

1 Python correction



```
1 import numpy as np;
  import matplotlib.pyplot as plt;
3 from scipy.spatial.distance import pdist;
  from scipy.stats import poisson
```

1.1 Conditional Poisson Process

The conditional Poisson Point Process uses a given number of point, contrary to the Poisson Process where the number of points follows a Poisson law. The result is illustrated in Fig.1.

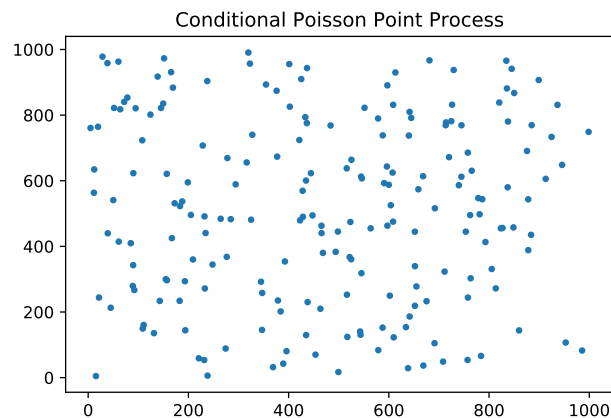


Figure 1: Conditional Poisson point process, with N=100 points.



```
def cond_Poisson(nb_points, xmin, xmax, ymin, ymax):
2     # Conditional Poisson Point Process
    # uniform distribution
4     # nb_points: number of points
    # xmin, xmax, ymin, ymax: defined the domain (window)
6     x = xmin + (xmax-xmin)*np.random.rand(nb_points)
    y = ymin + (ymax-ymin)*np.random.rand(nb_points);
8     return x,y

10 def test_ppp():
    # testing function
12     x,y = cond_Poisson(100, 0, 100, 0, 100);
    plt.plot(x,y, '+');
```

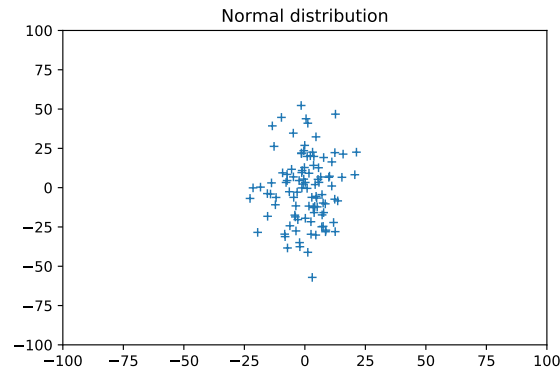


Figure 2: Normal distribution of points around $(0,0)$, with $\sigma = (10,20)$.

1.2 Normal distribution

The normal distribution is illustrated in Fig.2. Each marginal distribution (distribution on each axis) follows the normal distribution.



```
1 def normal_distribution(nb_points, mu, sigma):  
    # Normal distribution centered around the point mu with stdev sigma  
3     x = mu[0] + sigma[0]*np.random.randn(nb_points);  
     y = mu[1] + sigma[1]*np.random.randn(nb_points);  
5     return x,y;
```

1.3 Neyman-Scott Process

Neymann-Scott point process is an aggregated Poisson point process. It is illustrated in Fig.3. It consists on a “sub” point processes generated at locations corresponding to a point process.

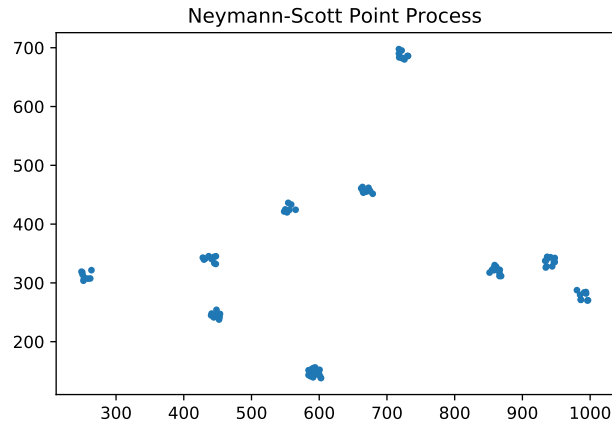


Figure 3: Neyman-Scott point process, with $\lambda S=10$ and $r_{\text{Son}}=10$.



```

1 def neyman_scott(nRoot, xmin, xmax, ymin, ymax, lambdaS, rSon):
2     # Neyman-scott process simulation
3     # nRoot: number of agregates
4     # xmin, xmax, ymin, ymax: domain
5     # lambdaS: number of points. lambda is a density, S is the spatial
6     #         ↪ domain
7     # rSon: radius around agregate (points are distributed in a square)
8
9     # number of sons
10    nSons = poisson.rvs(lambdaS, size=nRoot);
11
12    # results
13    x=[];
14    y=[];
15
16    # father points coordinates
17    xf,yf = cond_Poisson(nRoot, xmin, xmax, ymin, ymax);
18    for i in range(nRoot):
19        # loop over all agregates
20        xs,ys = cond_Poisson(nSons[i], xf[i]-rSon, xf[i]+rSon, yf[i]-rSon
21        ↪ , yf[i]+rSon);
22        x = np.concatenate((x,xs), axis=0);
23        y = np.concatenate((y,ys), axis=0);
24
25    return x,y

```

1.4 Gibbs Point Process

Gibbs point process allows attraction and repulsion at different distances. It is illustrated in Fig.4. The attraction/repulsion law is given by the following code for regular or aggregated point process.



```

def exampleEnergyFunction(distance):
2     """ This function returns e with the same size as distance
        e takes the value given in the variable energy according to the steps
        """
4     e = np.zeros(distance.shape);
6     e [distance <10] = 10;
    return e;

8
def aggregatedEnergyFunction(distance):
10     """
    Agregated energy function
    """
12     e = np.zeros(distance.shape);
14     e [distance <2] = 50;
    e [np.logical_and(distance >=2, distance <5)] = -10;
16     e [np.logical_and(distance >=5, distance <10)] = 5;

18     return e;

```

The evaluation of the energy computes all pairwise distances and sum up the energies associated, or it computes the distances between one single point to a set of points.



```

def energy(P, eFunction=exampleEnergyFunction):
2     """
        This computes the energy in the set of points P, with the energy
        ↪ function
        given as a parameter.
        return a float value
        """
6     P = np.transpose(P);
8     d = pdist(P);
    e = exampleEnergyFunction(d);
10    return np.sum(e);

12 def energyFromPoint(p, P, eFunction=exampleEnergyFunction):
    """
14    Compute energy from point p to all points of P
    """
16    dist = np.sqrt((p[0] - P[0,:])**2 + (p[1] - P[1,:])**2);
    ee = eFunction(dist);
18    return np.sum(ee);

```

The principle of the algorithm is to iteratively add one point that minimizes the energy after several trials. In order to speed up the process, notice that only one point is moved, and it is thus sufficient to only compute the distances from this point to all others.



```

def gibbs(nb_points, xmin, xmax, ymin, ymax, nbiter, eFunction=
↳ exampleEnergyFunction):
2     """
3     Gibbs point process
4     xmin, xmax, ymin, ymax represents the spatial window
5     nb_points: number of generated points
6     nbiter: number of iterations
7     returns (x,y) coordinates of the points
8     """

10    # start with a Poisson Point Process
    x,y = spat_pp.cond_Poisson(nb_points, xmin, xmax, ymin, ymax);
12    nb_moves = 0;
    e_prev = energy(np.vstack((x,y)), eFunction);
14    print("initial energy: {0:f}".format(e_prev));

16    for i in range(nbiter):
        # choose a random point
18        j = np.random.randint(0, nb_points);
        x2 = np.delete(x, j);
20        y2 = np.delete(y, j);

22        P = np.vstack((x2, y2));
        e1 = energyFromPoint([x[j], y[j]], P, eFunction);

24        for m in range(10):
            xm,ym = spat_pp.cond_Poisson(1, xmin, xmax, ymin, ymax);

26            e2 = energyFromPoint([xm, ym], P, eFunction);
            if e2<e1:
30                nb_moves+=1;
                x[j]=xm;
32                y[j]=ym;
                e1=e2;

34        print("Number of moves: " + str(nb_moves));
36        print("Final energy: " + str(e1));
    return x, y

```

1.5 Ripley functions

The Ripley functions are useful to characterize a point process. Agregation and repulsion can be observed with regard to the distance (see Fig.5). Notice that this function is biased because points in border of window are counted as points in the center. This could be corrected by the use of `scipy.spatial.distance.cdist`.

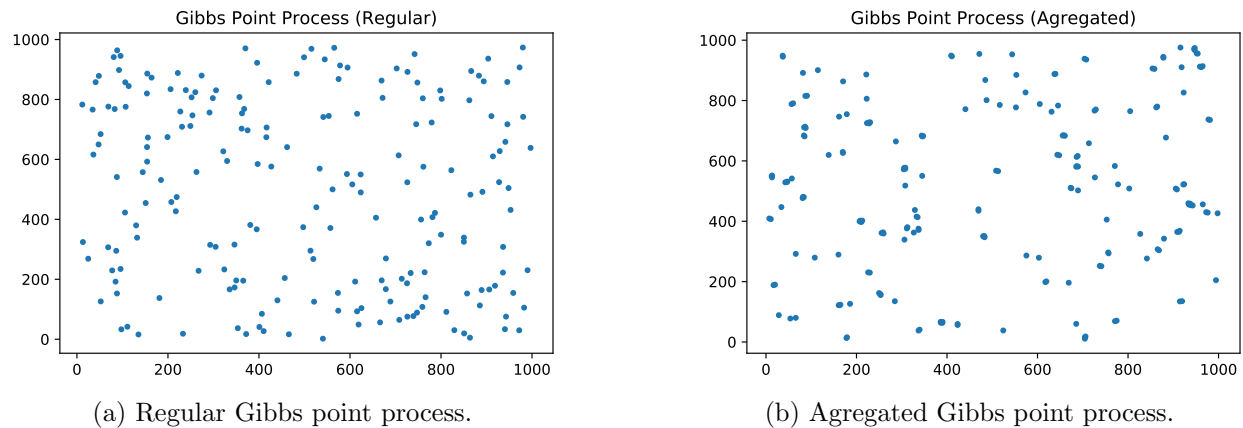


Figure 4: Gibbs point processes (with the same number of points).



```

def ripley(x, y, xmin, xmax, ymin, ymax, edges):
2   # Ripley K and L functions, vals is values of radius
   # this function has border effects!
4   # x, y: coordinates of points
   # xmin, xmax, ymin, ymax: window
6   # edges: values of bins for histogram evaluation

8   # number of points
   nb_points = x.size;

10
   # compute pairwise distances
12   P = np.transpose(np.vstack((x,y)));
   d = pdist(P);

14
   # compute cumulative histogram
16   h,edges = np.histogram(d, edges);
   H = np.cumsum(h);

18
   # normalization of K
20   K = 2*H/nb_points;
   area = (xmax-xmin) * (ymax-ymin);
22   density = float(nb_points) / area;
   K = K / density;

24
   # L
26   L = np.sqrt(K/np.pi);

28
   # edge values
   vals = edges[:-1] + np.diff(edges);

30
   return K, L, vals

```

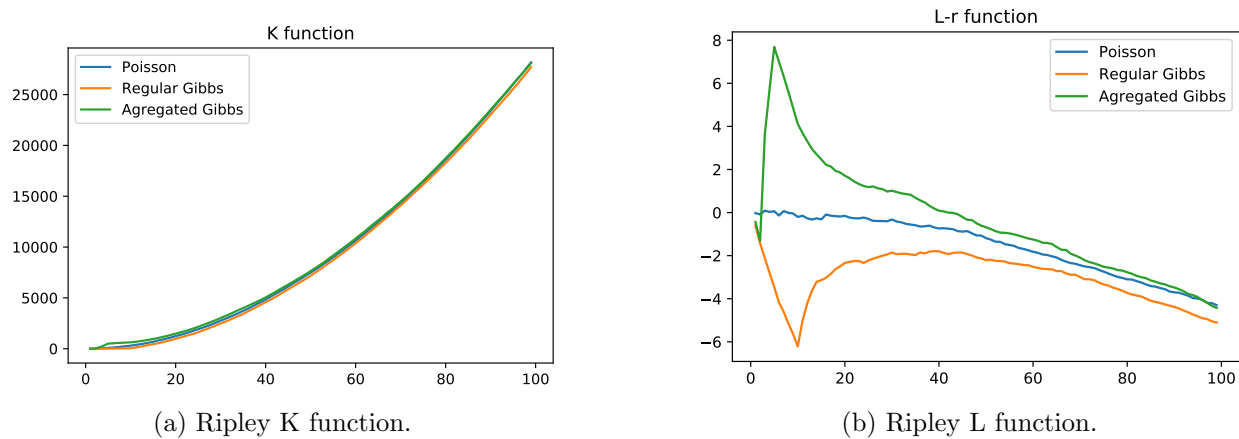


Figure 5: Ripley functions.

1.6 Marked Point Process

An illustration is presented in Fig.6. The algorithm simply consists in adding two new random variables in order to generate the radius and the color of each point.



```

1 def marked(nb_points, xmin, xmax, ymin, ymax):
2     """
3     marked point process
4     """
5
6     # points
7     x,y = spat_pp.cond_Poisson(nb_points, xmin, xmax, ymin, ymax);
8
9     # first mark: radii
10    sigma = 5;
11    mu = 10;
12    r = sigma * np.random.randn(nb_points) + mu;
13    r[r<0.1] = 0.1;
14
15    # second mark: colors
16    nb_colors = 10;
17    c = np.random.randint(nb_colors, size = nb_points);
18
19    # plot
20    plt.scatter(x, y, r**2, c, alpha=.5);
21
22    # save pdf figure
23    plt.savefig("marked.pdf");
24
25 nb_points = 100;
26 N=100;
27 marked(nb_points, 0, N, 0, N);

```

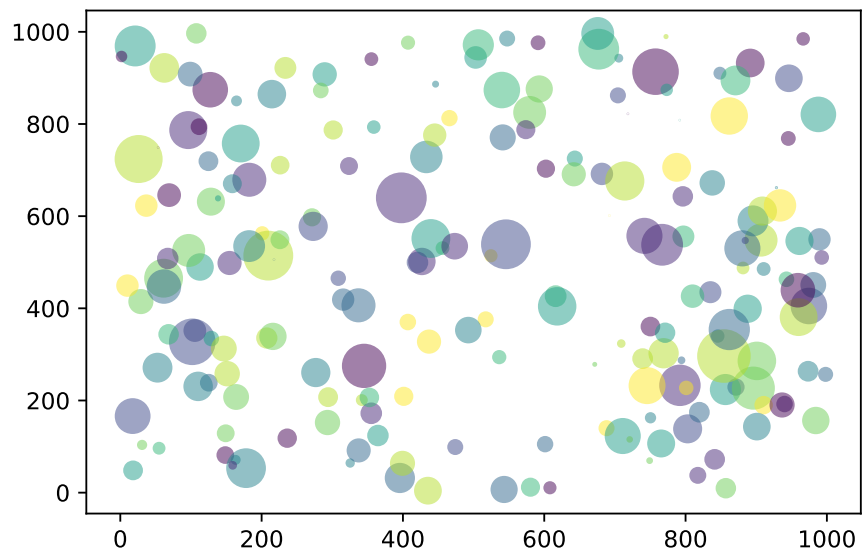


Figure 6: Marked point process.