```python
import distributions as dist
import fracture as frac
import numpy as np
import plotly.graph_objs as go
import plotly.offline as poff
import dill
import matlab_clones as mc
import trace


class DiscreteFractureNetwork:
    """
    Classe per la descrizione del DiscreteFractureNetwork
    """

    def __init__(self, N, Xmin, Xmax, Ymin, Ymax, Zmin, Zmax,
                 alpha_pl, radius_l, radius_u, k, mode_vector, tol=1.0e-10, fixed_n_edges=0):
        """
        Costruttore della classe
        :param N: numero di fratture
        :param Xmin, Xmax, Ymin, Ymax, Zmin, Zmax: estremi del dominio del DFN
        :param alpha_pl: parametro (> 1) della distribuzione Power LawBounded per i semiassi delle ascisse
        :param radius_l: limite inferiore dei semiassi delle ascisse
        :param radius_u: limite superiore dei semiassi delle ascisse
        :param k: parametro di concentrazione della legge Von Mises-Fisher
        :param mode_vector: punto medio della legge Von Mises-Fisher
        :param fixed_n_edges: numero fissato dei lati dei poligoni (>= 8); se non specificato ogni poligono ha
                              un numero casuale di lati da 8 a 16 (distribuiti uniformemente)
        :param tol: tolleranza
        """

        self.N = 0  # il valore verrà poi aggiornato nel metodo genfrac
        self.tol = tol
        self.Xmin = Xmin
        self.Xmax = Xmax
        self.Ymin = Ymin
        self.Ymax = Ymax
        self.Zmin = Zmin
        self.Zmax = Zmax

        self.fixed_n_edges = fixed_n_edges

        self.alpha_pl = alpha_pl
        self.radius_l = radius_l
        self.radius_u = radius_u
        # Si definisce la distribuzione power law bounded per i semiassi delle x
        self.pl_dist = dist.PowerLawBounded(alpha=alpha_pl, radius_l=radius_l, radius_u=radius_u)

        self.k = k
        self.mode_vector = mode_vector / np.linalg.norm(mode_vector) # nel caso il modulo non fosse unitario
        # Si definisce la distribuzione Von Mises-Fisher
        self.vmf_dist = dist.VonMisesFisher(k=k, mode_vector=self.mode_vector)

        self.fractures = []             # Lista delle fratture
        self.poss_intersezioni = []     # Lista di liste con le possibili intersezioni
        self.intersezioni = []          # Lista di liste con le effettive intersezioni
        self.frac_traces = []           # Lista di liste con le tracce per ogni poligono
        self.traces = []                # Lista con tutte le tracce del DFN

        self.genfrac(N)



```

```
64
65      def genfrac(self, n_to_gen):
66          """
67          Genera n_to_gen fratture, aggiornando opportunamente tutte le strutture dati del
     DFN
68          :param n_to_gen: numero di fratture da generare
69          """
70
71          semiaxes_x = self.pl_dist.sample(n_to_gen)  # Vettore di n_to_gen semiassi x
72          ars = np.random.uniform(1, 3, n_to_gen)      # Vettore di n_to_gen aspect ratio
73          normals = self.vmf_dist.sample(n_to_gen)    # Matrice 3 x n_to_gen contenente
    le normali
74
75          if self.fixed_n_edges == 0:
76              n_edges = np.random.random_integers(8, 16, n_to_gen)    # Vettore di
    n_to_gen numero di lati
77          else:
78              n_edges = [self.fixed_n_edges] * n_to_gen
79
80          alpha_angles = np.random.uniform(0, 2 * np.pi, n_to_gen) # Vettore di n_to_gen
    angoli di rotazione
81                                                                   # attorno alla
    normale
82
83          # Matrice n_to_gen x 3 contenente i baricentri
84          centers = np.random.uniform(np.array([self.Xmin, self.Ymin, self.Zmin]),
85                                      np.array([self.Xmax, self.Ymax, self.Zmax]),
86                                      (n_to_gen, 3))
87
88          for i in range(n_to_gen):
89              tmp = frac.Fracture(n_edges[i], semiaxes_x[i], alpha_angles[i], ars[i],
    normals[:, i], centers[i, :])
90              self.fractures.append(tmp)
91              self.poss_intersezioni.append([])
92              self.intersezioni.append([])
93              self.frac_traces.append([])
94
95          if self.N == 0: # Si entra in questo blocco solo al momento della creazione del
     DFN
96              self.N += n_to_gen
97              for i in range(self.N - 1):
98                  for j in range(i + 1, self.N):
99                      self.aggiorna_int(i, j)
100
101         else: # Si entra in questo blocco nel caso di aggiunte di poligoni al DFN già
    esistente
102             # Confrontiamo le vecchie fratture con le nuove
103             for i in range(self.N):
104                 for j in range(n_to_gen):
105                     self.aggiorna_int(i, self.N + j)
106
107             # Confrontiamo le nuove fratture tra loro
108             for i in range(self.N, self.N + n_to_gen - 1):
109                 for j in range(i + 1, self.N + n_to_gen):
110                     self.aggiorna_int(i, j)
111
112             self.N += n_to_gen
113
114     def rimuovi(self, v):
115         """
116         Rimuove le fratture nelle posizioni indicate dalla lista v, e aggiorna
    opportunamente le strutture dati del DFN
117         :param r: lista contenente gli indici dei poligoni da rimuovere
118         """
119         v = list(set(v))
120         v = sorted(v, reverse=True) # Per evitare errori che possono sorgere con la
    rinumerazione
121
122
```

```
123
124
125         for i in v:
126             self.fractures.pop(i)
127             self.poss_intersezioni.pop(i)
128             self.intersezioni.pop(i)
129             for j in range(len(self.frac_traces[i])):
130                 # Presa la j-esima traccia generata dall'i-esimo poligono, la si
    rimuove dalla lista delle tracce
131                 # generate dall'altro genitore
132                 tr = self.frac_traces[i][j]
133                 if tr.i1 == i:
134                     self.frac_traces[tr.i2].remove(tr)
135                 else:
136                     self.frac_traces[tr.i1].remove(tr)
137                 self.traces.remove(tr)
138             self.frac_traces.pop(i)
139
140             for j in range(len(self.poss_intersezioni)):
141                 if i in self.poss_intersezioni[j]:
142                     if i in self.intersezioni[j]:
143                         self.intersezioni[j].remove(i)
144
145                     self.poss_intersezioni[j].remove(i)
146
147                 # Si rinumerano gli indici relativi ai poligoni
148                 for k in range(len(self.poss_intersezioni[j])):
149                     if self.poss_intersezioni[j][k] > i:
150                         self.poss_intersezioni[j][k] -= 1
151                 for k in range(len(self.intersezioni[j])):
152                     if self.intersezioni[j][k] > i:
153                         self.intersezioni[j][k] -= 1
154             # Si rinumerano gli indici dei genitori delle tracce
155             for tr in self.traces:
156                 if tr.i1 > i:
157                     tr.i1 -= 1
158                 if tr.i2 > i:
159                     tr.i2 -= 1
160
161         self.N = self.N - len(v)
162
163     def inters_BB(self, fr1, fr2):
164         """
165         Metodo per controllare se i bounding box di due fratture si intersecano
166         :param fr1: oggetto della classe Fracture
167         :param fr2: oggetto della classe Fracture
168         :return: booleano
169         """
170         max_x_min = max(fr1.xmin, fr2.xmin)
171         min_x_max = min(fr1.xmax, fr2.xmax)
172         max_y_min = max(fr1.ymin, fr2.ymin)
173         min_y_max = min(fr1.ymax, fr2.ymax)
174         max_z_min = max(fr1.zmin, fr2.zmin)
175         min_z_max = min(fr1.zmax, fr2.zmax)
176         if max_x_min <= min_x_max and max_y_min <= min_y_max and max_z_min <= min_z_max
    :
177             return True
178         else:
179             return False
180
181     def gen_trace(self, i1, i2):
182         """
183         Metodo per calcolare la traccia tra due poligoni
184         :param i1, i2: indici dei poligoni
185         :return: oggetto della classe Trace (se la traccia esiste); None se la traccia
    non esiste
186         """
187
188         p1 = self.fractures[i1]
```

```
189         p2 = self.fractures[i2]
190 
191         # Calcoliamo la retta d'intersezione tra i piani contenenti i poligoni
192         t = np.cross(p1.vn, p2.vn)
193         A = np.array([p1.vn, p2.vn, t])
194         b = np.array([np.dot(p1.vertici[:, 0], p1.vn), np.dot(p2.vertici[:, 0], p2.vn)
    , 0])
195 
196         r0 = np.linalg.solve(A, b)
197         s = []
198         s.extend(self.inters_2D(p1, t, r0))
199         if len(s) == 2:
200             s.extend(self.inters_2D(p2, t, r0))
201             if len(s) == 4:
202                 q = np.argsort(s)
203                 if (q[0] == 0 and q[1] == 1) or (q[0] == 2 and q[1] == 3):
204                     return None
205                 else:
206                     s.sort()
207                     x1 = r0 + s[1] * t
208                     x2 = r0 + s[2] * t
209                     tr = trace.Trace(p1, p2, i1, i2, np.array([x1, x2]).T)
210                     return tr
211 
212         else:
213             return None
214 
215     def inters_2D(self, p, t, r0):
216         """
217         Metodo che, data una retta in forma parametrica (X(s) = r0 + s*t) e una frattura
    , determina se si intersecano
218         in un segemento e ne calcola gli estremi
219         :param p: oggetto della classe Fracture
220         :param t: direzione della retta
221         :param r0: punto della retta
222         :return: lista s con i due nuemeri reali [s1, s2] che parametrizzano il segmento
     se c'e' intersezione;
223                  lista vuota se non c'e' intersezione
224         """
225 
226         # Si ruota il poligono per ricondursi sul piano xy
227         rotMatrix = np.dot(np.dot(mc.rotz(- p.alpha), mc.roty(p.phi - np.pi / 2)), mc.
    rotz(- p.teta))
228         vertici = p.vertici.copy()
229 
230         for i in range(p.n):
231             vertici[:, i] -= p.bar
232         vertici = np.dot(rotMatrix, vertici)
233         t = np.dot(rotMatrix, t)
234         r0 = np.dot(rotMatrix, r0 - p.bar)
235 
236         conta = 0    # Contatore dei lati intersecati
237         i = 0
238         s = []
239         while conta < 2 and i < p.n:
240             j = (i + 1) % p.n
241             A = np.array([[t[0], vertici[0, i] - vertici[0, j]], [t[1], vertici[1, i]
     - vertici[1, j]]])
242             b = np.array([vertici[0, i] - r0[0], vertici[1, i] - r0[1]])
243             if - self.tol <= np.linalg.det(A) <= self.tol: # Si controlla il
    parallelismo
244                 s1 = (vertici[0, i] - r0[0]) / t[0]
245                 s2 = (vertici[1, i] - r0[1]) / t[1]
246                 if abs(s1 - s2) < self.tol:
247                     s.append(s1)
248                     s2 = (vertici[0, j] - r0[0]) / t[0]
249                     s.append(s2)
250                     return s
251                 i += 1
```

```
252             else:
253                 x = np.linalg.solve(A, b)
254                 if - self.tol <= x[1] <= 1 + self.tol:
255                     conta += 1
256                     s.append(x[0])
257                 i += 1
258 
259         s.sort()
260         return s
261 
262     def aggiorna_int(self, i, j):
263         """
264         Metodo che aggiorna le intersezioni possibili, reali e se ci sono trova le tracce
265         :param i: indice della prima frattura
266         :param j: indice della seconda frattura
267         """
268 
269         fr1 = self.fractures[i]
270         fr2 = self.fractures[j]
271         if self.inters_BB(fr1, fr2) is True:
272             self.poss_intersezioni[i].append(j)
273             self.poss_intersezioni[j].append(i)
274             tr = self.gen_trace(i, j)
275             if tr is not None:
276                 self.traces.append(tr)
277                 self.frac_traces[i].append(tr)
278                 self.frac_traces[j].append(tr)
279                 self.intersezioni[i].append(j)
280                 self.intersezioni[j].append(i)
281 
282     def visual3D(self, filename='tmp-plot'):
283         """
284         Metodo per la visualizzazione grafica delle fratture e delle tracce
285         :param filename: nome del file che verrà creato
286         """
287 
288         all_polygons = []
289         for i in range(self.N):
290             x = self.fractures[i].vertici[0, :].tolist()
291             y = self.fractures[i].vertici[1, :].tolist()
292             z = self.fractures[i].vertici[2, :].tolist()
293             x.append(x[0])
294             y.append(y[0])
295             z.append(z[0])
296             perimetro = go.Scatter3d(x=x, y=y, z=z, mode='lines', marker=dict(color='red'))
297             area = go.Mesh3d(x=x, y=y, z=z, color='#FFB6C1', opacity=0.60)
298 
299             # Caso in cui il poligono sia parallelo ad uno dei piani coordinati
300             if np.linalg.norm(self.fractures[i].vn - np.array([1, 0, 0])) < self.tol:
301                 area.delaunayaxis = 'x'
302             elif np.linalg.norm(self.fractures[i].vn - np.array([0, 1, 0])) < self.tol:
303                 area.delaunayaxis = 'y'
304             elif np.linalg.norm(self.fractures[i].vn - np.array([0, 0, 1])) < self.tol:
305                 area.delaunayaxis = 'z'
306 
307             all_polygons.append(perimetro)
308             all_polygons.append(area)
309 
310         for i in range(len(self.traces)):
311             x = self.traces[i].estremi[0, :].tolist()
312             y = self.traces[i].estremi[1, :].tolist()
313             z = self.traces[i].estremi[2, :].tolist()
314             segmento = go.Scatter3d(x=x, y=y, z=z, mode='lines', marker=dict(color='black'))
315             all_polygons.append(segmento)
```

```
316
317         fig_3d_alltogether = go.Figure(data=all_polygons)
318         poff.plot(fig_3d_alltogether, filename=filename+'.html')
319
320     def scrittura1(self, filename='file1.txt'):
321         """
322         Metodo per scrivere su file come richiesto al punto 7
323         """
324
325         with open(filename, 'w') as f1:
326             print(self.N, file=f1)
327             for i in range(self.N):
328                 print(i, self.fractures[i].n, file=f1)
329                 for j in range(self.fractures[i].n):
330                     print(self.fractures[i].vertici[0, j],
331                           self.fractures[i].vertici[1, j],
332                           self.fractures[i].vertici[2, j], file=f1)
333
334     def scrittura2(self, filename='file2.txt'):
335         """
336         Metodo per scrivere su file come richiesto al punto 8
337         """
338
339         with open(filename, 'w') as f2:
340             print(self.N, file=f2)
341
342             somma_vertici = 0
343             for i in range(self.N):
344                 somma_vertici = somma_vertici + self.fractures[i].n
345             print(somma_vertici, file=f2)
346
347             indice = 0
348             for i in range(self.N):
349                 print(i, self.fractures[i].n, indice, file=f2)
350                 indice = indice + self.fractures[i].n
351
352             for i in range(self.N):
353                 for j in range(self.fractures[i].n):
354                     print(self.fractures[i].vertici[0, j],
355                           self.fractures[i].vertici[1, j],
356                           self.fractures[i].vertici[2, j], file=f2)
357
358     def save(self):
359         """
360         Salva l'oggetto DiscreteFractureNetwork come file .pkl
361         """
362
363         with open('DFN.pkl', 'wb') as f3:
364             dill.dump(self, f3)
```

```python
import numpy as np
import matlab_clones as mc


class Fracture:
    """
    Classe per la descrizione delle fratture: riceve i vari parametri in ingresso ed effettua le dovute rotazioni
    """

    def __init__(self, n, rx, alpha, ar, v, bar):
        """
        Costruttore della classe
        :param n: numero di vertici
        :param rx: lunghezza semiasse ascisse
        :param alpha: angolo di rotazione attorno all'asse z (prima rotazione)
        :param ar: aspect ratio: rapporto tra semiasse x e semiasse y dell'ellisse circoscritta al poligono
        :param v: versore normale
        :param bar: baricentro
        """

        self.n = n
        self.rx = rx
        self.ry = rx / ar
        self.alpha = alpha
        self.phi = mc.cart2sph(v[0], v[1], v[2])[1]
        self.teta = mc.cart2sph(v[0], v[1], v[2])[0]
        self.vn = v
        self.bar = bar
        # vertici = matrice 3xn contenente i vertici come vettore colonna ordinati in senso antiorario
        self.vertici = np.zeros((3, n))

        # Si generano i vertici del poligono
        for i in range(n):
            self.vertici[0, i] = self.rx * np.cos(i * 2 * np.pi / n)
            self.vertici[1, i] = self.ry * np.sin(i * 2 * np.pi / n)

        # Si ruotano i vertici
        rotMatrix = np.dot(np.dot(mc.rotz(self.teta), mc.roty(np.pi / 2 - self.phi)), mc.rotz(alpha))
        self.vertici = np.dot(rotMatrix, self.vertici)

        # Si traslano i vertici
        for i in range(n):
            self.vertici[:, i] = self.vertici[:, i] + bar

        self.xmin = np.min(self.vertici[0, :])
        self.xmax = np.max(self.vertici[0, :])
        self.ymin = np.min(self.vertici[1, :])
        self.ymax = np.max(self.vertici[1, :])
        self.zmin = np.min(self.vertici[2, :])
        self.zmax = np.max(self.vertici[2, :])
```

```python
import numpy as np
import DFN as dfn
import dill

# Settiamo i parametri del costruttore del DFN

N = 5
Xmin = 0
Xmax = 5
Ymin = 0
Ymax = 5
Zmin = 0
Zmax = 5
alpha_pl = 2
radius_l = 2
radius_u = 3
k = 3
mode_vector = np.array([[0.], [0.], [1.]])
fixed_n_edges = 0

# Creiamo un network

network = dfn.DiscreteFractureNetwork(N, Xmin, Xmax, Ymin, Ymax, Zmin, Zmax,
                                      alpha_pl, radius_l, radius_u, k, mode_vector,
fixed_n_edges)
v = [1, 3]
network.visual3D('1')
network.scrittura1()
network.scrittura2()
print(network.poss_intersezioni)
print(network.intersezioni)
print(network.frac_traces)
print(network.traces)
network.rimuovi(v)
network.scrittura1('file3.txt')
network.scrittura2('file4.txt')
network.visual3D('2')
print(network.poss_intersezioni)
print(network.intersezioni)
print(network.frac_traces)
print(network.traces)
network.save()
with open('DFN.pkl','rb') as f:
    network2 = dill.load(f)
network2.visual3D('3')
```

```python
import numpy as np
import DFN as dfn

# Settiamo i parametri del costruttore del DFN

N = 4
Xmin = 0
Xmax = 5
Ymin = 0
Ymax = 5
Zmin = 0
Zmax = 5
alpha_pl = 2
radius_l = 1.5
radius_u = 2.5
k = 3
mode_vector = np.array([[0.], [0.], [1.]])
fixed_n_edges = 0

# Creiamo un network

network = dfn.DiscreteFractureNetwork(N, Xmin, Xmax, Ymin, Ymax, Zmin, Zmax,
                                      alpha_pl, radius_l, radius_u, k, mode_vector,
fixed_n_edges)
network.visual3D('1')
network.scrittura1()
network.scrittura2()
print(network.poss_intersezioni)
print(network.intersezioni)
print(network.frac_traces)
print(network.traces)
network.genfrac(2)
network.scrittura1('file3.txt')
network.scrittura2('file4.txt')
network.visual3D('2')
print(network.poss_intersezioni)
print(network.intersezioni)
print(network.frac_traces)
print(network.traces)
```

```python
import DFN
import numpy as np
import matlab_clones as mc

class Trace:
    """
    Classe per la descrizione delle tracce
    """
    def __init__(self, parent1, parent2, i1, i2, estremi):
        """
        :param parent1, parent2: oggetti della classe Fracture, "genitori" della traccia
        :param i1, i2: indici di parent1 e parent2 all'interno della lista di tutte le fratture del DFN
        :param estremi: matrice 3 x 2, avente come colonne gli estremi della traccia
        """

        self.i1 = i1
        self.i2 = i2
        self.estremi = estremi
        self.parent1 = parent1
        self.parent2 = parent2

    def direzione(self):
        """
        :return: vettore direzione normalizzato (array di numpy di dimensione 1) della retta su cui giace la traccia
        """
        t = np.cross(self.parent1.vn, self.parent2.vn)
        return t


```