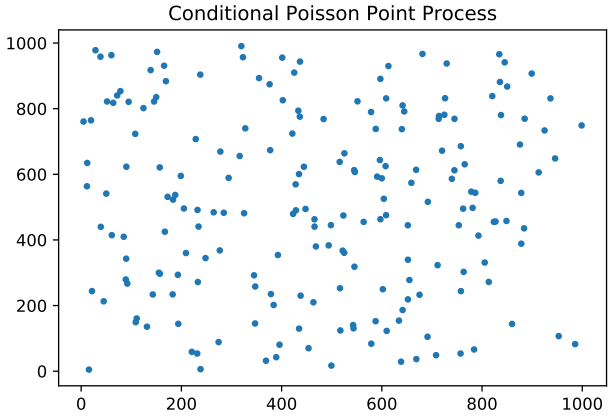
1 Python correction

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import pdist
```

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.



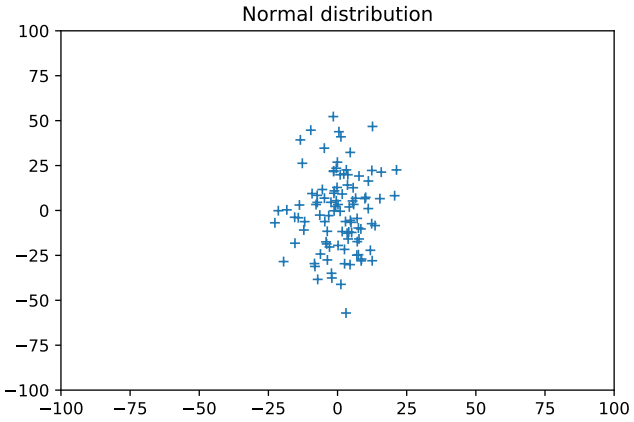
```python
def cond_Poisson(nb_points, xmin, xmax, ymin, ymax):
    # Conditional Poisson Point Process
    # uniform distribution
    # nb_points: number of points
    # xmin, xmax, ymin, ymax: defined the domain (window)
    x = xmin + (xmax-xmin)*np.random.rand(nb_points)
    y = ymin + (ymax-ymin)*np.random.rand(nb_points);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

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

1.2 Normal distribution

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

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

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

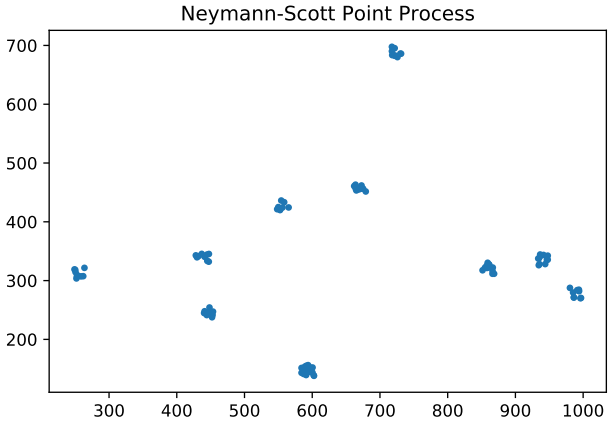


1.3 Neyman-Scott Process

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

```python
def neyman_scott(nRoot, xmin, xmax, ymin, ymax, lambdaS, rSon):
    # Neyman-scott process simulation
    # nRoot: number of agregates
    # xmin, xmax, ymin, ymax: domain
    # lambdaS: number of points. lambda is a density, S is the spatial domain
    # rSon: radius around agregate (points are distributed in a square)

    # number of sons, follows random law
    nSons = poisson.rvs(lambdaS, size=nRoot);
    # results
    x=[];
    y=[];
    # father points coordinates
    xf,yf = cond_Poisson(nRoot, xmin, xmax, ymin, ymax);
    for i in range(nRoot):
        # loop over all agregates
        xs,ys = cond_Poisson(nSons[i], xf[i]-rSon, xf[i]+rSon, yf[i]-rSon , yf[i]+rSon);
        x = np.concatenate((x,xs), axis=0);
        y = np.concatenate((y,ys), axis=0);

    return x,y
```

Figure 3: Neyman-Scott point process, with lambdaS=10 and rSon=10.



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 agregated point process.

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

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

    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.

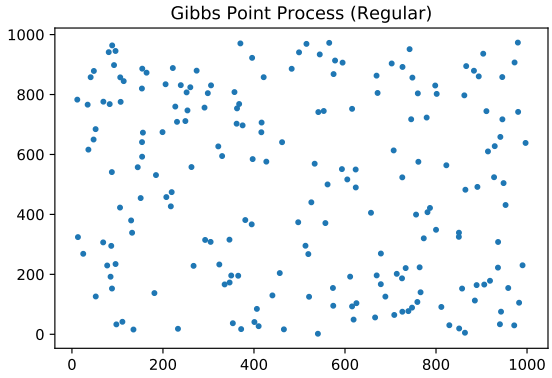
```
def energy(P, eFunction=exampleEnergyFunction):
    """
    This computes the energy in the set of points P, with the energy
       ↪ function
    given as a parameter.
    return a float value
    """
    P = np.transpose(P);
    d = pdist(P);
    e = eFunction(d);
    return np.sum(e);

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

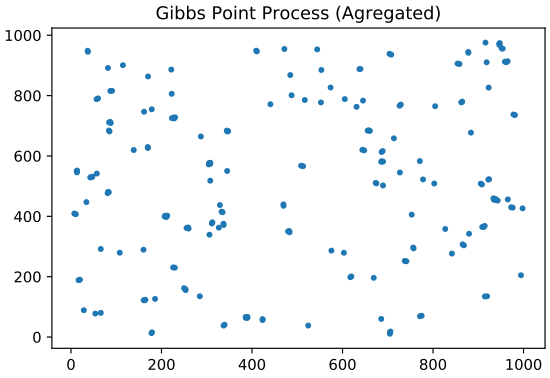
```
def gibbs(nb_points, xmin, xmax, ymin, ymax, nbiter, eFunction=
    ↪ exampleEnergyFunction):
    """
    Gibbs point process
    xmin, xmax, ymin, ymax represents the spatial window
    nb_points: number of generated points
    nbiter: number of iterations
    returns (x,y) coordinates of the points
    """

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

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

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

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

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

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

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

(a) Regular Gibbs point process.



(b) Agregated Gibbs point process.



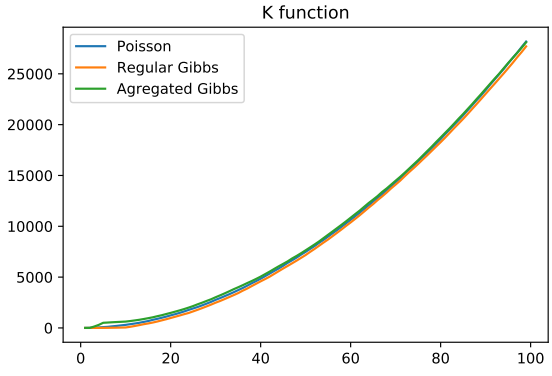
1.5 Ripley functions

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`.

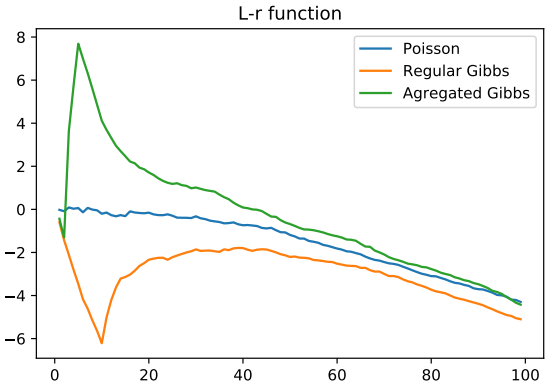
```python
def ripley(x, y, xmin, xmax, ymin, ymax, edges):
    # Ripley K and L functions, vals is values of radius
    # this function has border effects!
    # x, y: coordinates of points
    # xmin, xmax, ymin, ymax: window
    # edges: values of bins for histogram evaluation

    # number of points
    nb_points = x.size;

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

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

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

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

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

    return K, L, vals
```

Figure 5: Ripley functions.

(a) Ripley K function.



(b) Ripley L function.



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.

```
def marked(nb_points, xmin, xmax, ymin, ymax):
    """
    marked point process
    """

    # points
    x,y = cond_Poisson(nb_points, xmin, xmax, ymin, ymax);

    # first mark: radii
    sigma = 5;
    mu = 10;
    r = sigma * np.random.randn(nb_points) + mu;
    r[r<0.1] = 0.1;

    # second mark: colors
    nb_colors = 10;
    c = np.random.randint(nb_colors,size = nb_points);

    # plot
    plt.scatter(x, y, r**2, c, alpha=.5);

    # save pdf figure
    plt.savefig("marked.pdf");

nb_points = 100;
N=100;
marked(nb_points, 0, N, 0, N);
```

Figure 6: Marked point process.