# %%

## CODE THAT CREATES WIND SPEED REDUCTION MATRIX

## FUNCTION: WIND SPEED REDUCTION (WSR) (NEW!)

## description: returns a matrix of wind attenuation with nturb turbine rows, ndir direction columns

def windspeedreduction(positionlist,directions,g\_model):

    ndir = int(np.size(directions))

    nturb = int(np.size(positionlist)/2)

    total\_att=np.ones((nturb,ndir))

    for i in range(0,ndir):

        angle\_to\_rotate = 90 + directions[i]; ## create the angle to rotate by in order for wind to be seen as westerly

        westerly\_pos\_list = rotate(angle\_to\_rotate,positionlist) ## rotate position list for westerly wind

        att\_vector = g\_model.predict(westerly\_pos\_list, nturb) ## vector of attenuations (from GP\_functions module)

        att\_vector = att\_vector.reshape(nturb,)

        total\_att[:,i] = att\_vector ##

    total\_att = total\_att/8 ## Get attenuation as a fraction compared to the assumed base wind speed in gaussian model of 8 m/s

    return total\_att

## TESTING

wind\_directions\_for\_WSR = wind\_directions[0]

print('NEW WSR')

wsr = windspeedreduction(positionlist[1:,:],angles,predict\_class) ## the "turbine" at origin is substantion and hence removed

print(f'positionlist.shape = {positionlist.shape}')

print(f'wind\_directions\_for\_WSR.shape = {wind\_directions\_for\_WSR.shape}     (number of wind angles!)')

print(f'type(wsr) = {type(wsr)}\nwsr.shape = {wsr.shape}\nCONCLUSION: the first dimension (y = {wsr.shape[0]}) is nturb, the second dimension (x = {wsr.shape[1]}) is ndir')

# %%

## CODE THAT TRAINS AND USES GP MODELS TO PREDICT WIND SPEEDS

## TRAIN MODEL and assign to predicting class

name = "all\_dataset.csv"

training\_class = GP\_train() ## create training class

training\_model = training\_class.train\_model() ## trains model on "all\_dataset.csv"

## CREATE PREDICTING CLASS from which attenuation predictions are made

predict\_class = GP\_predict(training\_model) ## create predicting class

# %%

## CODE THAT PACKAGES GP FUNCTIONS FOR USE IN MAIN SCRIPT

import pandas as pd

import numpy as np

import matplotlib.pyplot as plt

import GPy

from IPython.display import display

import three\_desc\_model as exponential\_new

from cutoffs import Polynomial

GPy.plotting.change\_plotting\_library('matplotlib')

## CLASS FOR TRAINING

class GP\_train:

    def \_\_init\_\_(self):

        return

    def train\_model(self):

        ## LOAD DATA

        dataset\_full = pd.read\_csv("all\_dataset.csv", index\_col=0)

        symbol="Siemens"

        k=0

        count=0

        dataset\_full["IDnum"]=pd.Series()

        for i in range(len(dataset\_full)):

            dataset\_full.at[i,"IDnum"]=k

            count = count +1

            if (count==dataset\_full["Num\_tot\_turb"].iloc[i]):

                k=k+1

                count=0

        dataset\_full["Num\_tot\_turb"]=dataset\_full["Num\_tot\_turb"].astype(int)

        dataset\_full["Turb\_num"]=dataset\_full["Turb\_num"].astype(int)

        dataset\_full["IDnum"]=dataset\_full["IDnum"].astype(int)

        numsims=dataset\_full["IDnum"].iloc[-1]+1

        turb = [symbol]

        ## INITIALISE NEIGHBOUR LIST

        nl=exponential\_new.NeighborlistCalculator(cutoff=4001,cone\_grad=0.12582561117875557, cone\_offset=72.24947126849844)

        ## INITIALISE FINGPR

        Gs = {"Siemens": [{"type":"G2", "turbine":"Siemens","eta":4.25387599, "offset": 1.0151402},

                          {"type":"G4", "elements":"Siemens","eta":2.56450515, "gamma":8.04475192, "zeta": 2.5356155},

                          {"type":"G6", "elements":"Siemens","eta":2.33043463, "gamma": 0.50753377, "zeta": 0.93372721}

                         ]}

        finpr = exponential\_new.FingerprintCalculator(cutoff=4001,Gs=Gs,Rct=3000,delta\_R=100,cone\_grad=0.12582561117875557, cone\_offset=72.24947126849844)

        ## CALCULATE NL AND FINGPR

        ## outputs: fingerprints and reference velocities

        count=0

        dataset\_fp=np.empty(shape=(0, 3))

        dataset\_ws=np.empty(shape=(0, 1))

        dataset\_pos=np.empty(shape=(0, 2))

        for i in range(numsims):

            numturb=dataset\_full["Num\_tot\_turb"].iloc[count]

            position = np.empty((numturb,2))

            ws = np.empty((numturb,1))

            fp = np.empty((numturb,3))

            for k in range(numturb):

                position[k,0]=dataset\_full.at[count,"X\_coord"]

                position[k,1]=dataset\_full.at[count,"Y\_coord"]

                ws[k,0]=dataset\_full.at[count,"Ref\_wind"]

                count = count+ 1

            neigh=nl.calculate(turb\*numturb,position)

            fingerprints=finpr.calculate(turb\*numturb,position,neigh,symbol)

            fingerprints=np.array(fingerprints)

            dataset\_fp=np.append(dataset\_fp,fingerprints,axis=0)

            dataset\_ws=np.append(dataset\_ws,ws,axis=0)

            dataset\_pos=np.append(dataset\_pos,position,axis=0)

        dataset=np.concatenate((dataset\_fp, dataset\_ws),axis=1)

        dataset = pd.DataFrame(dataset, columns = ['Fingerprint(G2)','Fingerprint(G4)','Fingerprint(G6)','Ref\_Wind\_Speed'])

        X = dataset[["Fingerprint(G2)","Fingerprint(G4)","Fingerprint(G6)"]].to\_numpy()

        Y = dataset[["Ref\_Wind\_Speed"]].to\_numpy()

        train\_dataset = dataset.sample(frac=0.8, random\_state=0)

        test\_dataset = dataset.drop(train\_dataset.index)

        Xtrain=train\_dataset[["Fingerprint(G2)","Fingerprint(G4)","Fingerprint(G6)"]].to\_numpy()

        Ytrain=train\_dataset[["Ref\_Wind\_Speed"]].to\_numpy()

        Xtest=test\_dataset[["Fingerprint(G2)","Fingerprint(G4)","Fingerprint(G6)"]].to\_numpy()

        Ytest=test\_dataset[["Ref\_Wind\_Speed"]].to\_numpy()

        ## DEFINE KERNEL

        ker = GPy.kern.RBF(3, lengthscale = 0.1)

        ## CREATE GP MODEL

        m = GPy.models.GPRegression(Xtrain, Ytrain, ker)

        ## OPTIMISE

        m.optimize(messages = True, max\_f\_eval = 1000)

        return m

## CLASS FOR PREDICTING

class GP\_predict:

    def \_\_init\_\_(self, model):

        self.model = model

        return

    def predict(self, pos\_array, num\_turbs\_predict):

        ## DECLARE TURBINE TYPE

        symbol = "Siemens"

        turb = [symbol]

        ## INITIALISE NEIGHBOUR LIST

        nl = exponential\_new.NeighborlistCalculator(cutoff = 4001,cone\_grad = 0.12582561117875557, cone\_offset = 72.24947126849844)

        ## INITIALISE FINGPR

        Gs = {"Siemens": [{"type":"G2", "turbine":"Siemens","eta":4.25387599, "offset": 1.0151402},

                          {"type":"G4", "elements":"Siemens","eta":2.56450515, "gamma":8.04475192, "zeta": 2.5356155},

                          {"type":"G6", "elements":"Siemens","eta":2.33043463, "gamma": 0.50753377, "zeta": 0.93372721}

                         ]}

        finpr = exponential\_new.FingerprintCalculator(cutoff = 4001,Gs = Gs,Rct = 3000,delta\_R = 100,cone\_grad = 0.12582561117875557, cone\_offset = 72.24947126849844)

        ## MAKE PREDICTIONS

        position = pos\_array;

        neigh = nl.calculate(turb\*num\_turbs\_predict, position)

        fingerprints = np.array(finpr.calculate(turb\*num\_turbs\_predict, position, neigh, symbol))

        refwind, refstdev = self.model.predict(fingerprints)

        return refwind

# %%

## FUNCTION: SIMPLE ROTATE

def rotate(angle, coords):

    angle = np.pi\*angle/180.

    rot\_x\_corrds = []

    rot\_y\_corrds = []

    for coord in coords:

        rot\_x\_corrds += [coord[0]\*np.cos(angle) - coord[1]\*np.sin(angle)]

        rot\_y\_corrds += [coord[0]\*np.sin(angle) + coord[1]\*np.cos(angle)]

    rot\_coords = [rot\_x\_corrds,rot\_y\_corrds]

    rot\_coords = np.array(rot\_coords).T

    return rot\_coords

# %%

## CODE TO SAVE VARIABLE TO FILE FOR USE IN GAUSSIAN CODE

import pickle

optimised\_farm\_variables = {'smallestpositionlist': smallestpositionlist, 'smallestindiceslist': smallestindiceslist, 'OptimizedTPO': OptimizedTPO, 'smallestTotalCost': smallestTotalCost, 'smallestCostPerWatt': smallestCostPerWatt, 'smallestExportDistance': smallestExportDistance, 'MSTWeightSum':MSTWeightSum}

with open('optimised\_farm\_variables.pkl', 'wb') as f:

    pickle.dump(optimised\_farm\_variables, f)

# %%

## CODE TO IMPORT LAYOUT

## note: only valid for the same parameters as the code that generated this layout

import pickle

with open('optimised\_farm\_variables.pkl', 'rb') as f:

    my\_variables = pickle.load(f)

PositionList\_imported = my\_variables['smallestpositionlist'] ## valid import

IndicesList\_imported = my\_variables['smallestindiceslist'] ## valid import

ExportDistance\_imported = my\_variables['smallestExportDistance'] ## valid import

MSTWeightSum\_imported = my\_variables['MSTWeightSum'] ## valid import (total length of cabling)