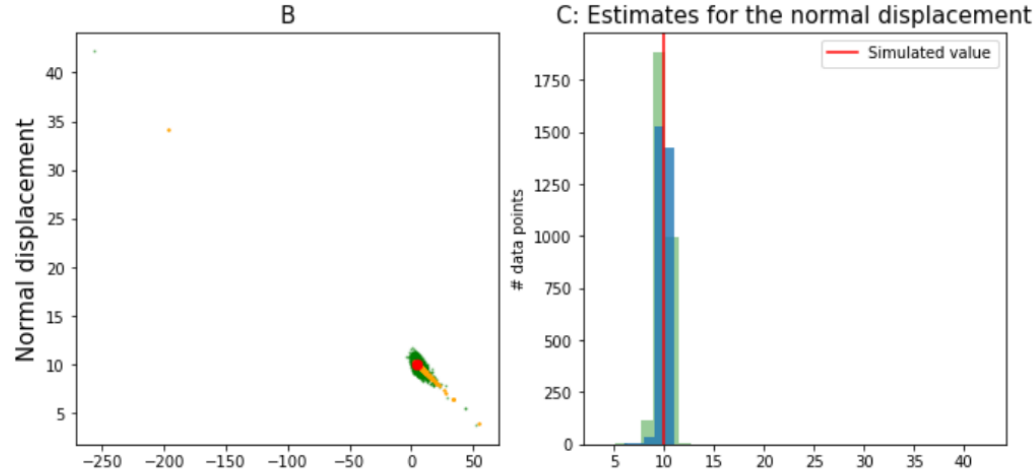
Simulation for:

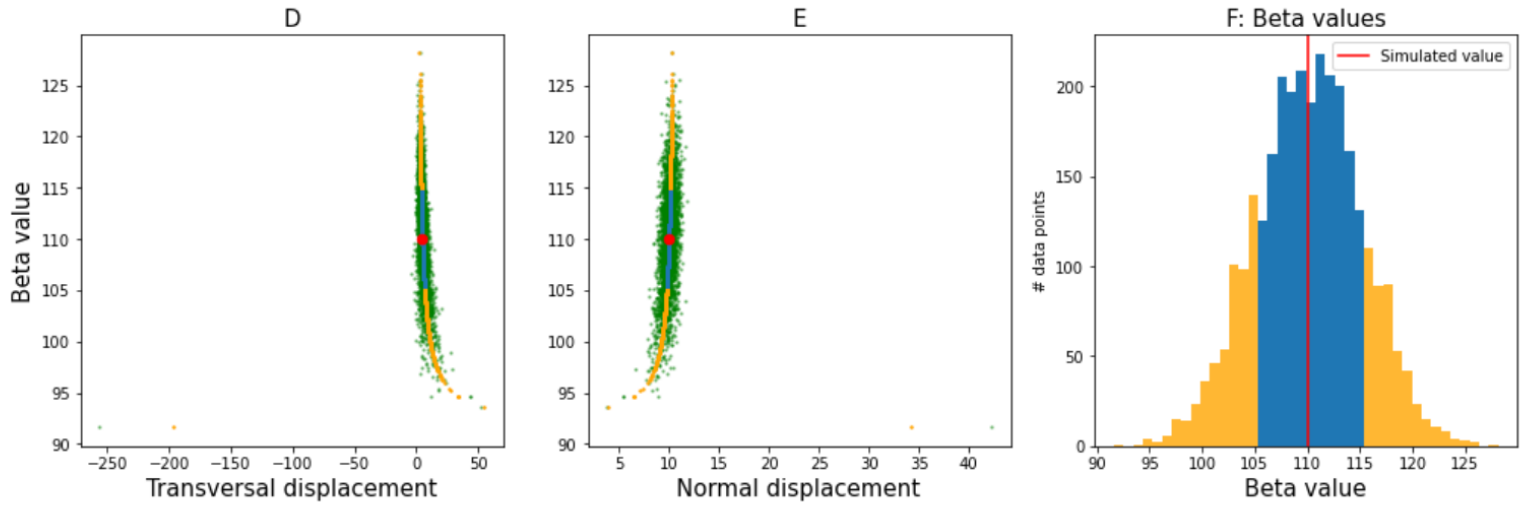
$\beta = 100.0$
$\gamma_t = 0$
$\gamma_l = 0$

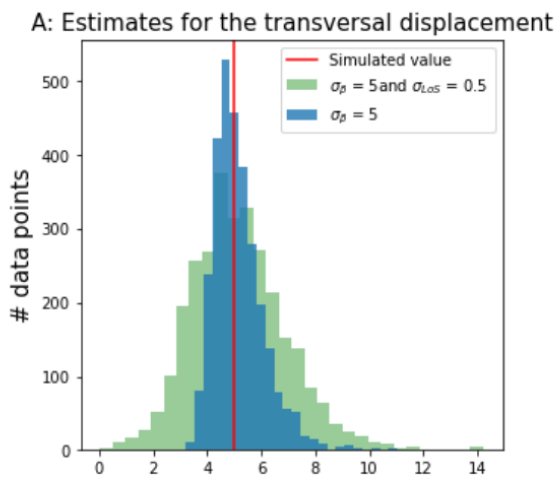


Simulation for:

$\beta = 110.0$
$\gamma_t = 0$
$\gamma_l = 0$

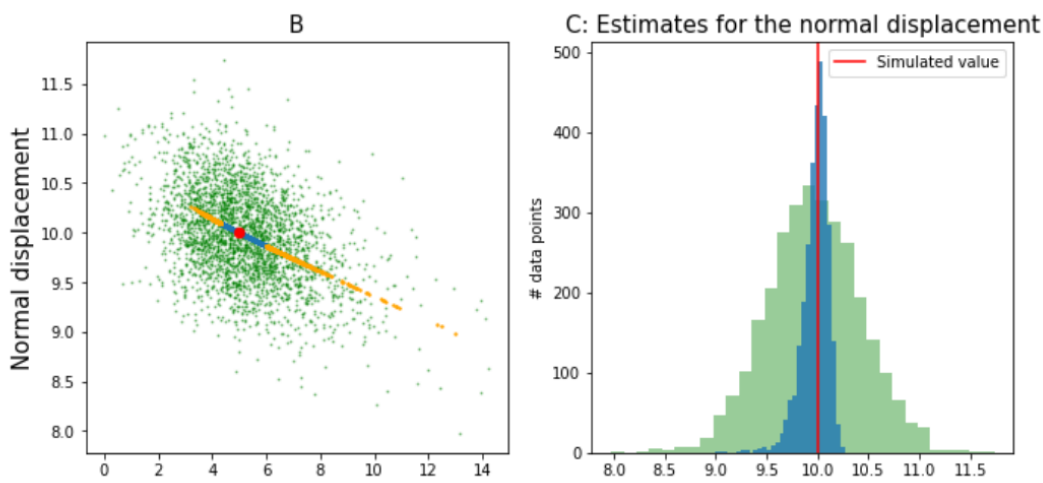


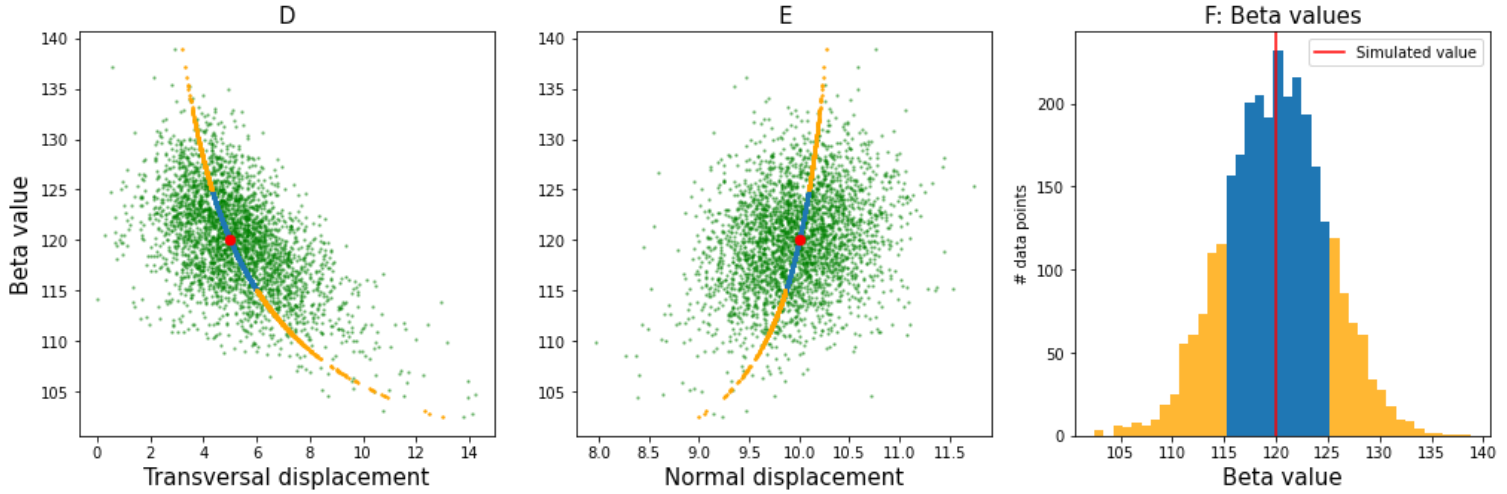


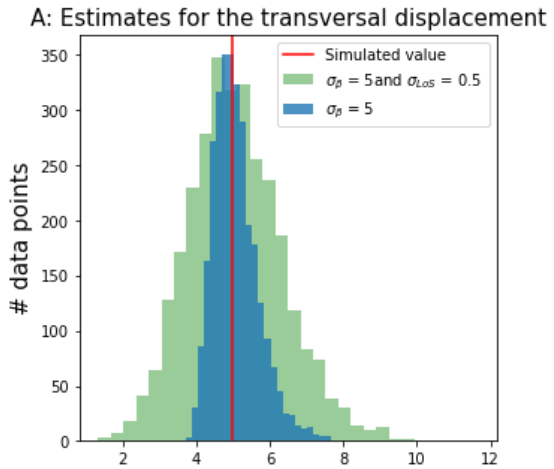


Simulation for:

$\beta = 120.0$
$\gamma_t = 0$
$\gamma_l = 0$





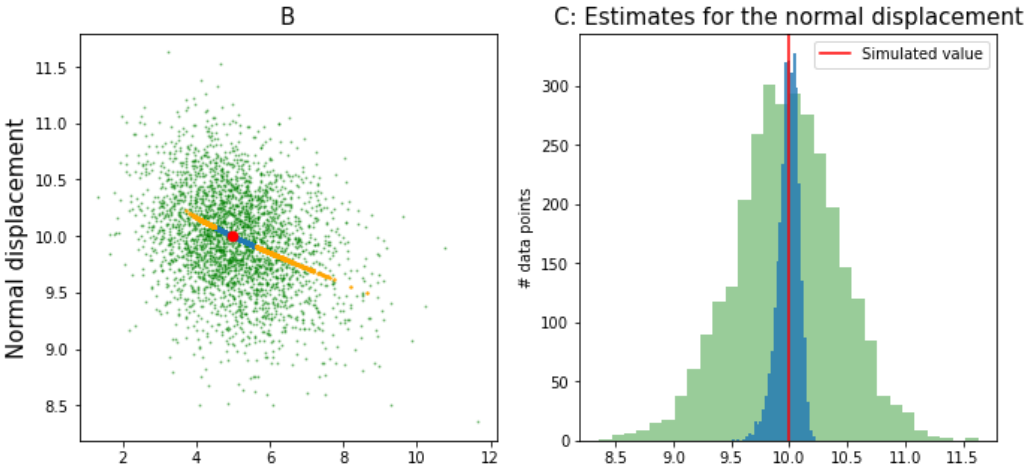


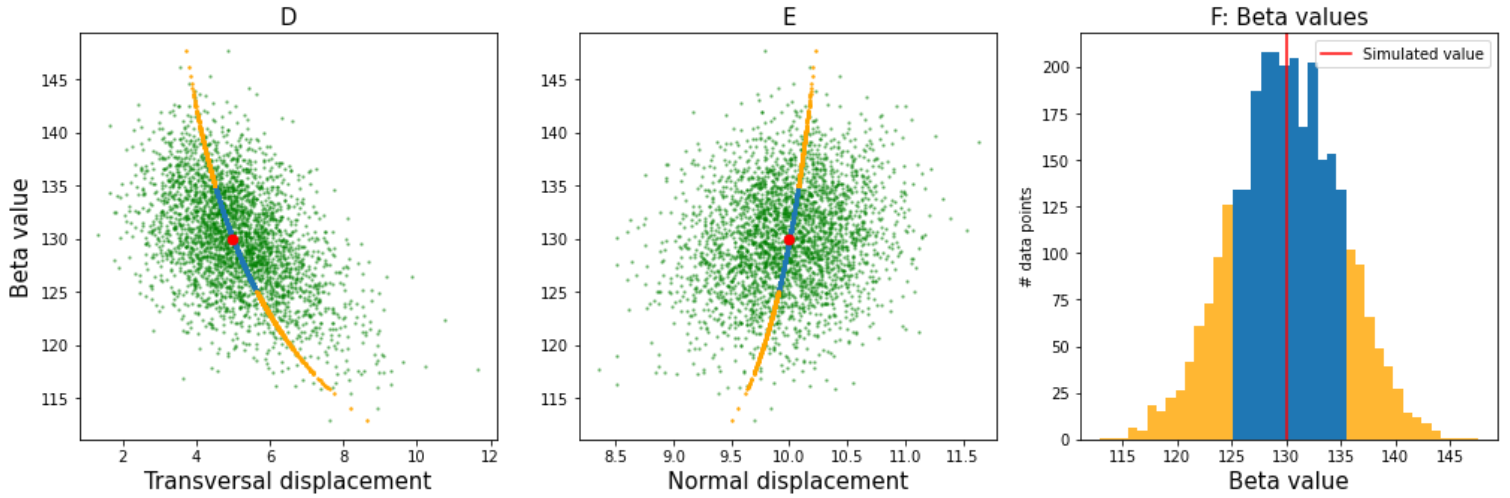
Simulation for:

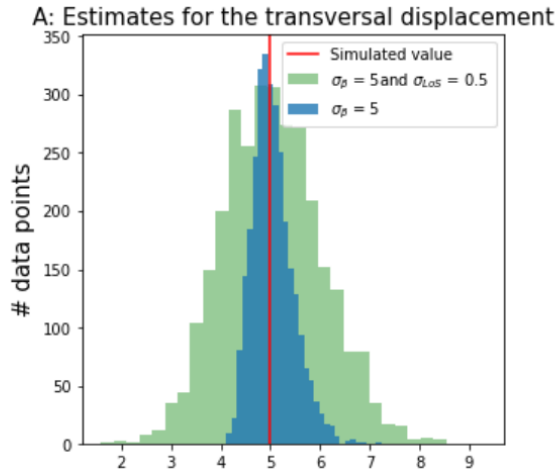
$\beta = 130.0$

$\gamma_t = 0$

$\gamma_l = 0$

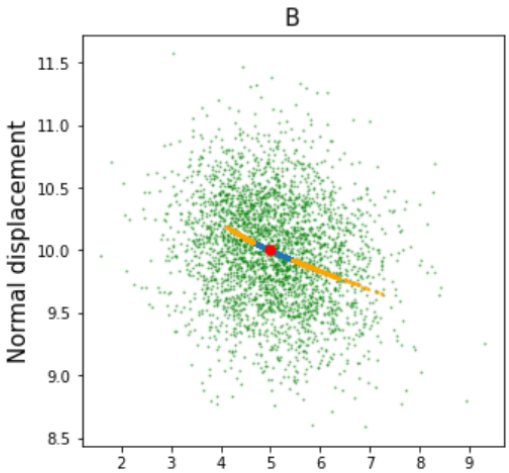


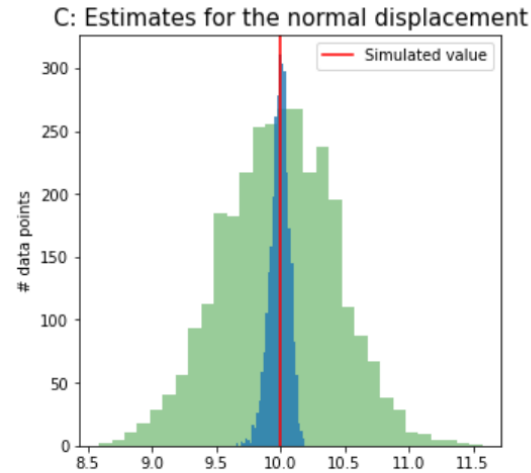


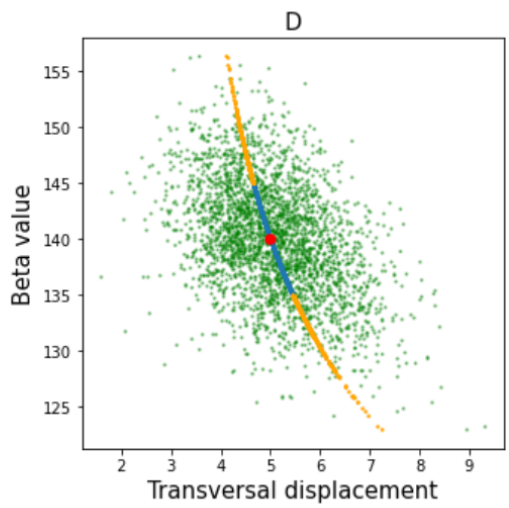


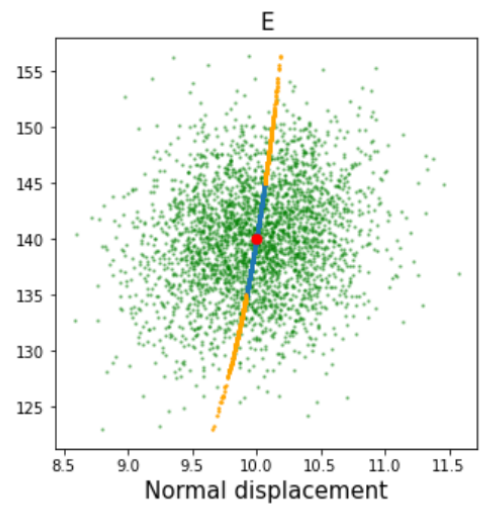
Simulation for:

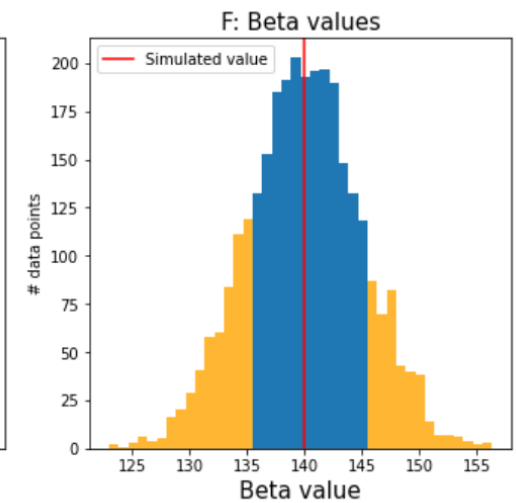
$\beta = 140.0$
$\gamma_t = 0$
$\gamma_l = 0$

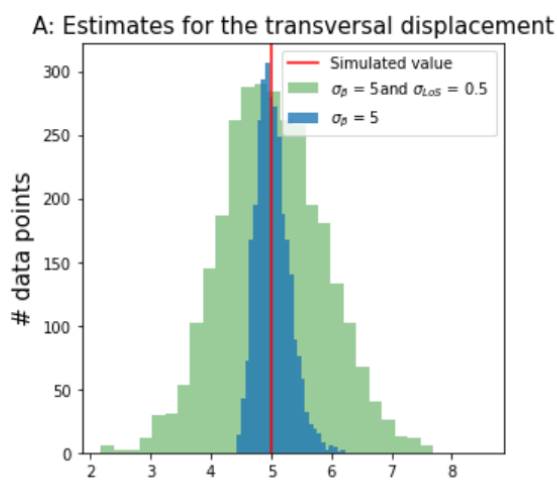






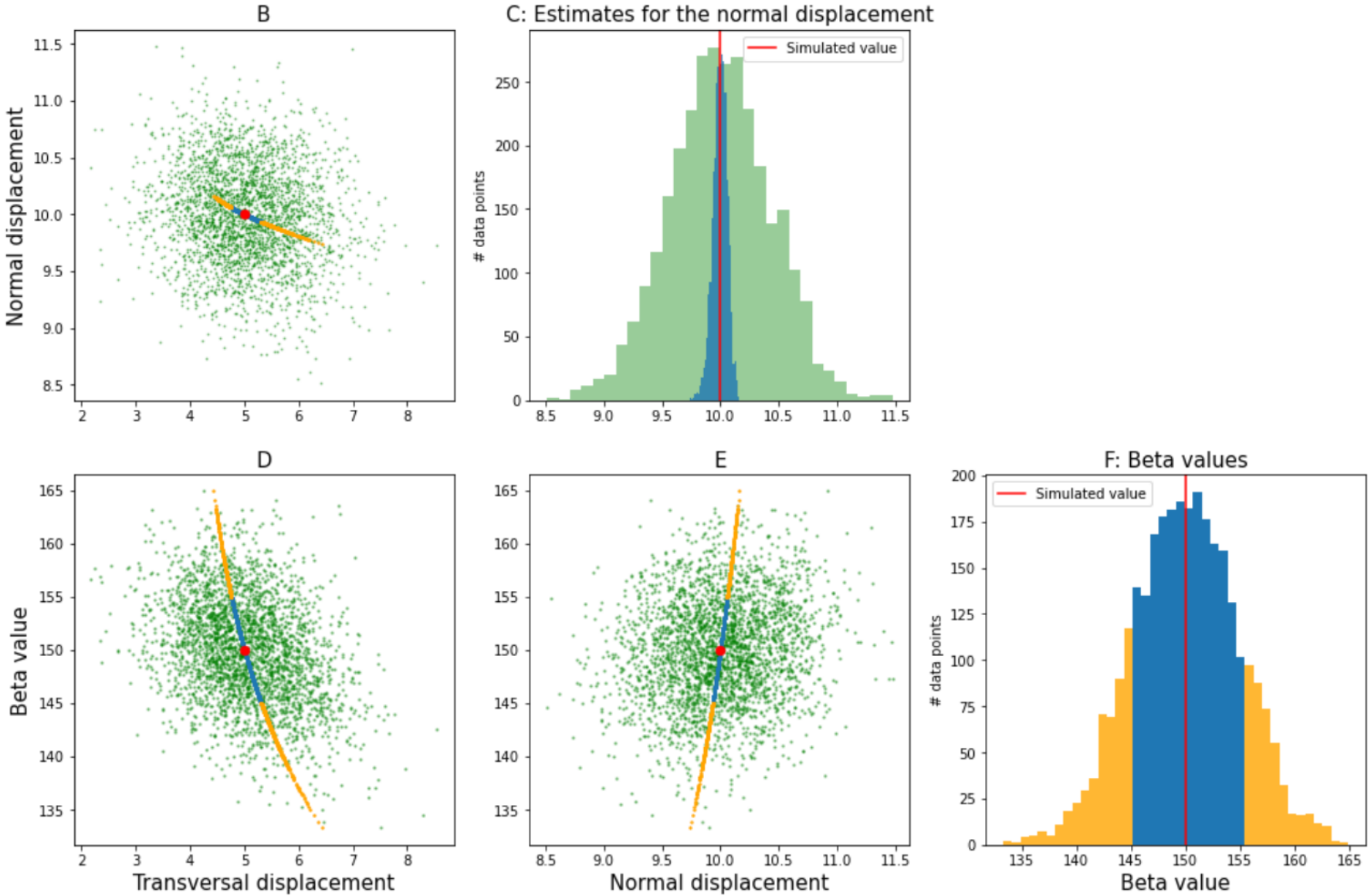


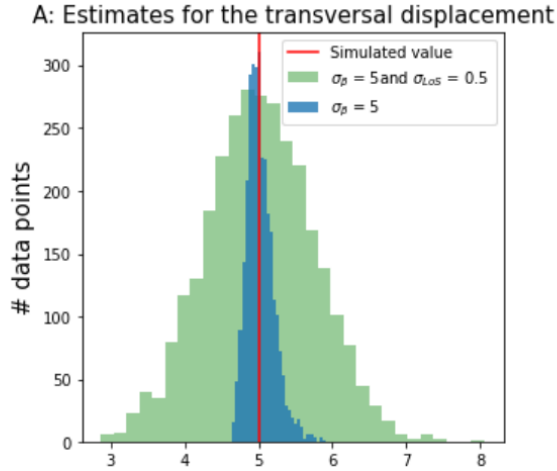




Simulation for:

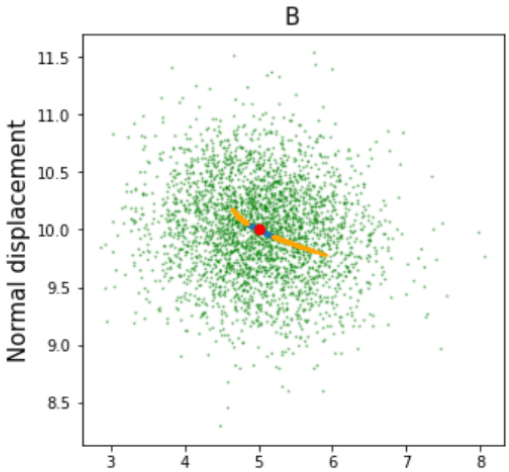
$\beta = 150.0$
$\gamma_t = 0$
$\gamma_l = 0$

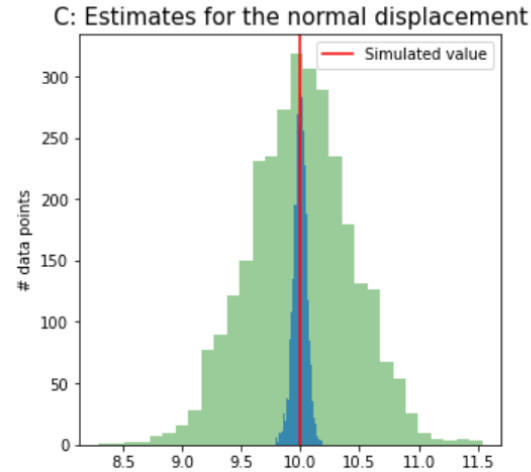

B
Normal displacement
C: Estimates for the normal displacement
Simulated value
data points
D
Beta value
Transversal displacement
E
Normal displacement
F: Beta values
Simulated value
data points
Beta value

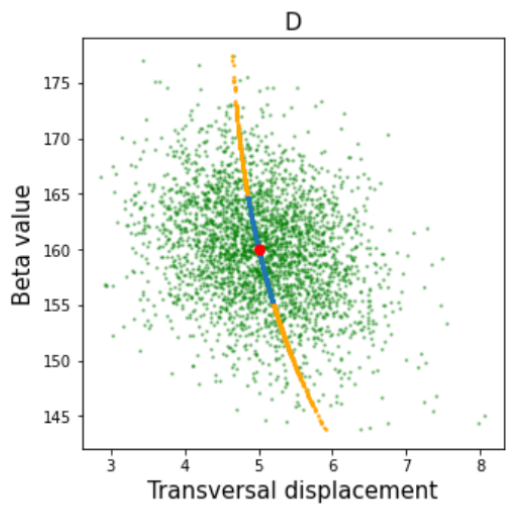


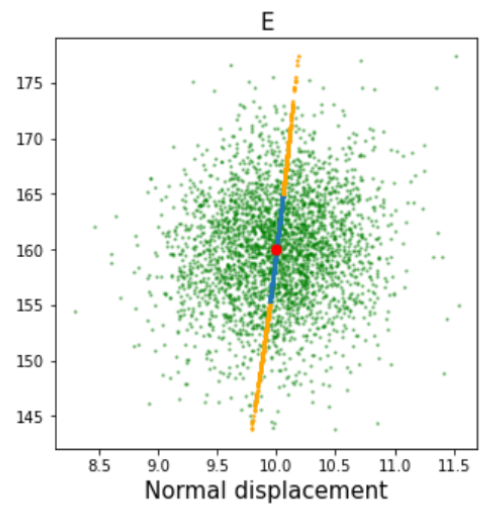
Simulation for:

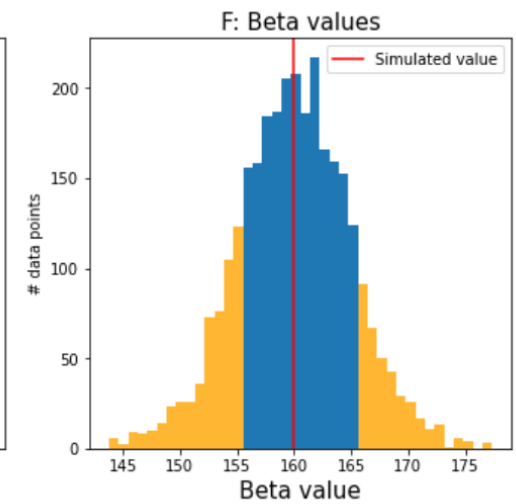
$\beta = 160.0$
$\gamma_t = 0$
$\gamma_l = 0$

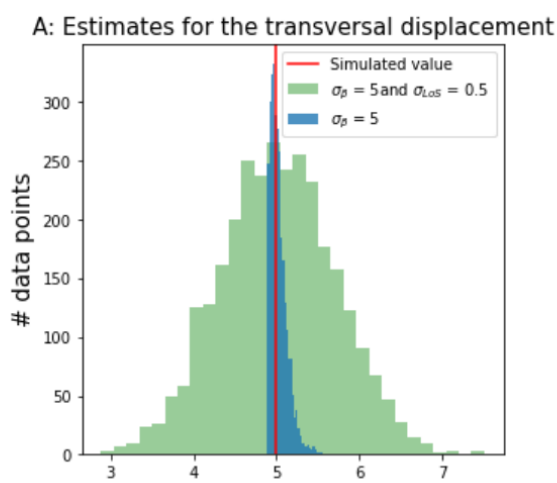






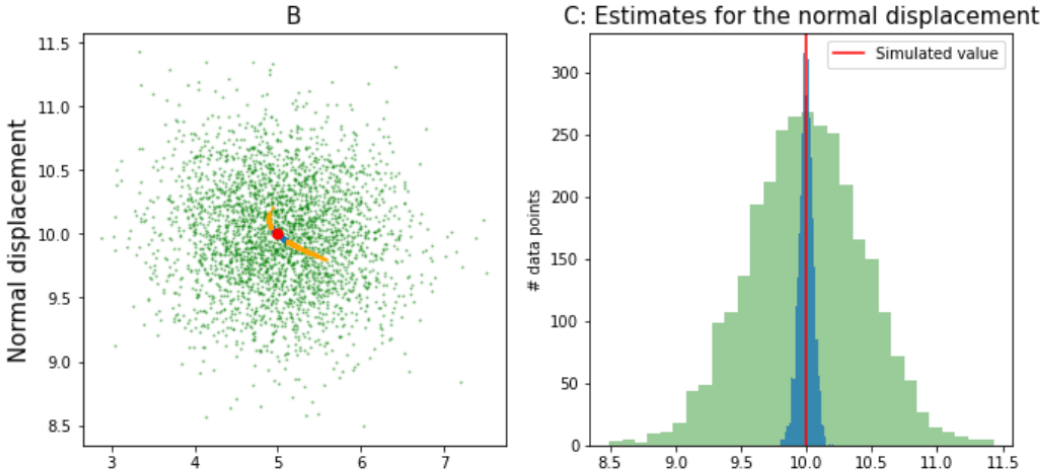


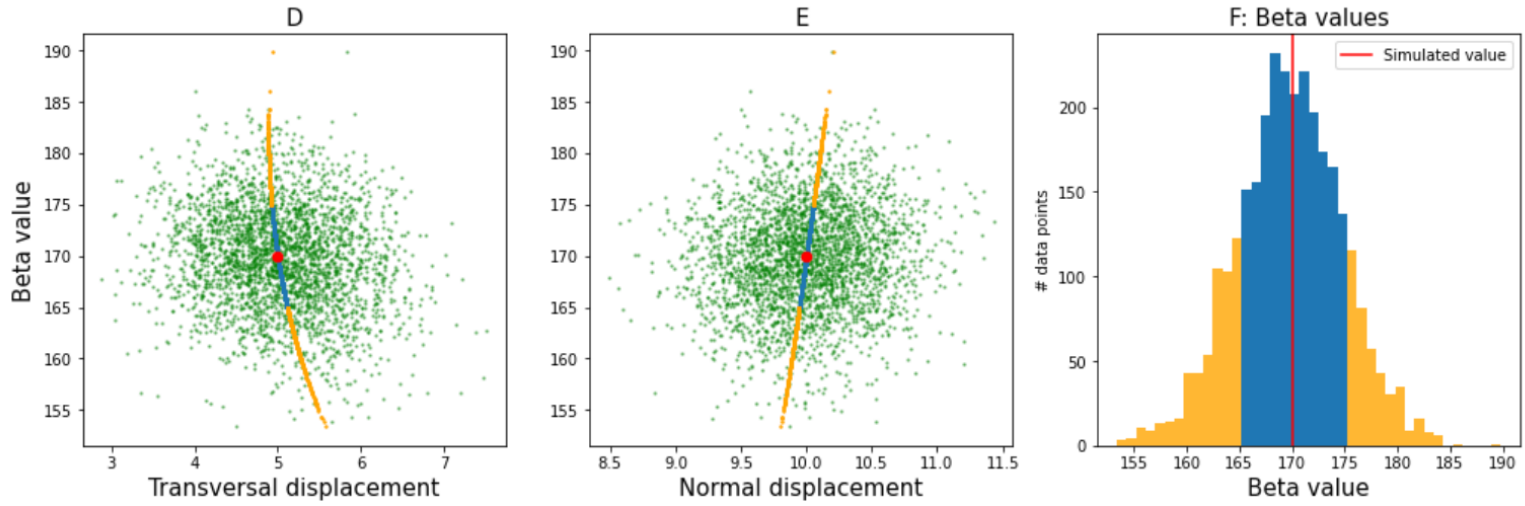


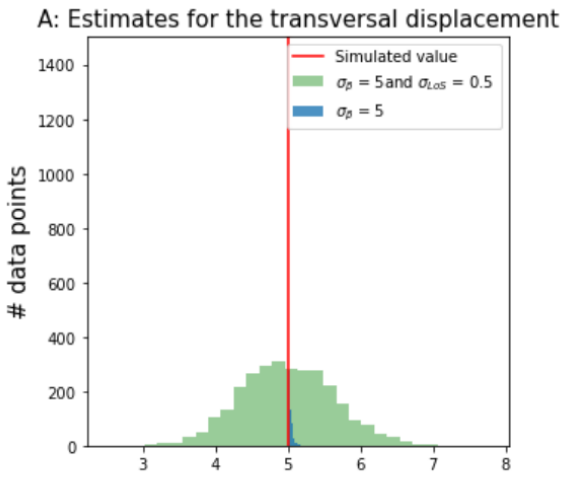


Simulation for:

$\beta = 170.0$
$\gamma_t = 0$
$\gamma_l = 0$





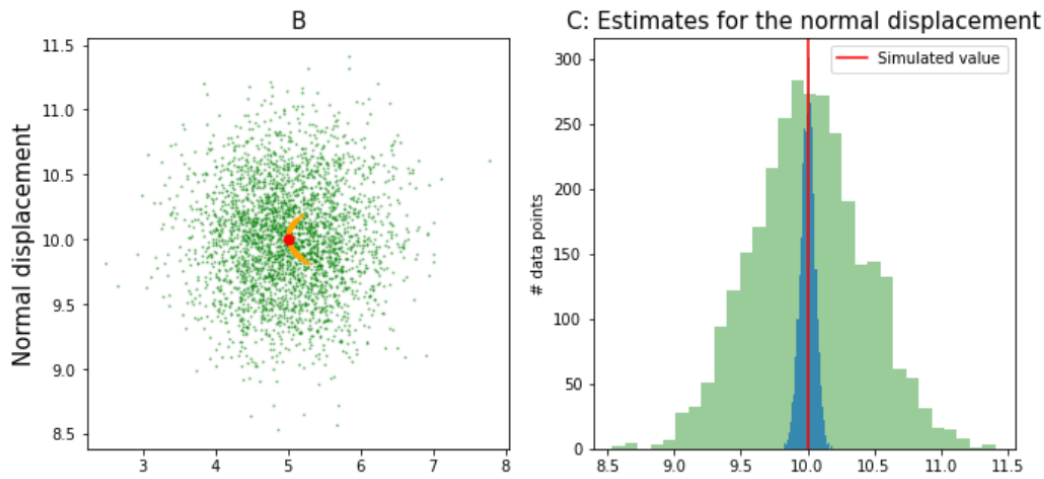


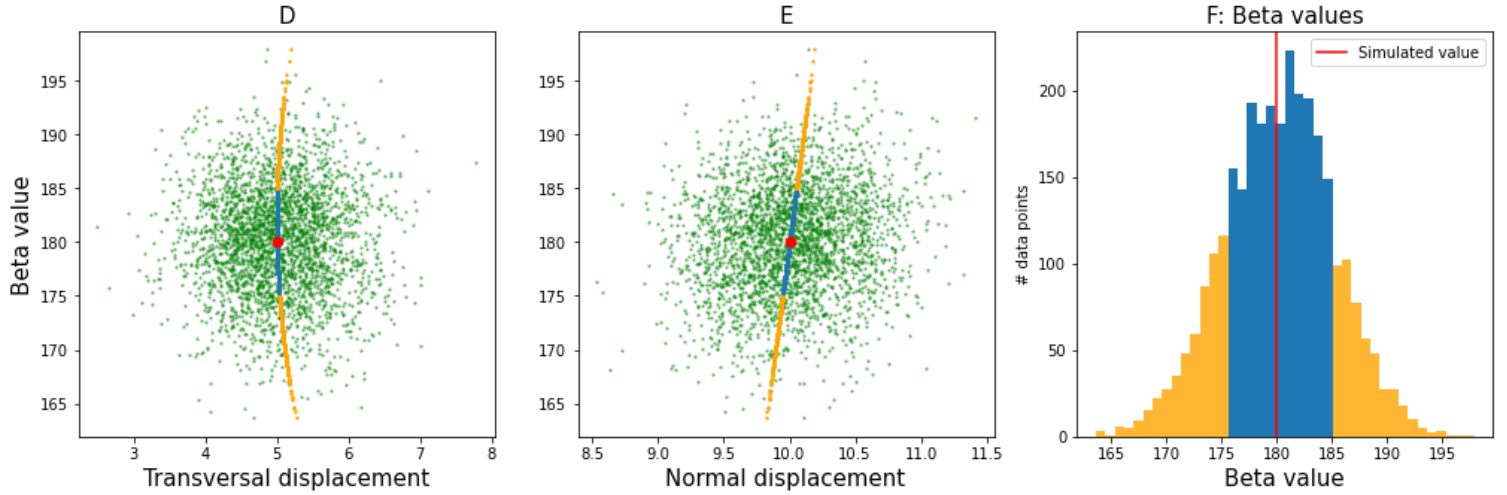
Simulation for:

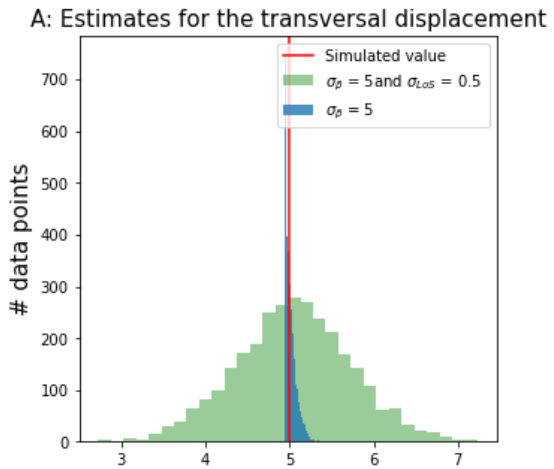
$\beta = 180.0$

$\gamma_t = 0$

$\gamma_l = 0$





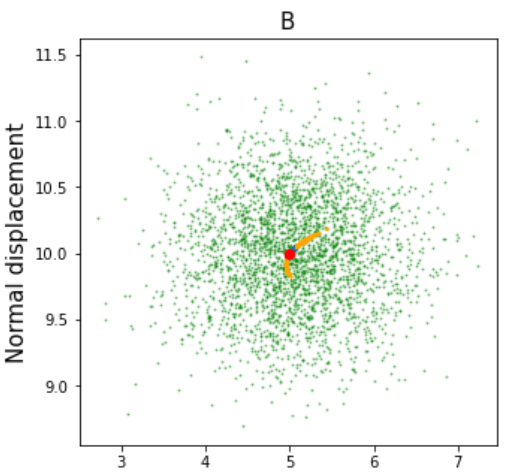


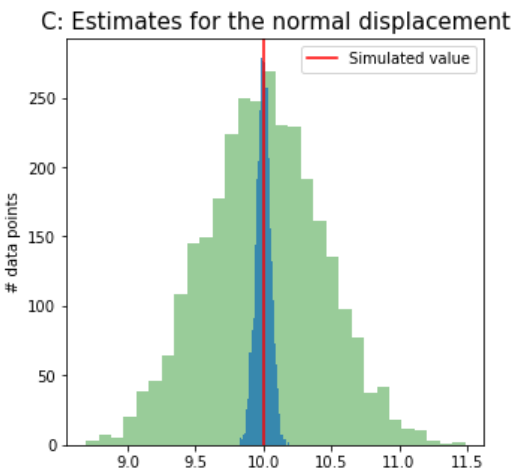
Simulation for:

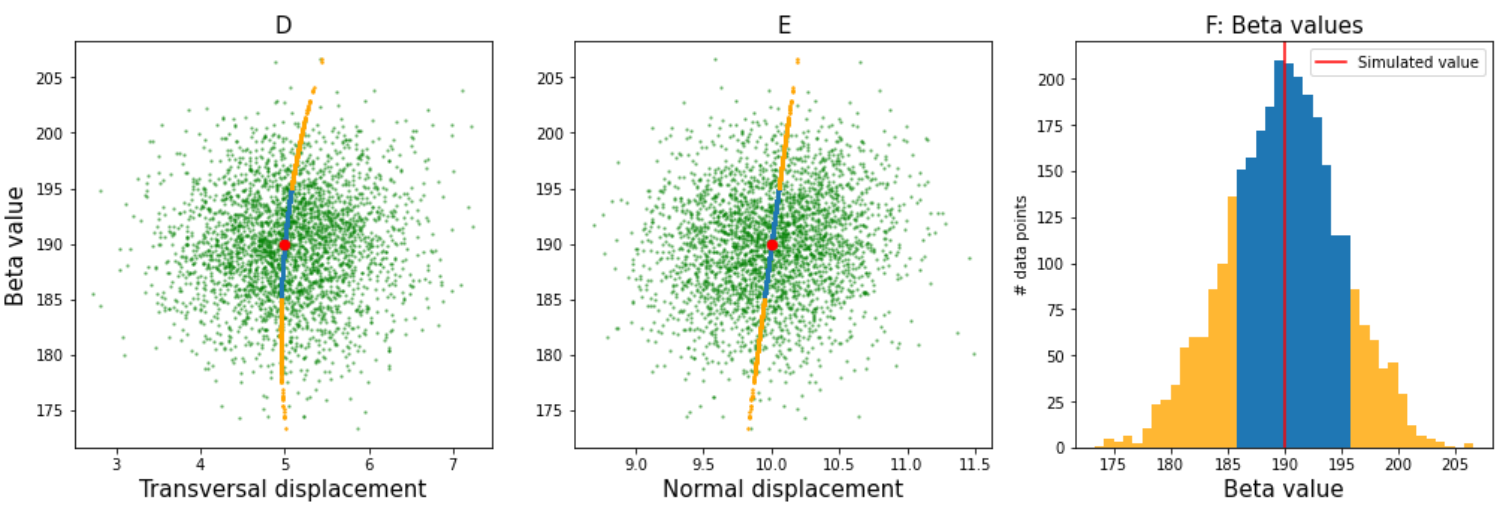
$\beta = 190.0$

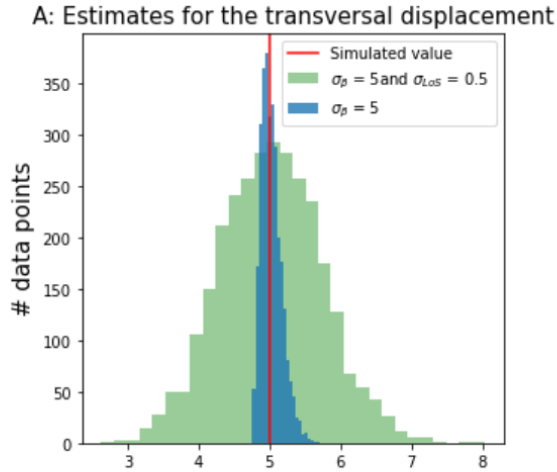
$\gamma_t = 0$

$\gamma_l = 0$



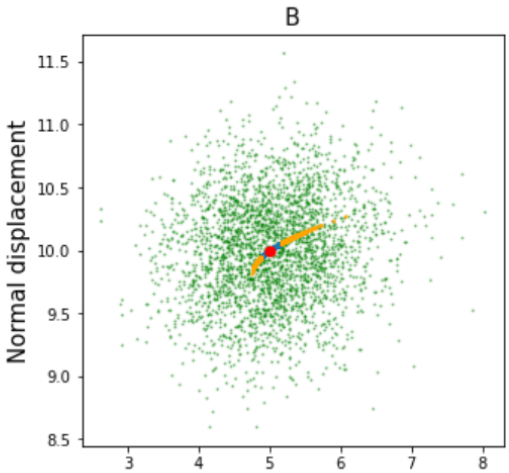


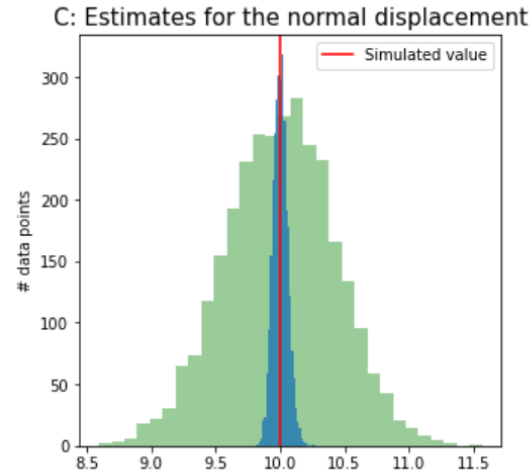


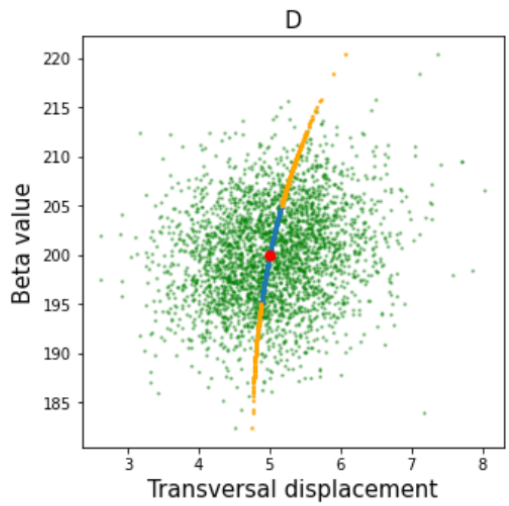


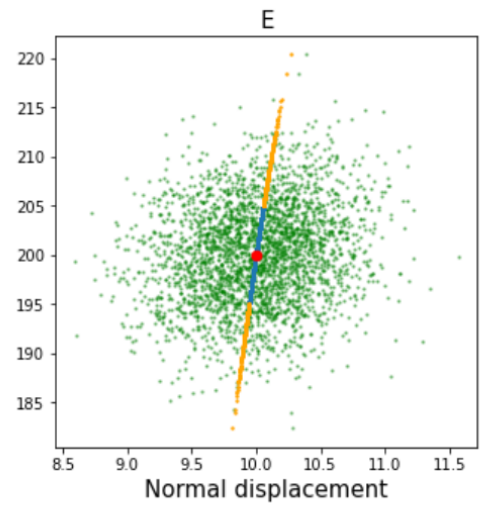
Simulation for:

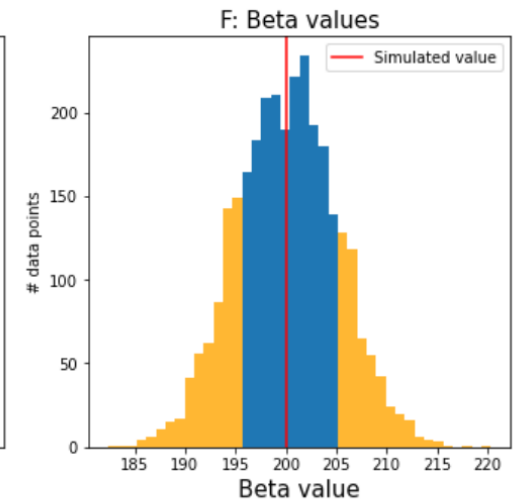
$\beta = 200.0$
$\gamma_t = 0$
$\gamma_l = 0$

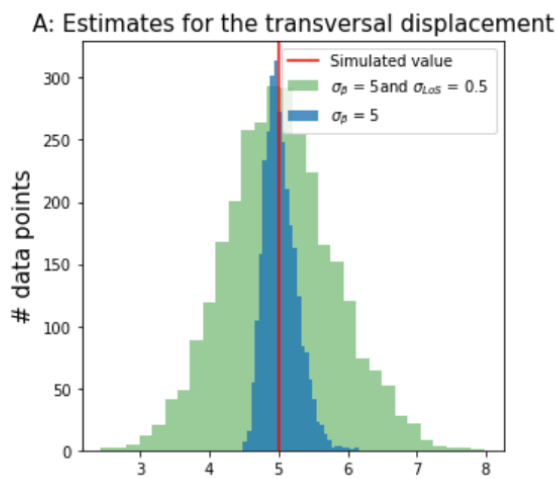






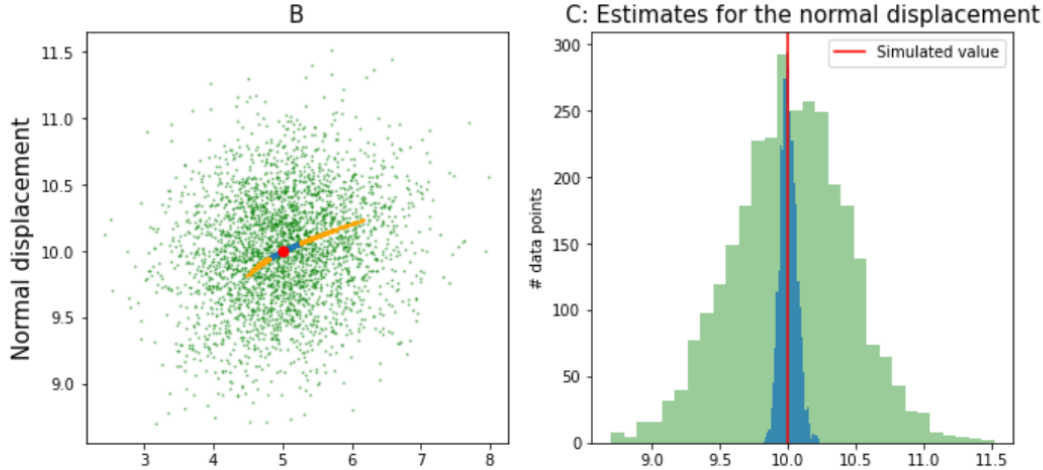


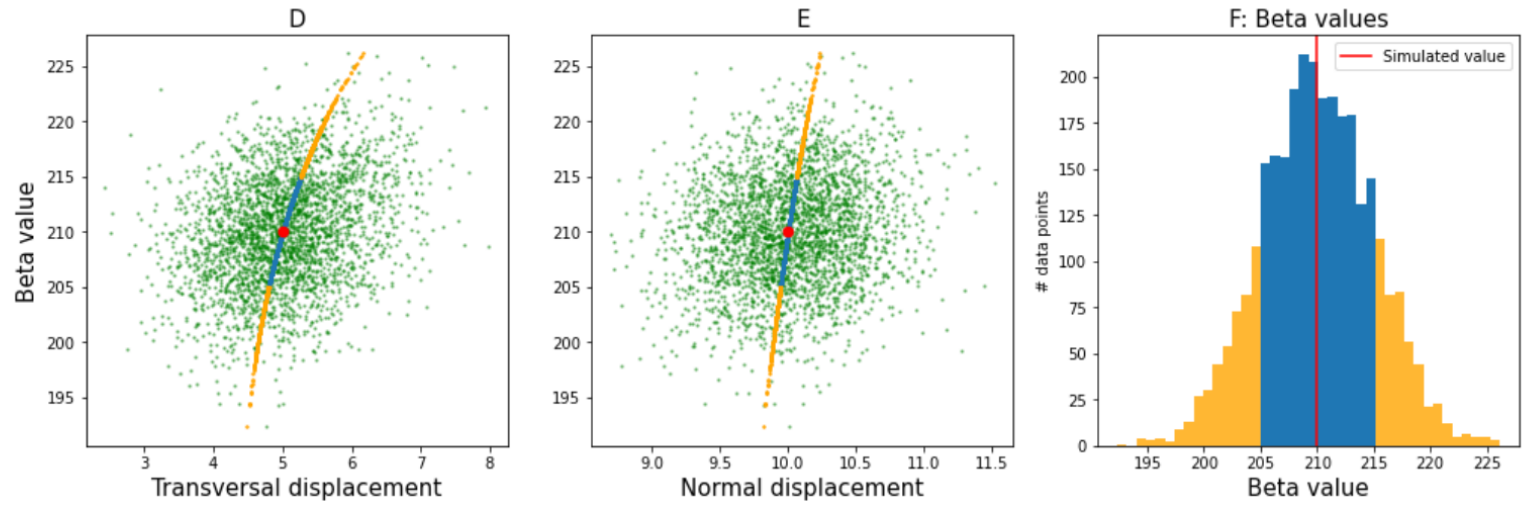


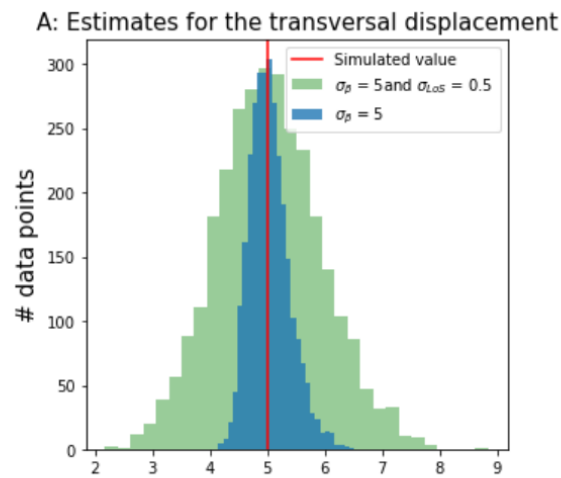


Simulation for:

$\beta = 210.0$
$\gamma_t = 0$
$\gamma_l = 0$





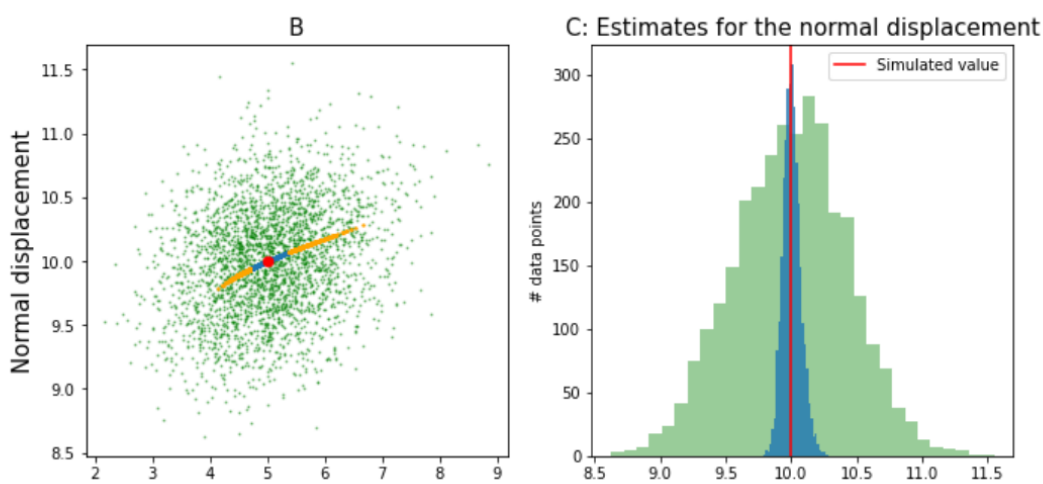


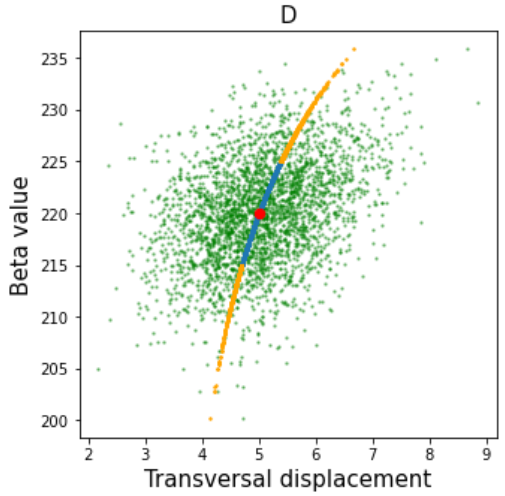
Simulation for:

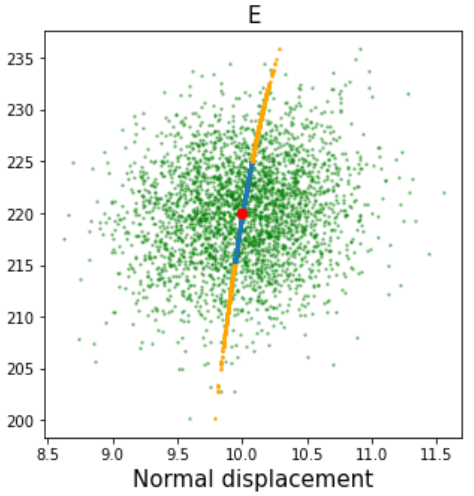
$\beta = 220.0$

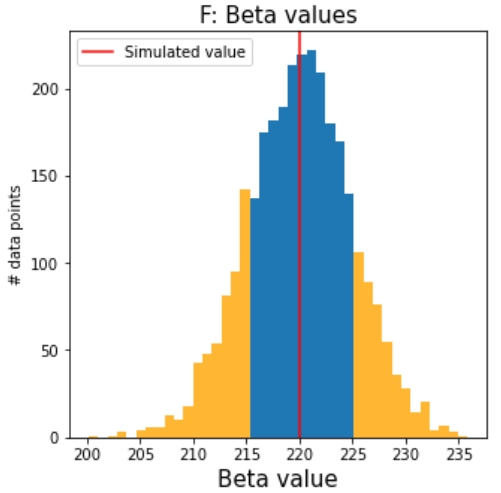
$\gamma_t = 0$

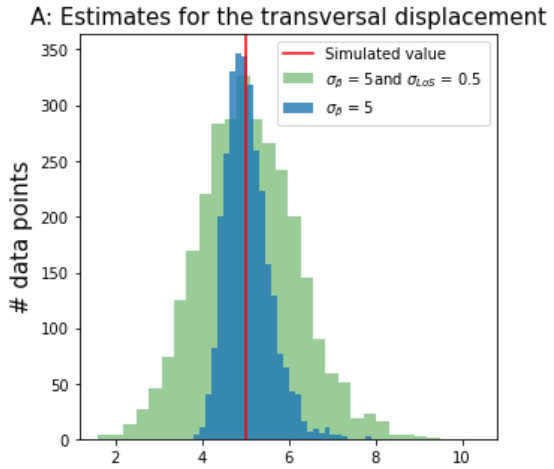
$\gamma_l = 0$





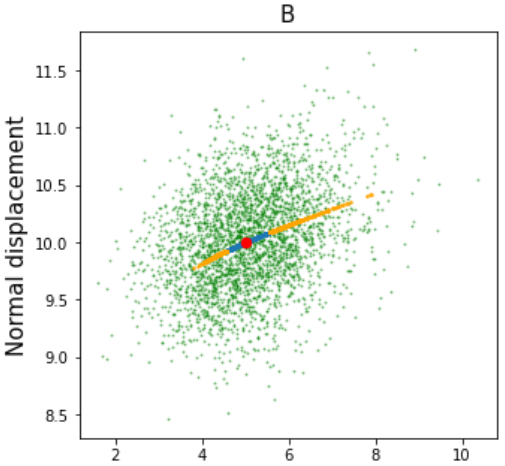


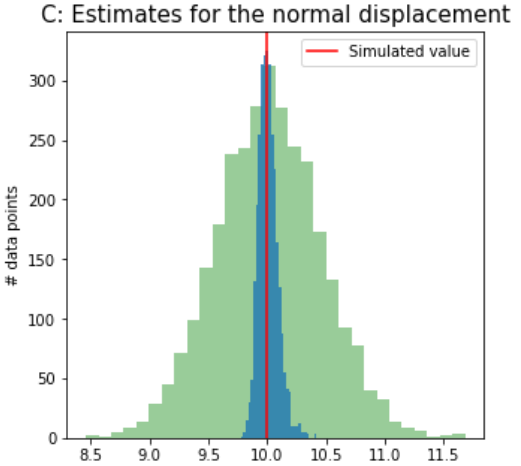


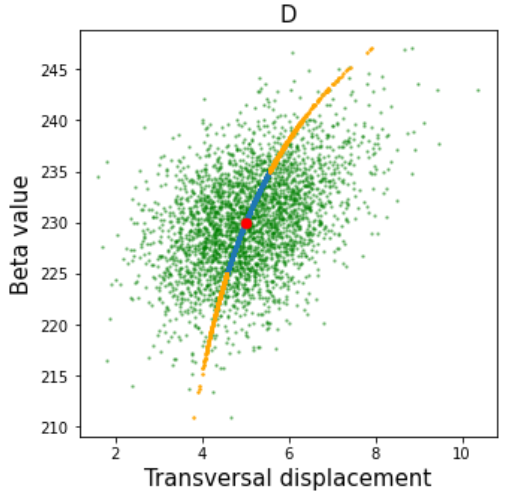


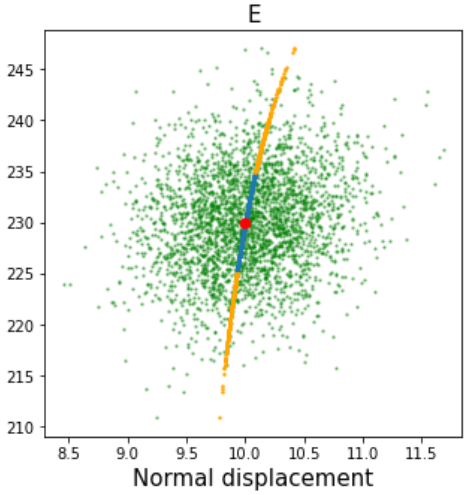
Simulation for:

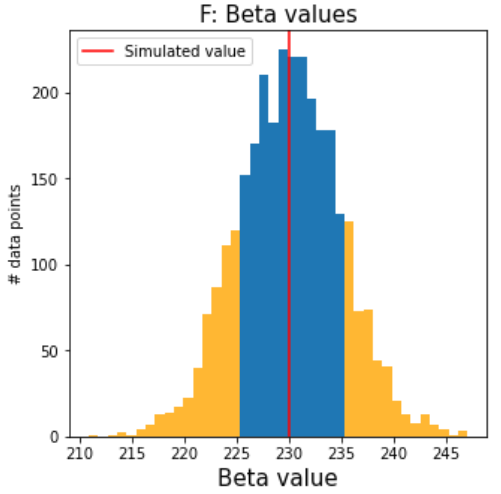
$\beta = 230.0$
$\gamma_t = 0$
$\gamma_l = 0$

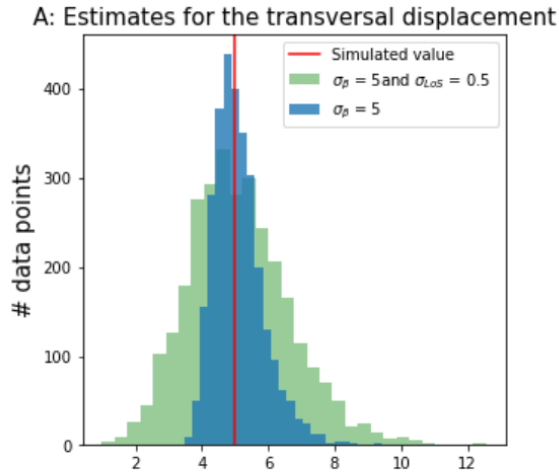






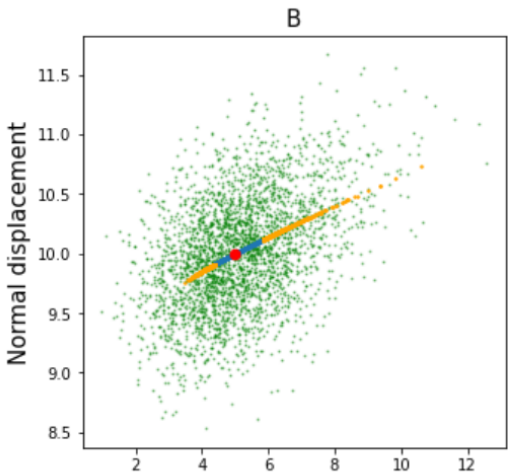


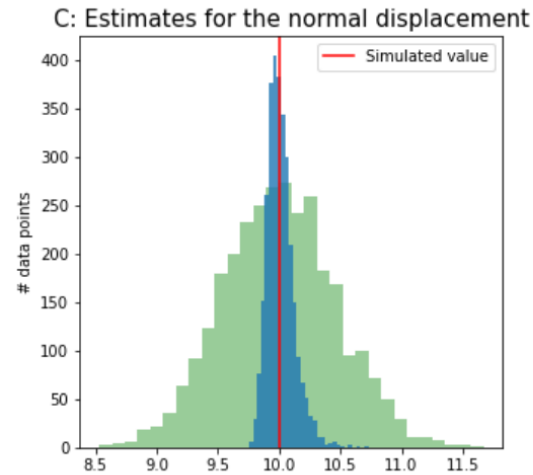


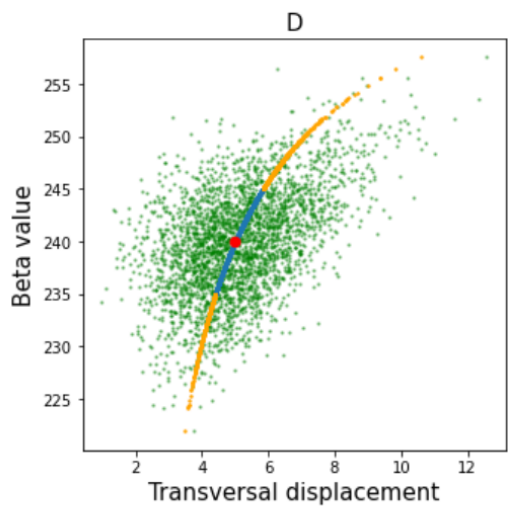


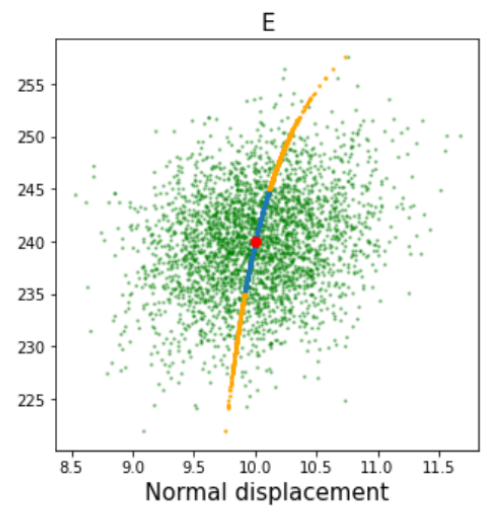
Simulation for:

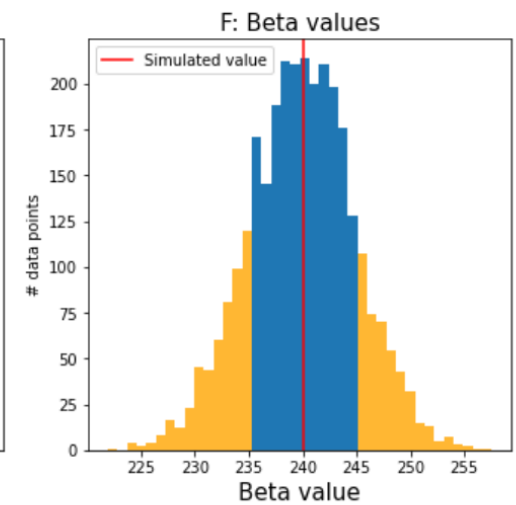
$\beta = 240.0$
$\gamma_t = 0$
$\gamma_l = 0$

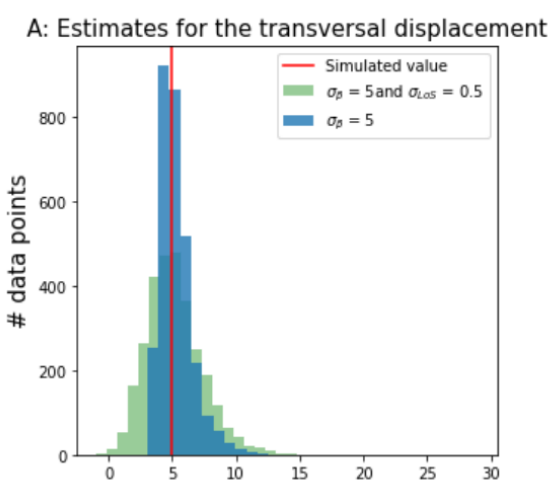






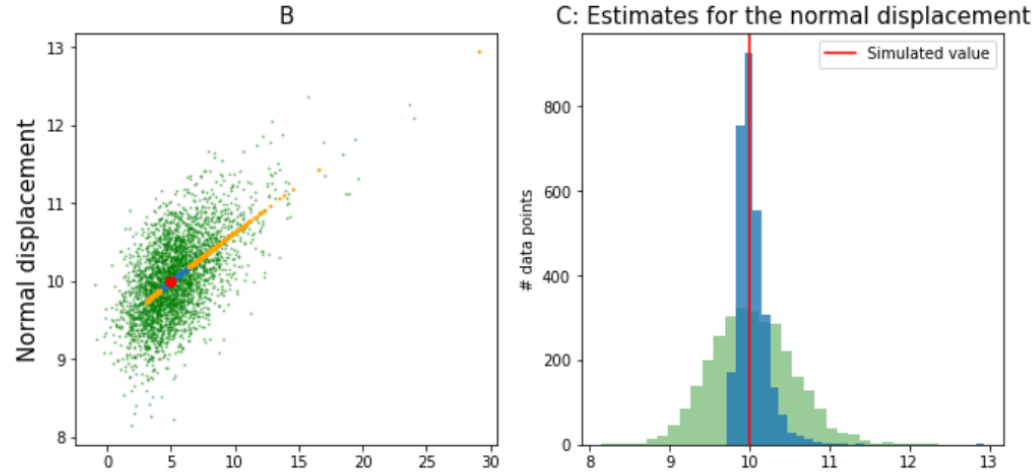


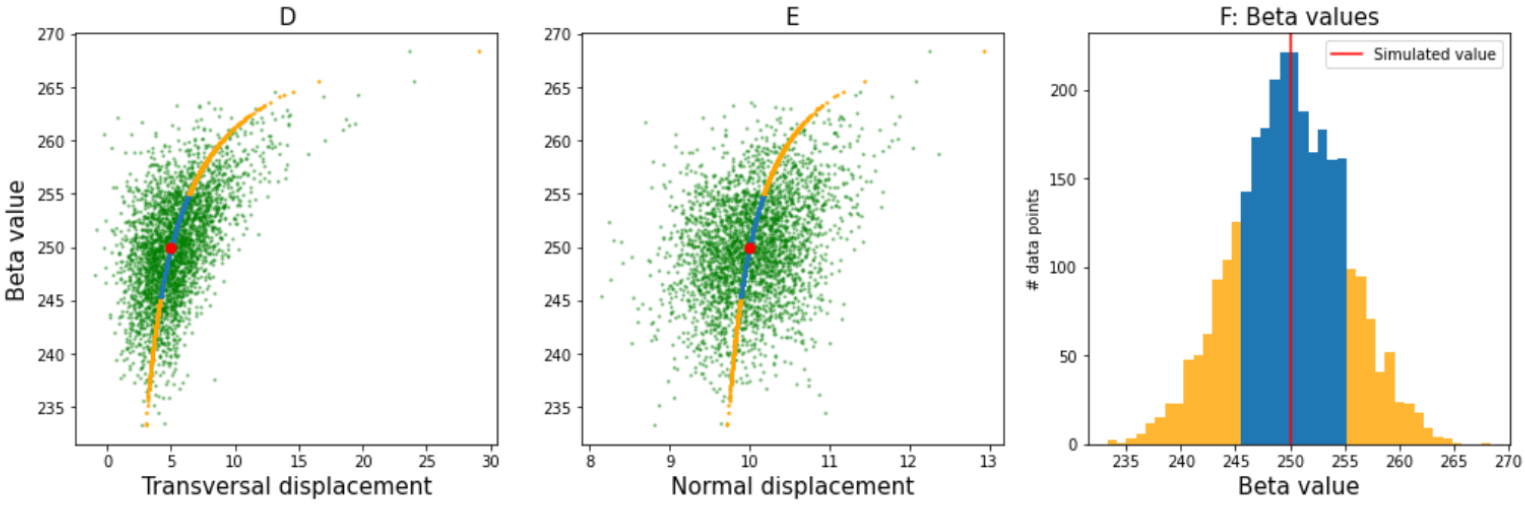


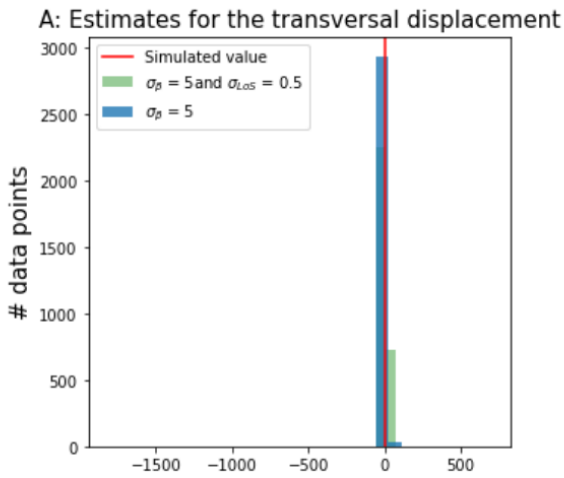


Simulation for:

$\beta = 250.0$
$\gamma_t = 0$
$\gamma_l = 0$

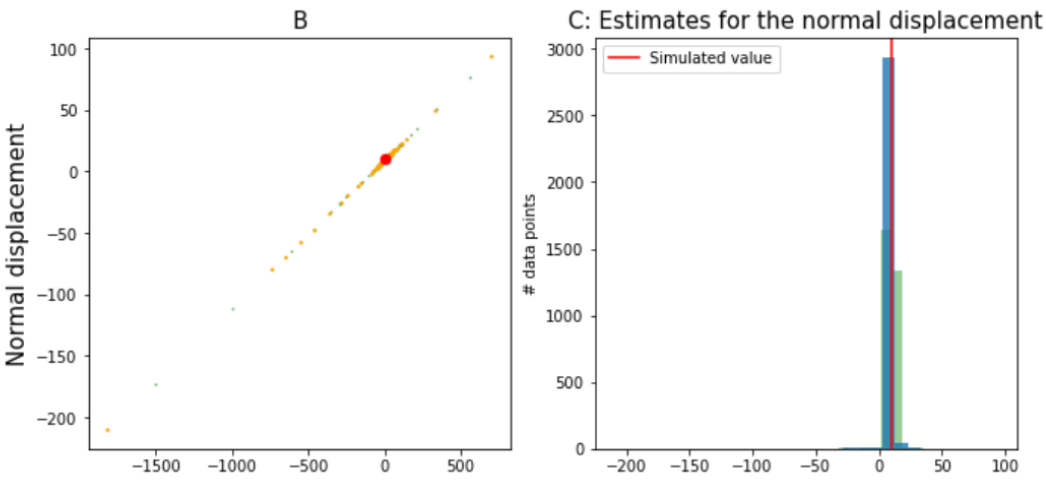


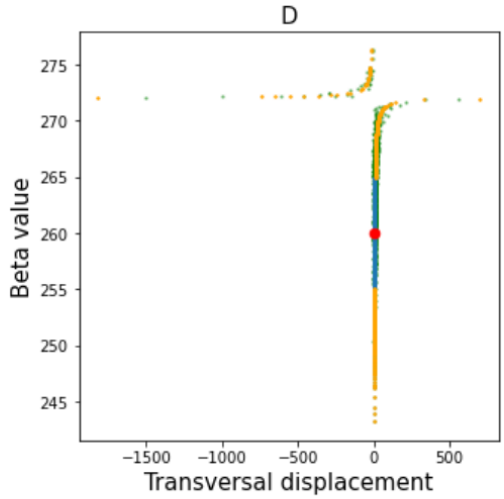


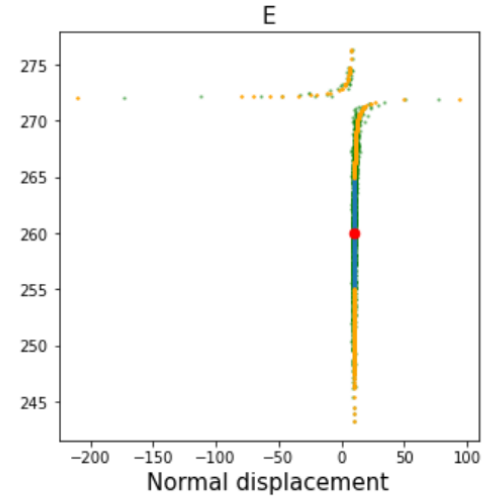


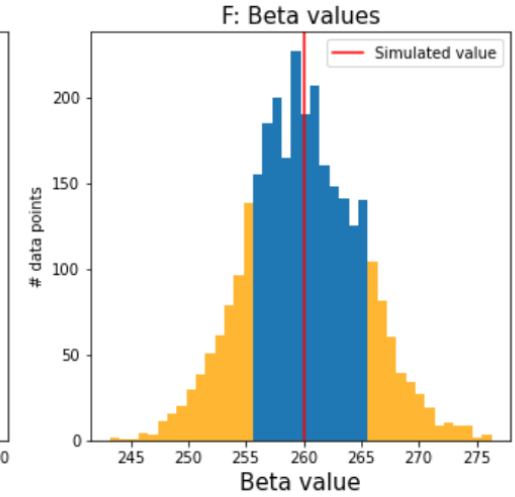
Simulation for:

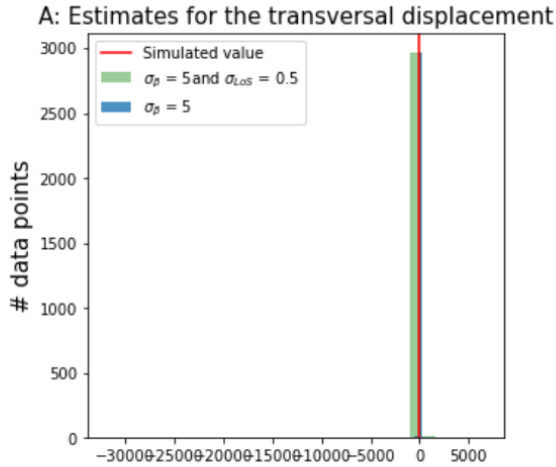
$\beta = 260.0$
$\gamma_t = 0$
$\gamma_l = 0$





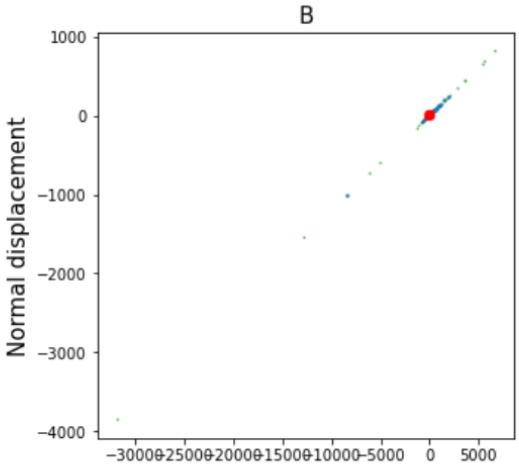


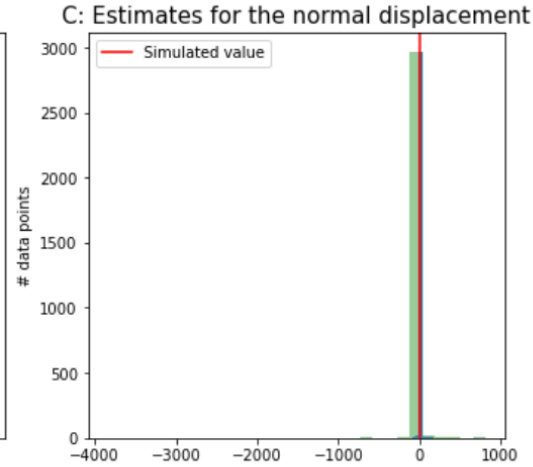


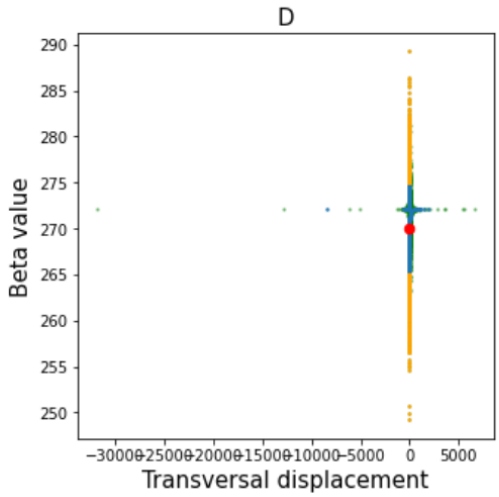


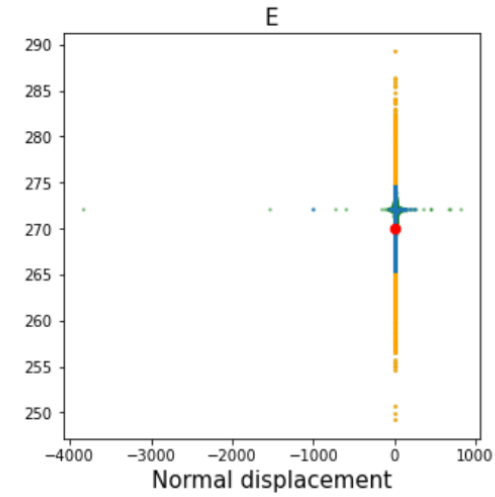
Simulation for:

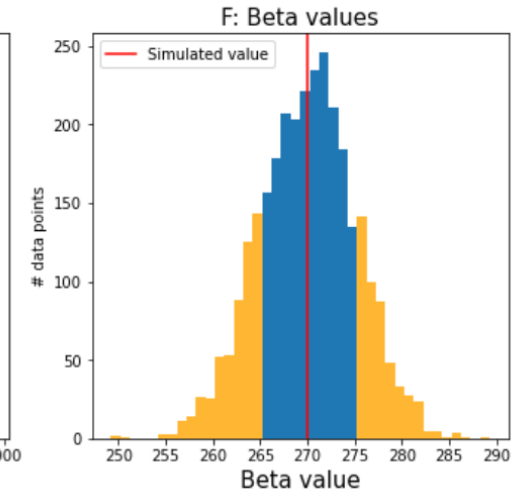
$\beta = 270.0$
$\gamma_t = 0$
$\gamma_l = 0$

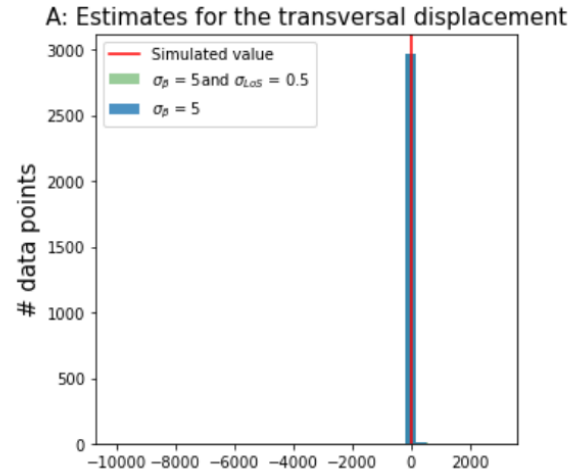






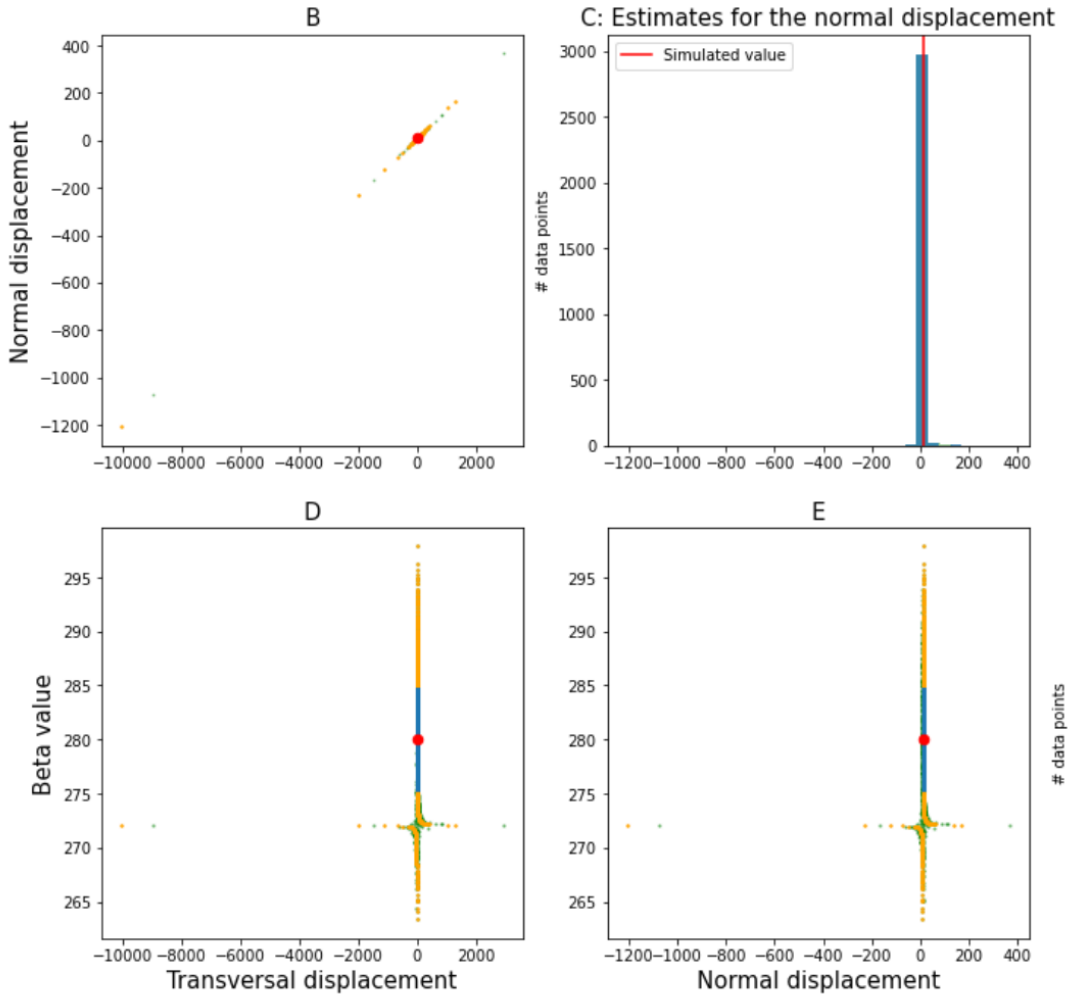


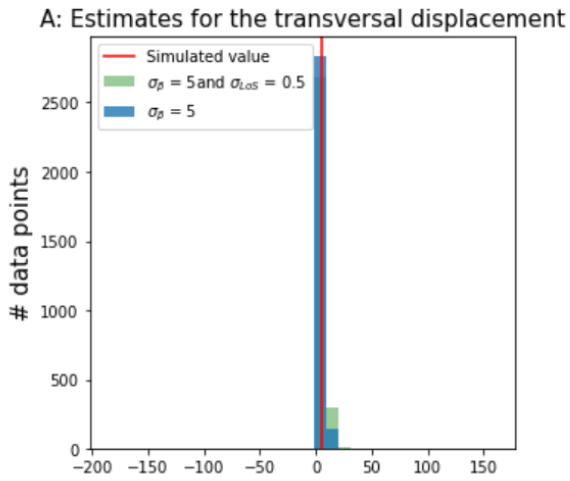




Simulation for:

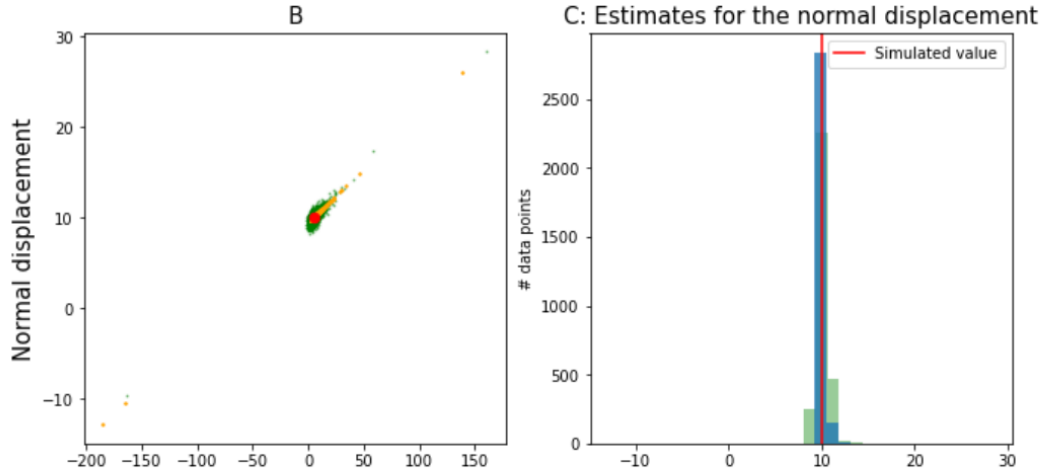
$\beta = 280.0$
$\gamma_t = 0$
$\gamma_l = 0$

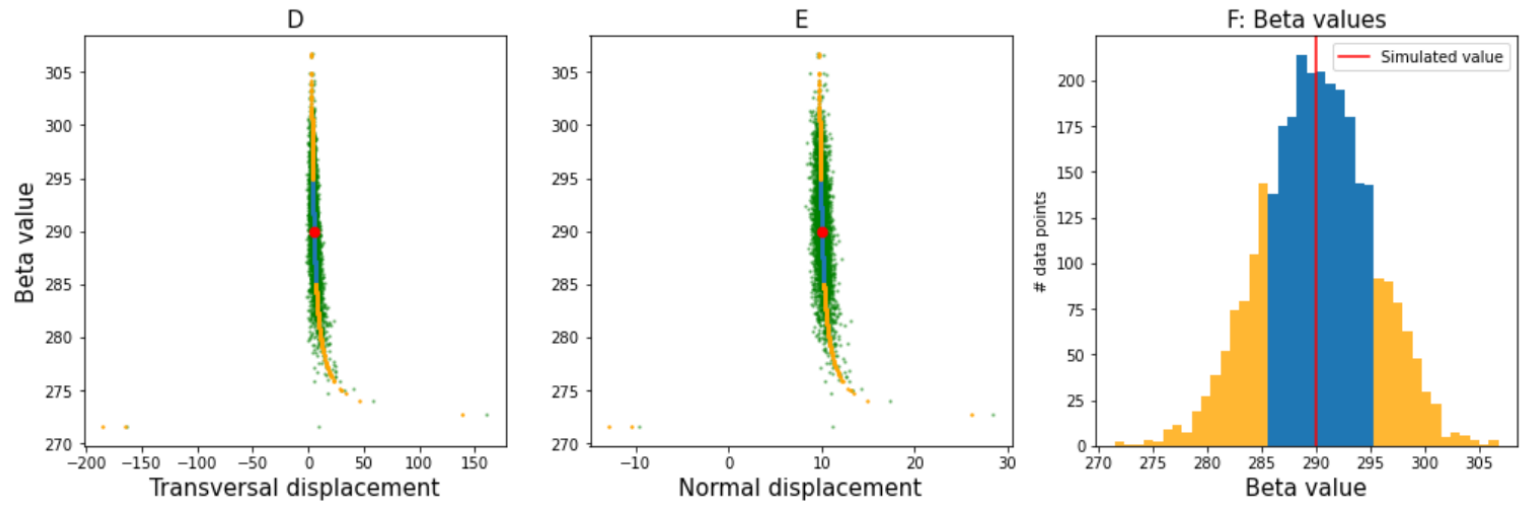


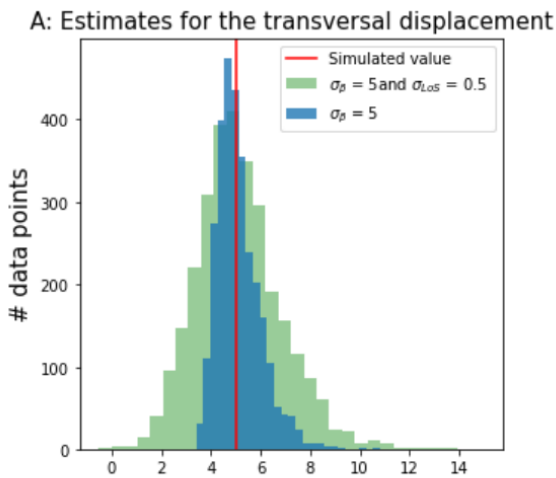


Simulation for:

$\beta = 290.0$
$\gamma_t = 0$
$\gamma_l = 0$

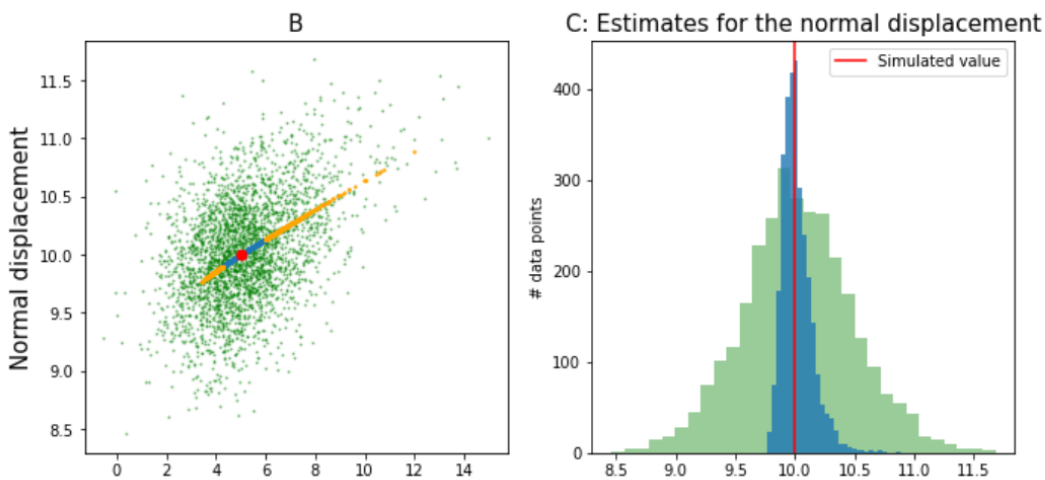


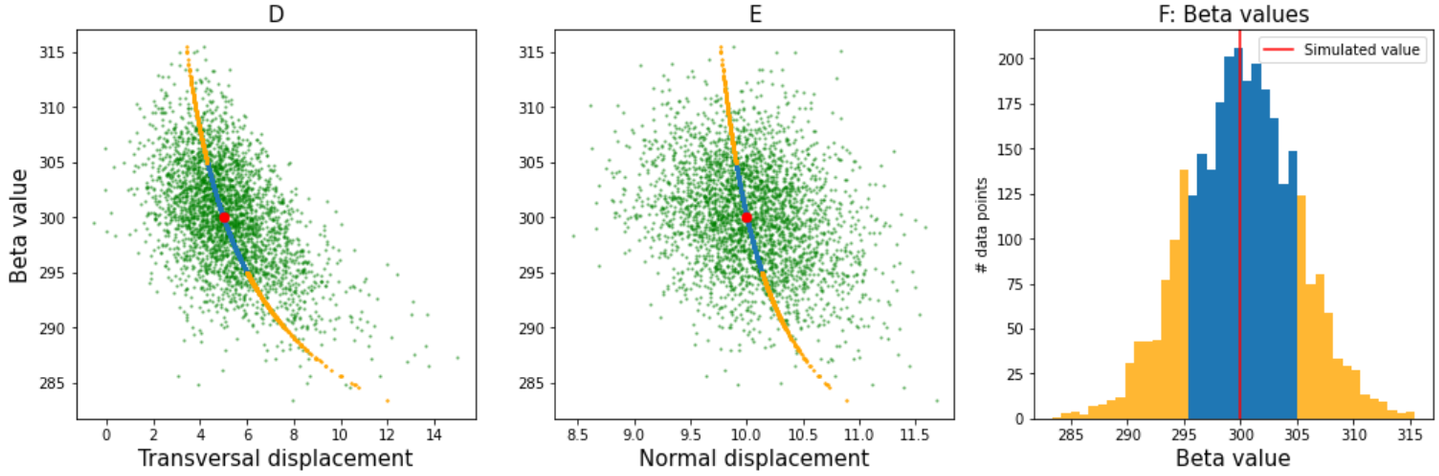


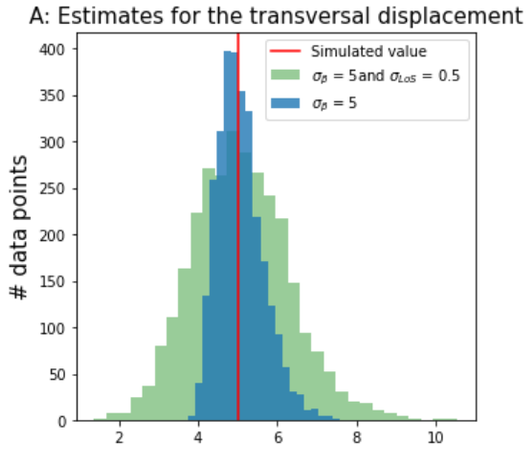


Simulation for:

$\beta = 300.0$
$\gamma_t = 0$
$\gamma_l = 0$

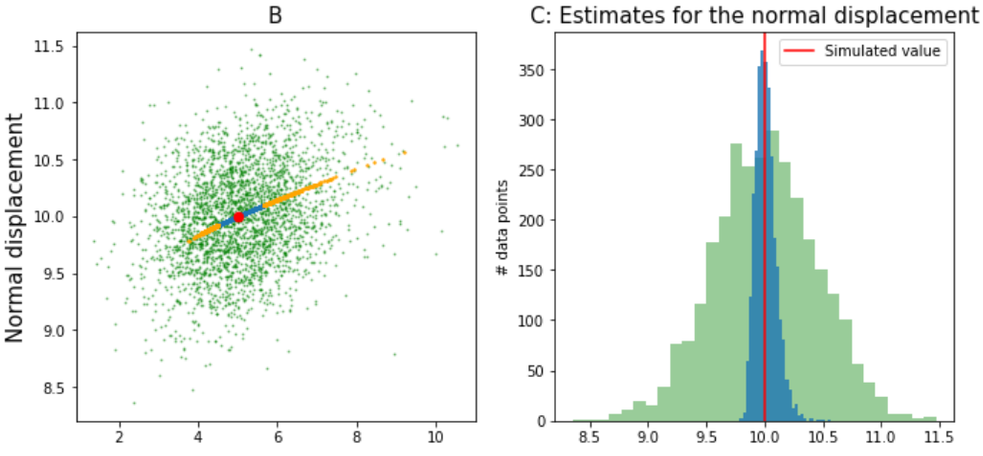


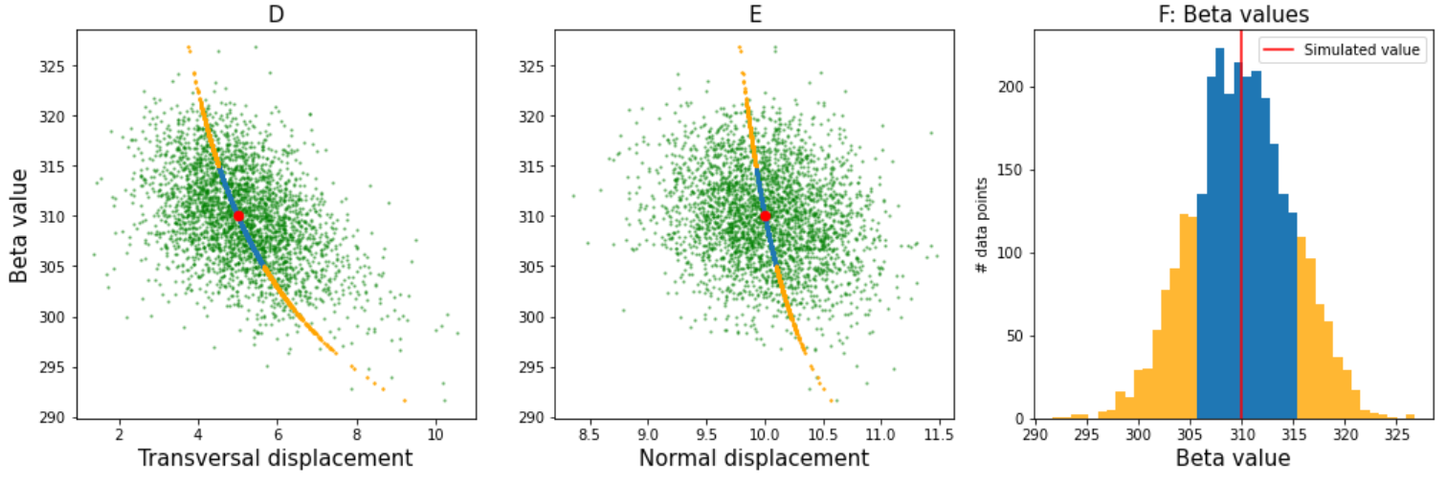


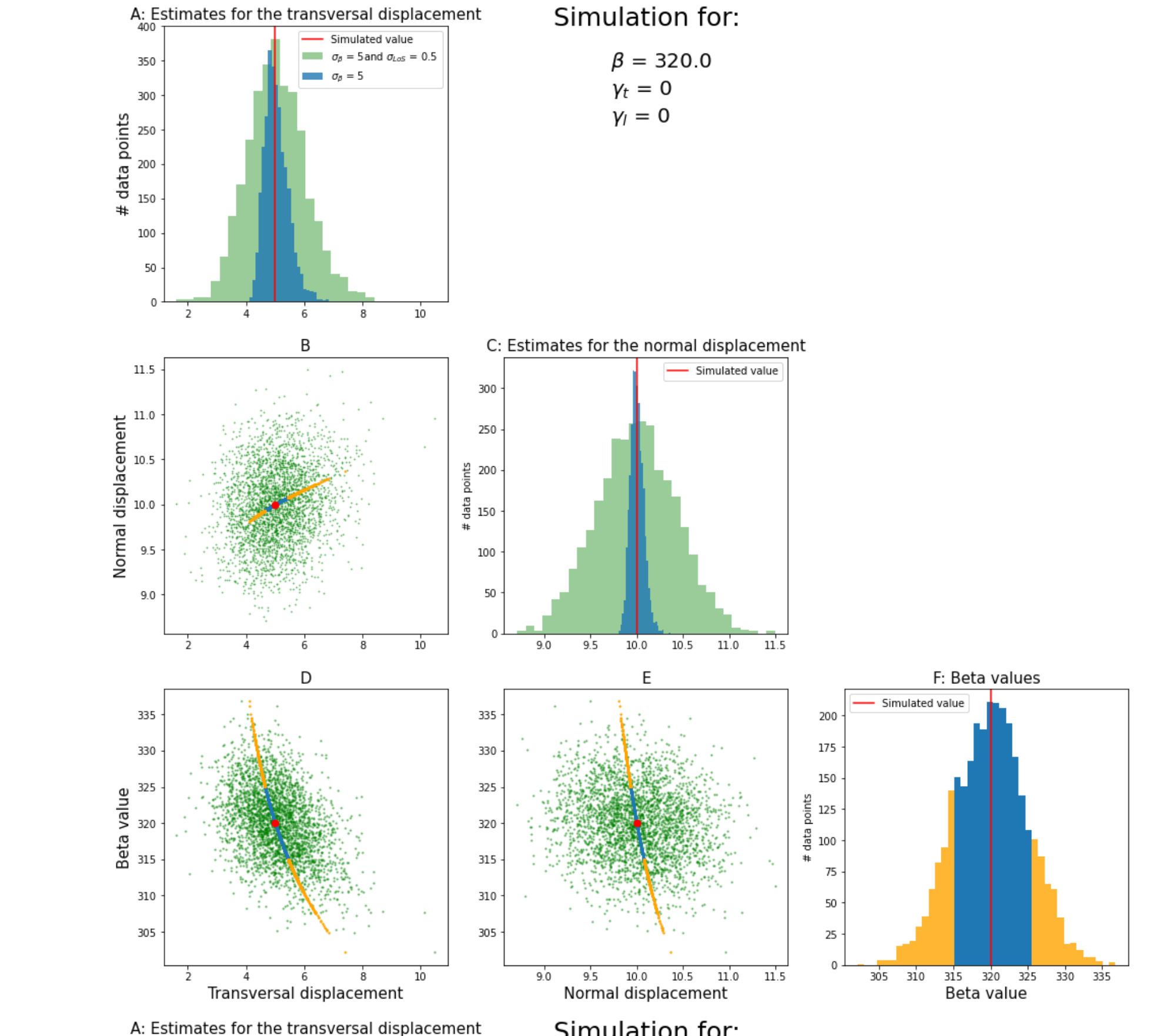


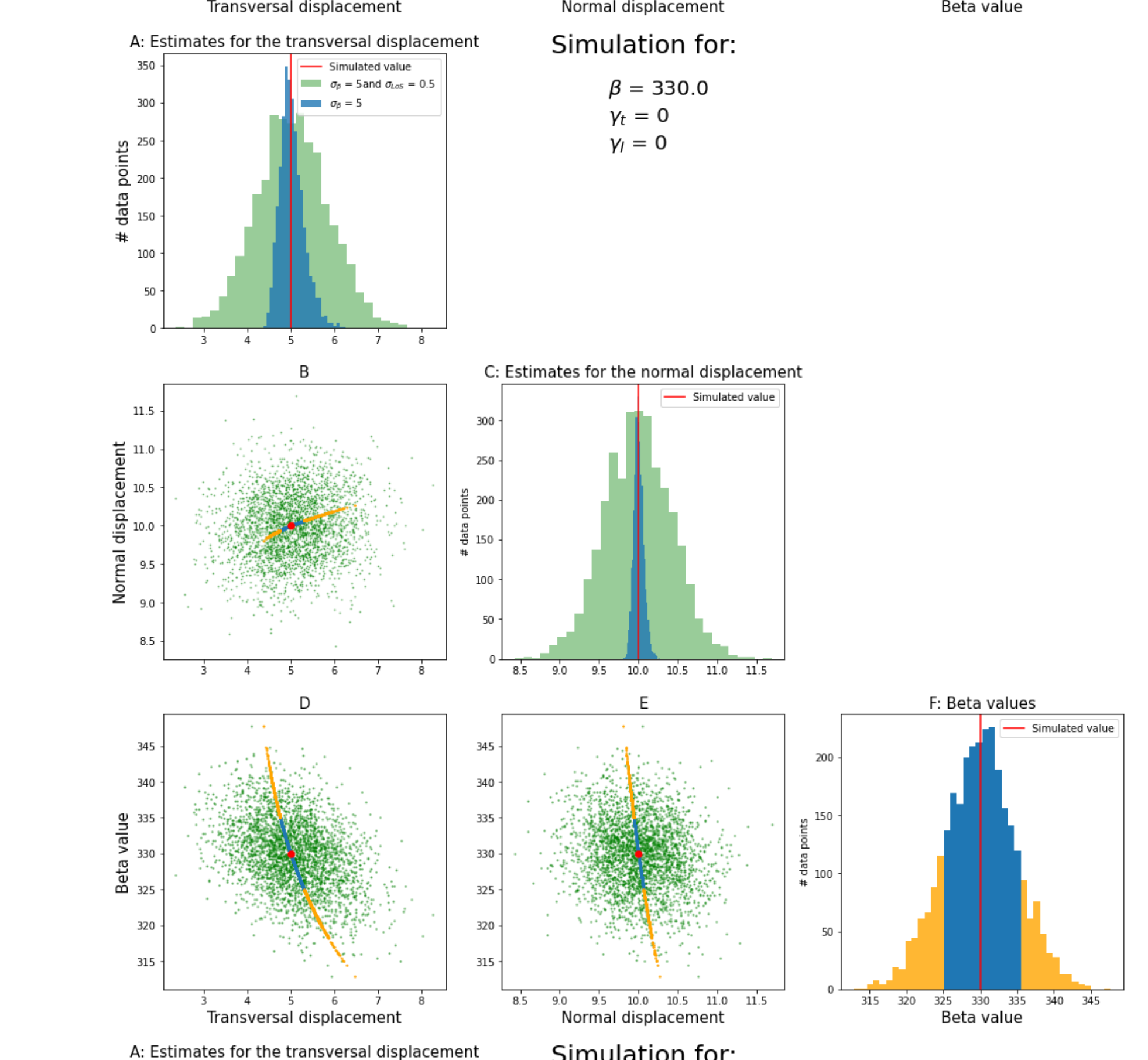
Simulation for:

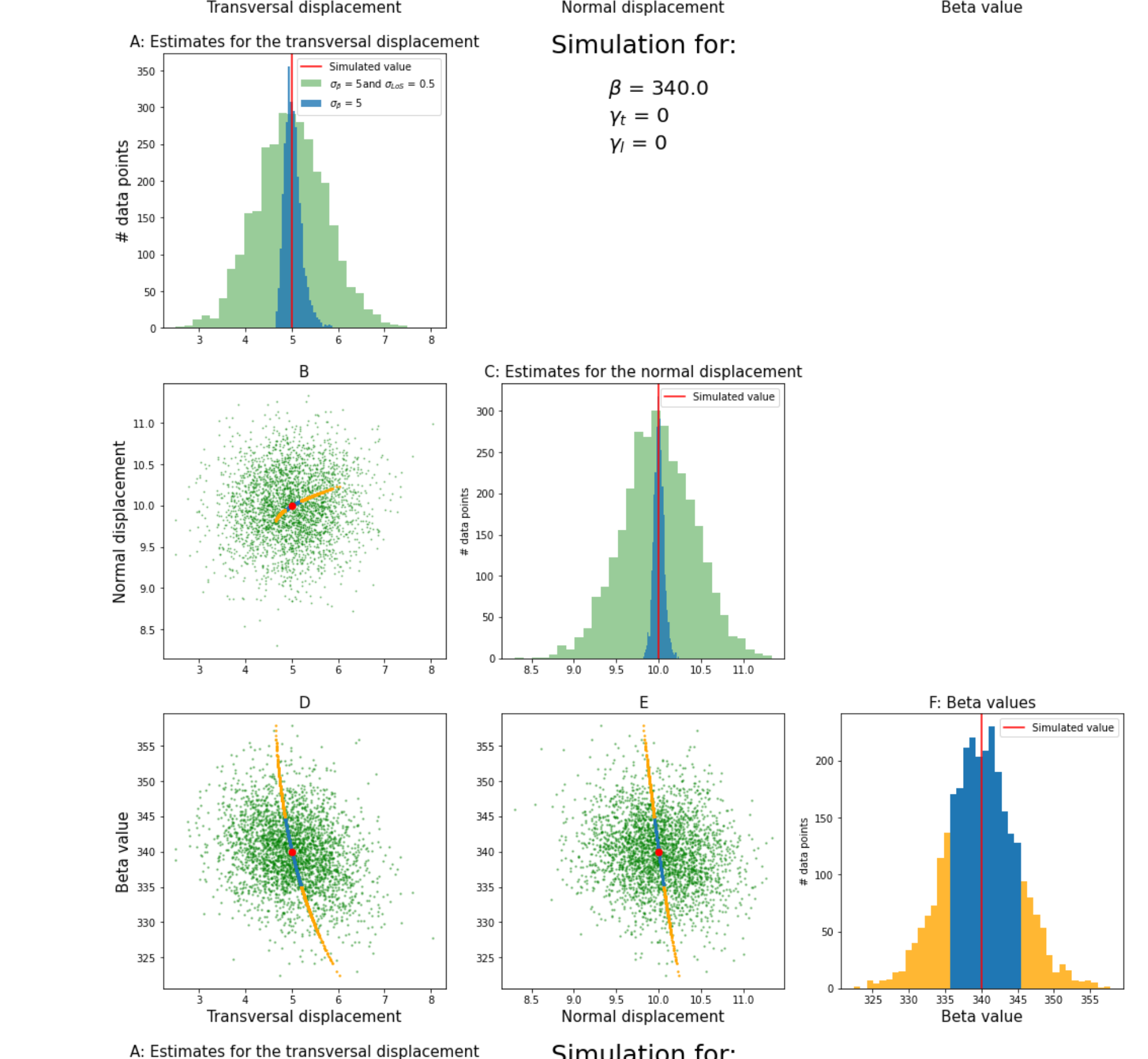
$\beta = 310.0$
$\gamma_t = 0$
$\gamma_l = 0$

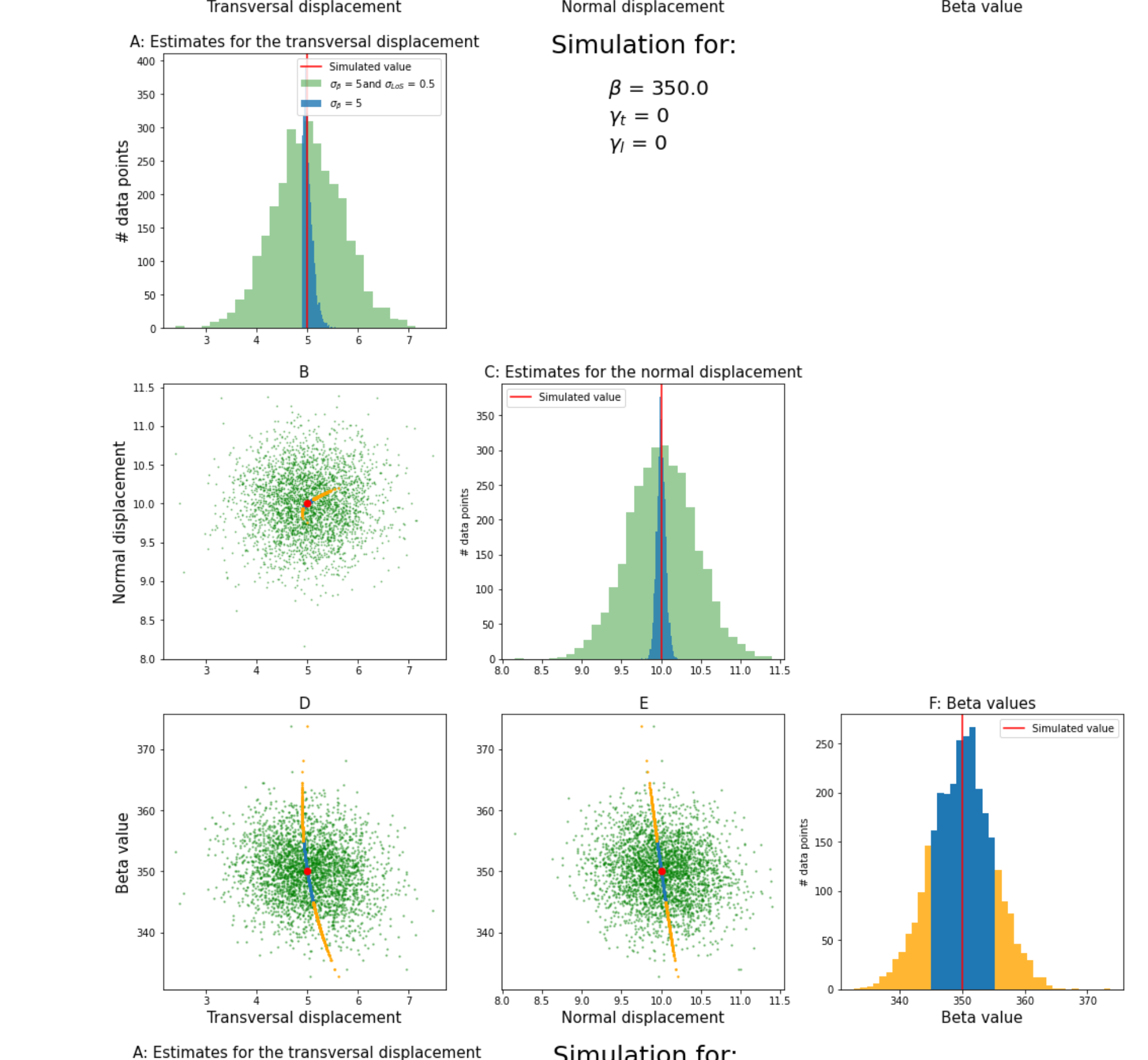


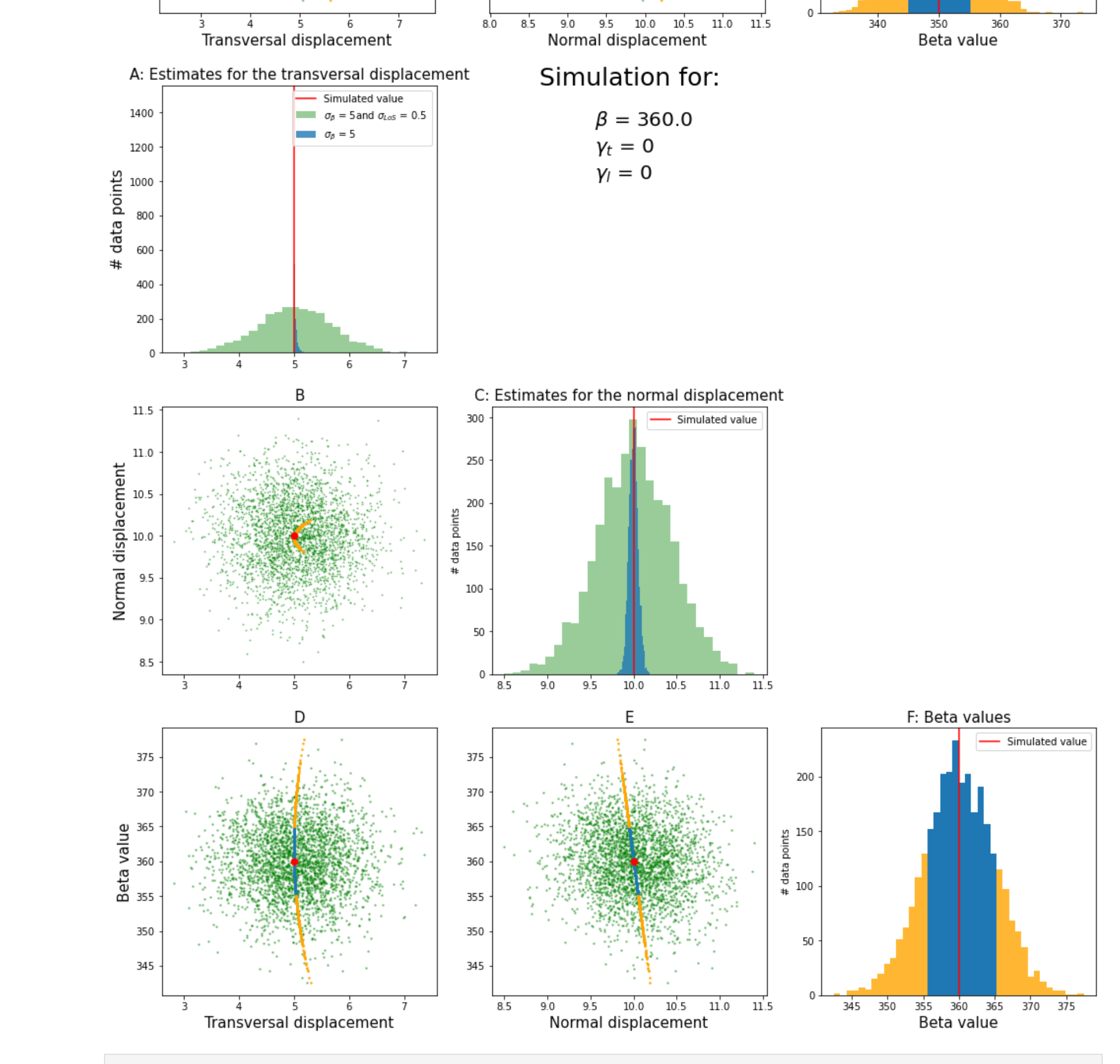












In [ ]:

In [ ]:

In [ ]:

In [4]:

```
b = 40
```

In [ ]:

In [ ]: