**fracture.py**

**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, :])

**DFN.py**

**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)

**def** genfrac(self, n\_to\_gen):  
 *"""  
 Genera n\_to\_gen fratture, aggiornando opportunamente tutte le strutture dati del DFN* **:param** *n\_to\_gen: numero di fratture da generare  
 """* semiaxes\_x = self.pl\_dist.sample(n\_to\_gen) *# Vettore di n\_to\_gen semiassi x* ars = np.random.uniform(1, 3, n\_to\_gen) *# Vettore di n\_to\_gen aspect ratio* normals = self.vmf\_dist.sample(n\_to\_gen) *# Matrice 3 x n\_to\_gen contenente le normali* **if** self.fixed\_n\_edges == 0:  
 n\_edges = np.random.random\_integers(8, 16, n\_to\_gen) *# Vettore di n\_to\_gen numero di*

*lati* **else**:  
 n\_edges = [self.fixed\_n\_edges] \* n\_to\_gen  
  
 alpha\_angles = np.random.uniform(0, 2 \* np.pi, n\_to\_gen) *# Vettore di n\_to\_gen angoli di*

*rotazione attorno alla normale  
  
 # Matrice n\_to\_gen x 3 contenente i baricentri* centers = np.random.uniform(np.array([self.Xmin, self.Ymin, self.Zmin]),  
 np.array([self.Xmax, self.Ymax, self.Zmax]),  
 (n\_to\_gen, 3))  
  
 **for** i **in** range(n\_to\_gen):  
 tmp = frac.Fracture(n\_edges[i], semiaxes\_x[i], alpha\_angles[i], ars[i], normals[:, i],

centers[i, :])  
 self.fractures.append(tmp)  
 self.poss\_intersezioni.append([])  
 self.intersezioni.append([])  
 self.frac\_traces.append([])  
  
 **if** self.N == 0: *# Si entra in questo blocco solo al momento della creazione del DFN* self.N += n\_to\_gen  
 **for** i **in** range(self.N - 1):  
 **for** j **in** range(i + 1, self.N):  
 self.aggiorna\_int(i, j)  
  
 **else**: *# Si entra in questo blocco nel caso di aggiunte di poligoni al DFN già esistente  
 # Confrontiamo le vecchie fratture con le nuove* **for** i **in** range(self.N):  
 **for** j **in** range(n\_to\_gen):  
 self.aggiorna\_int(i, self.N + j)  
  
 *# Confrontiamo le nuove fratture tra loro* **for** i **in** range(self.N, self.N + n\_to\_gen - 1):  
 **for** j **in** range(i + 1, self.N + n\_to\_gen):  
 self.aggiorna\_int(i, j)  
  
 self.N += n\_to\_gen  
  
 **def** aggiungi(self, v):  
 *"""  
 Aggiunge i poligoni, già esistenti, presenti nella lista v* **:param** *v: lista di oggetti di classe Fracture  
 """* **for** i **in** range(len(v)):  
 self.fractures.append(v[i])  
 self.poss\_intersezioni.append([])  
 self.intersezioni.append([])  
 self.frac\_traces.append([])  
  
 *# Confrontiamo le vecchie fratture con le nuove* **for** i **in** range(self.N):  
 **for** j **in** range(len(v)):  
 self.aggiorna\_int(i, self.N + j)  
  
 *# Confrontiamo le nuove fratture tra loro* **for** i **in** range(self.N, self.N + len(v) - 1):  
 **for** j **in** range(i + 1, self.N + len(v)):  
 self.aggiorna\_int(i, j)  
  
 self.N += len(v)

**def** rimuovi(self, v):  
 *"""  
 Rimuove le fratture nelle posizioni indicate dalla lista v, e aggiorna opportunamente le*

*strutture dati del DFN* **:param** *r: lista contenente gli indici dei poligoni da rimuovere  
 """* v = list(set(v))  
 v = sorted(v, reverse=**True**) *# Per evitare errori che possono sorgere con la rinumerazione* **for** i **in** v:  
 self.fractures.pop(i)  
 self.poss\_intersezioni.pop(i)  
 self.intersezioni.pop(i)  
 **for** j **in** range(len(self.frac\_traces[i])):  
 *# Presa la j-esima traccia generata dall'i-esimo poligono, la si rimuove dalla lista*

*delle tracce  
 # generate dall'altro genitore* tr = self.frac\_traces[i][j]  
 **if** tr.i1 == i:  
 self.frac\_traces[tr.i2].remove(tr)  
 **else**:  
 self.frac\_traces[tr.i1].remove(tr)  
 self.traces.remove(tr)  
 self.frac\_traces.pop(i)  
  
 **for** j **in** range(len(self.poss\_intersezioni)):  
 **if** i **in** self.poss\_intersezioni[j]:  
 **if** i **in** self.intersezioni[j]:  
 self.intersezioni[j].remove(i)  
  
 self.poss\_intersezioni[j].remove(i)  
  
 *# Si rinumerano gli indici relativi ai poligoni* **for** k **in** range(len(self.poss\_intersezioni[j])):  
 **if** self.poss\_intersezioni[j][k] > i:  
 self.poss\_intersezioni[j][k] -= 1  
 **for** k **in** range(len(self.intersezioni[j])):  
 **if** self.intersezioni[j][k] > i:  
 self.intersezioni[j][k] -= 1  
 *# Si rinumerano gli indici dei genitori delle tracce* **for** tr **in** self.traces:  
 **if** tr.i1 > i:  
 tr.i1 -= 1  
 **if** tr.i2 > i:  
 tr.i2 -= 1  
  
 self.N = self.N - len(v)  
  
 **def** inters\_BB(self, fr1, fr2):  
 *"""  
 Metodo per controllare se i bounding box di due fratture si intersecano* **:param** *fr1: oggetto della classe Fracture* **:param** *fr2: oggetto della classe Fracture* **:return***: booleano  
 """* max\_x\_min = max(fr1.xmin, fr2.xmin)  
 min\_x\_max = min(fr1.xmax, fr2.xmax)  
 max\_y\_min = max(fr1.ymin, fr2.ymin)  
 min\_y\_max = min(fr1.ymax, fr2.ymax)  
 max\_z\_min = max(fr1.zmin, fr2.zmin)  
 min\_z\_max = min(fr1.zmax, fr2.zmax)  
 **if** max\_x\_min <= min\_x\_max **and** max\_y\_min <= min\_y\_max **and** max\_z\_min <= min\_z\_max:  
 **return True  
 else**:  
 **return False**

**def** gen\_trace(self, i1, i2):  
 *"""  
 Metodo per calcolare la traccia tra due poligoni* **:param** *i1, i2: indici dei poligoni* **:return***: oggetto della classe Trace (se la traccia esiste); None se la traccia non esiste  
 """* p1 = self.fractures[i1]  
 p2 = self.fractures[i2]  
  
 *# Calcoliamo la retta d'intersezione tra i piani contenenti i poligoni* t = np.cross(p1.vn, p2.vn)  
 A = np.array([p1.vn, p2.vn, t])  
 b = np.array([np.dot(p1.vertici[:, 0], p1.vn), np.dot(p2.vertici[:, 0], p2.vn), 0])  
  
 r0 = np.linalg.solve(A, b)  
 s = []  
 s.extend(self.inters\_2D(p1, t, r0))  
 **if** len(s) == 2:  
 s.extend(self.inters\_2D(p2, t, r0))  
 **if** len(s) == 4:  
 q = np.argsort(s)  
 **if** (q[0] == 0 **and** q[1] == 1) **or** (q[0] == 2 **and** q[1] == 3):  
 **return None  
 else**:  
 s.sort()  
 x1 = r0 + s[1] \* t  
 x2 = r0 + s[2] \* t  
 tr = trace.Trace(p1, p2, i1, i2, np.array([x1, x2]).T)  
 **return** tr  
  
 **else**:  
 **return None  
  
 def** inters\_2D(self, p, t, r0):  
 *"""  
 Metodo che, data una retta in forma parametrica (X(s) = r0 + s\*t) e una frattura, determina*

*se si intersecano in un segemento e ne calcola gli estremi* **:param** *p: oggetto della classe Fracture* **:param** *t: direzione della retta* **:param** *r0: punto della retta* **:return***: lista s con i due nuemeri reali [s1, s2] che parametrizzano il segmento se c'e'*

*intersezione; lista vuota se non c'e' intersezione  
 """  
  
 # Si ruota il poligono per ricondursi sul piano xy* rotMatrix = np.dot(np.dot(mc.rotz(- p.alpha), mc.roty(p.phi - np.pi / 2)),

mc.rotz

(- p.teta))  
 vertici = p.vertici.copy()  
  
 **for** i **in** range(p.n):  
 vertici[:, i] -= p.bar  
 vertici = np.dot(rotMatrix, vertici)  
 t = np.dot(rotMatrix, t)  
 r0 = np.dot(rotMatrix, r0 - p.bar)  
  
 conta = 0 *# Contatore dei lati intersecati* i = 0  
 s = []  
 **while** conta < 2 **and** i < p.n:  
 j = (i + 1) % p.n  
 A = np.array([[t[0], vertici[0, i] - vertici[0, j]],

[t[1], vertici[1, i] - vertici[1, j]]])  
 b = np.array([vertici[0, i] - r0[0], vertici[1, i] - r0[1]])  
 **if** - self.tol <= np.linalg.det(A) <= self.tol: *# Si controlla il parallelismo* s1 = (vertici[0, i] - r0[0]) / t[0]  
 s2 = (vertici[1, i] - r0[1]) / t[1]  
 **if** abs(s1 - s2) < self.tol:  
 s.append(s1)  
 s2 = (vertici[0, j] - r0[0]) / t[0]  
 s.append(s2)  
 **return** s  
 i += 1  
 **else**:  
 x = np.linalg.solve(A, b)  
 **if** - self.tol <= x[1] <= 1 + self.tol:  
 conta += 1  
 s.append(x[0])  
 i += 1  
  
 s.sort()  
 **return** s  
  
 **def** aggiorna\_int(self, i, j):  
  
 fr1 = self.fractures[i]  
 fr2 = self.fractures[j]  
 **if** self.inters\_BB(fr1, fr2) **is True**:  
 self.poss\_intersezioni[i].append(j)  
 self.poss\_intersezioni[j].append(i)  
 tr = self.gen\_trace(i, j)  
 **if** tr **is not None**:  
 self.traces.append(tr)  
 self.frac\_traces[i].append(tr)  
 self.frac\_traces[j].append(tr)  
 self.intersezioni[i].append(j)  
 self.intersezioni[j].append(i)  
  
 **def** visual3D(self, filename=**'tmp-plot'**):  
 *"""  
 Metodo per la visualizzazione grafica delle fratture e delle tracce* **:param** *filename: nome del file che verrà creato  
 """* all\_polygons = []  
 **for** i **in** range(self.N):  
 x = self.fractures[i].vertici[0, :].tolist()  
 y = self.fractures[i].vertici[1, :].tolist()  
 z = self.fractures[i].vertici[2, :].tolist()  
 x.append(x[0])  
 y.append(y[0])  
 z.append(z[0])  
 perimetro = go.Scatter3d(x=x, y=y, z=z, mode=**'lines'**, marker=dict(color=**'red'**))  
 area = go.Mesh3d(x=x, y=y, z=z, color=**'#FFB6C1'**, opacity=0.60)  
  
 *# Caso in cui il poligono sia parallelo ad uno dei piani coordinati* **if** np.linalg.norm(self.fractures[i].vn - np.array([1, 0, 0])) < self.tol:  
 area.delaunayaxis = **'x'  
 elif** np.linalg.norm(self.fractures[i].vn - np.array([0, 1, 0])) < self.tol:  
 area.delaunayaxis = **'y'  
 elif** np.linalg.norm(self.fractures[i].vn - np.array([0, 0, 1])) < self.tol:  
 area.delaunayaxis = **'z'** all\_polygons.append(perimetro)  
 all\_polygons.append(area)  
  
 **for** i **in** range(len(self.traces)):  
 x = self.traces[i].estremi[0, :].tolist()  
 y = self.traces[i].estremi[1, :].tolist()  
 z = self.traces[i].estremi[2, :].tolist()  
 segmento = go.Scatter3d(x=x, y=y, z=z, mode=**'lines'**, marker=dict(color=**'black'**))  
 all\_polygons.append(segmento)  
  
 fig\_3d\_alltogether = go.Figure(data=all\_polygons)  
 poff.plot(fig\_3d\_alltogether, filename=filename+**'.html'**)  
  
 **def** scrittura1(self, filename=**'file1.txt'**):  
 *"""  
 Metodo per scrivere su file come richiesto al punto 7  
 """* **with** open(filename, **'w'**) **as** f1:  
 print(self.N, file=f1)  
 **for** i **in** range(self.N):  
 print(i, self.fractures[i].n, file=f1)  
 **for** j **in** range(self.fractures[i].n):  
 print(self.fractures[i].vertici[0, j],  
 self.fractures[i].vertici[1, j],  
 self.fractures[i].vertici[2, j], file=f1)

**def** scrittura2(self, filename=**'file2.txt'**):  
 *"""  
 Metodo per scrivere su file come richiesto al punto 8  
 """* **with** open(filename, **'w'**) **as** f2:  
 print(self.N, file=f2)  
  
 somma\_vertici = 0  
 **for** i **in** range(self.N):  
 somma\_vertici = somma\_vertici + self.fractures[i].n  
 print(somma\_vertici, file=f2)  
  
 indice = 0  
 **for** i **in** range(self.N):  
 print(i, self.fractures[i].n, indice, file=f2)  
 indice = indice + self.fractures[i].n  
  
 **for** i **in** range(self.N):  
 **for** j **in** range(self.fractures[i].n):  
 print(self.fractures[i].vertici[0, j],  
 self.fractures[i].vertici[1, j],  
 self.fractures[i].vertici[2, j], file=f2)  
  
 **def** save(self):  
 *"""  
 Salva l'oggetto DiscreteFractureNetwork come file .pkl  
 """* **with** open(**'DFN.pkl'**, **'wb'**) **as** f3:  
 dill.dump(self, f3)

**trace.py**

**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

**main.py**

**import** numpy **as** np  
**import** DFN **as** dfn  
**import** dill

N = 4  
Xmin = 0  
Xmax = 3  
Ymin = 0  
Ymax = 3  
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  
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'**)

**main1.py**

**import** numpy **as** np  
**import** DFN **as** dfn  
N = 4  
Xmin = 0  
Xmax = 3  
Ymin = 0  
Ymax = 3  
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  
  
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)  
n2 = dfn.DiscreteFractureNetwork(2, Xmin, Xmax, Ymin, Ymax, Zmin, Zmax,  
 alpha\_pl, radius\_l, radius\_u, k, mode\_vector, fixed\_n\_edges)  
network.aggiungi(n2.fractures)  
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)

**main2.py**

**import** numpy **as** np  
**import** DFN **as** dfn  
N = 4  
Xmin = 0  
Xmax = 3  
Ymin = 0  
Ymax = 3  
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  
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)