```python
class Env_cellular():
    def __init__(self, fd, Ts, n_x, n_y, L, C, maxM, min_dis, max_dis, max_p, p_n,
power_num):
        self.fd = fd
        self.Ts = Ts
        self.n_x = n_x
        self.n_y = n_y
        self.L = L
        self.C = C
        self.maxM = maxM   # user number in one BS
        self.min_dis = min_dis #km
        self.max_dis = max_dis #km
        self.max_p = max_p #dBm
        self.p_n = p_n     #dBm
        self.power_num = power_num

        self.c = 3*self.L*(self.L+1) + 1 # adjascent BS
        self.K = self.maxM * self.c # maximum adjascent users, including itself
#        self.state_num = 2*self.C + 1    #  2*C + 1
        self.state_num = 3*self.C + 2    #  C + 1
        self.N = self.n_x * self.n_y # BS number
        self.M = self.N * self.maxM # maximum users
        self.W = np.ones((self.M), dtype = dtype)          #[M]
        self.sigma2 = 1e-3*pow(10., self.p_n/10.)
        self.maxP = 1e-3*pow(10., self.max_p/10.)
        self.p_array, self.p_list = self.generate_environment()

    def get_power_set(self, min_p):
        power_set = np.hstack([np.zeros((1), dtype=dtype), 1e-3*pow(10., np.linspace(min_p,
self.max_p, self.power_num-1)/10.)])
        return power_set

    def set_Ns(self, Ns):
        self.Ns = int(Ns)
```

The selected text is the beginning of the `Env_cellular` class definition. The `__init__` method is the constructor for the class and takes several arguments to initialize the object's attributes. These arguments include `fd`, `Ts`, `n_x`, `n_y`, `L`, `C`, `maxM`, `min_dis`, `max_dis`, `max_p`, `p_n`, and `power_num`. These attributes represent various parameters of the cellular environment such as Doppler frequency, symbol period, number of cells in x and y directions, maximum number of users in one base station, minimum and maximum distance between user and base station, maximum transmit power, AWGN power, and number of power levels.

The constructor also calculates several derived attributes such as the number of adjacent base stations (`self.c`), the maximum number of adjacent users including itself (`self.K`), the total number of base stations (`self.N`), and the maximum number of users (`self.M`). It also initializes some other attributes such as the weight vector (`self.W`), noise power (`self.sigma2`), maximum transmit power (`self.maxP`), and calls the `generate_environment` method to generate the environment.

The `get_power_set` method takes a minimum power level as an argument and returns an array of power levels from minimum to maximum with a specified number of levels. The `set_Ns` method takes an integer argument and sets the attribute `self.Ns` to that value.

```
def generate_H_set(self):
    '''
    Jakes model
    '''
    H_set = np.zeros([self.M,self.K,self.Ns], dtype=dtype)
    pho = np.float32(scipy.special.k0(2*np.pi*self.fd*self.Ts))
    H_set[:,:,0] = np.kron(np.sqrt(0.5*(np.random.randn(self.M, 
self.c)**2+np.random.randn(self.M, self.c)**2)), np.ones((1,self.maxM), dtype=np.int32))
    for i in range(1,self.Ns):
        H_set[:,:,i] = H_set[:,:,i-1]*pho + np.sqrt((1.-
pho**2)*0.5*(np.random.randn(self.M, self.K)**2+np.random.randn(self.M, self.K)**2))
    path_loss = self.generate_path_loss()
    H2_set = np.square(H_set) * np.tile(np.expand_dims(path_loss, axis=2), [1,1,self.Ns])
    return H2_set
```

The selected text defines the generate_H_set method of the Env_cellular class. This method generates a set of channel gains between users and base stations using the Jakes model. The method first initializes an array H_set of shape (self.M, self.K, self.Ns) with all zeros. It then calculates the value of pho using the Bessel function of the second kind (scipy.special.k0) with an argument that depends on the Doppler frequency (self.fd) and symbol period (self.Ts). The first “slice” of H_set along the third dimension is then filled with values generated using a Kronecker product between a square root of the sum of two squared normally distributed random variables and an array of ones. For each subsequent “slice” along the third dimension, the values are calculated as a weighted sum of the previous slice and a square root of the sum of two squared normally distributed random variables. The weights are pho and sqrt((1-pho^2)/2) respectively.

The method then calls the generate_path_loss method to calculate path loss and multiplies the square of H_set element-wise with a tiled version of path_loss to obtain H2_set. Finally, it returns H2_set.

```
def generate_environment(self):
    path_matrix = self.M*np.ones((self.n_y + 2*self.L, self.n_x + 2*self.L, self.maxM),
dtype = np.int32)
    for i in range(self.L, self.n_y+self.L):
        for j in range(self.L, self.n_x+self.L):
            for l in range(self.maxM):
                path_matrix[i,j,l] = ((i-self.L)*self.n_x + (j-self.L))*self.maxM + l
    p_array = np.zeros((self.M, self.K), dtype = np.int32)
    for n in range(self.N):
        i = n//self.n_x
        j = n%self.n_x
        Jx = np.zeros((0), dtype = np.int32)
        Jy = np.zeros((0), dtype = np.int32)
        for u in range(i-self.L, i+self.L+1):
            v = 2*self.L+1-np.abs(u-i)
            jx = j - (v-i%2)//2 + np.linspace(0, v-1, num = v, dtype = np.int32) + self.L
            jy = np.ones((v), dtype = np.int32)*u + self.L
            Jx = np.hstack((Jx, jx))
            Jy = np.hstack((Jy, jy))
        for l in range(self.maxM):
            for k in range(self.c):
                for u in range(self.maxM):
                    p_array[n*self.maxM+l,k*self.maxM+u] = path_matrix[Jy[k],Jx[k],u]
    p_main = p_array[:,(self.c-1)//2*self.maxM:(self.c+1)//2*self.maxM]
    for n in range(self.N):
        for l in range(self.maxM):
            temp = p_main[n*self.maxM+l,l]
            p_main[n*self.maxM+l,l] = p_main[n*self.maxM+l,0]
            p_main[n*self.maxM+l,0] = temp
    p_inter = np.hstack([p_array[:,:(self.c-1)//2*self.maxM],
p_array[:,(self.c+1)//2*self.maxM:]])
    p_array =  np.hstack([p_main, p_inter])
    p_list = list()
    for m in range(self.M):
        p_list_temp = list()
        for k in range(self.K):
            p_list_temp.append([p_array[m,k]])
        p_list.append(p_list_temp)
    return p_array, p_list
```

The selected text defines the generate_environment method of the Env_cellular class. This method generates the environment by creating two arrays: p_array and p_list. The method first initializes a path_matrix array of shape (self.n_y + 2*self.L, self.n_x + 2*self.L, self.maxM) with all elements equal to self.M. It then fills this array with values calculated using nested loops over the indices i, j, and l.

The method then initializes the p_array array of shape (self.M, self.K) with all zeros. It uses nested loops over the indices n, i, j, u, l, k, and another u to fill this array with values calculated using the previously defined path_matrix and several other intermediate variables.

The method then creates two new arrays: p_main and p_inter. The former is a "slice" of p_array along the second dimension, while the latter is created by horizontally stacking two "slices" of p_array along the second dimension. The method then swaps some elements of p_main and horizontally stacks it with p_inter to obtain an updated version of p_array.

The method then initializes an empty list p_list and uses a nested loop over the indices m and k to append lists of lists to it. Each inner list contains a single element from the corresponding row of p_array.

Finally, the method returns both p_array and p_list. These arrays represent the environment generated by this method.

```
def generate_path_loss(self):
    p_tx = np.zeros((self.n_y, self.n_x))
    p_ty = np.zeros((self.n_y, self.n_x))
    p_rx = np.zeros((self.n_y, self.n_x, self.maxM))
    p_ry = np.zeros((self.n_y, self.n_x, self.maxM))
    dis_rx = np.random.uniform(self.min_dis, self.max_dis, size = (self.n_y, self.n_x,
self.maxM))
    phi_rx = np.random.uniform(-np.pi, np.pi, size = (self.n_y, self.n_x, self.maxM))
    for i in range(self.n_y):
        for j in range(self.n_x):
            p_tx[i,j] = 2*self.max_dis*j + (i%2)*self.max_dis
            p_ty[i,j] = np.sqrt(3.)*self.max_dis*i
            for k in range(self.maxM):
                p_rx[i,j,k] = p_tx[i,j] + dis_rx[i,j,k]*np.cos(phi_rx[i,j,k])
                p_ry[i,j,k] = p_ty[i,j] + dis_rx[i,j,k]*np.sin(phi_rx[i,j,k])
    dis = 1e10 * np.ones((self.p_array.shape[0], self.K), dtype = dtype)
    lognormal = np.random.lognormal(size = (self.p_array.shape[0], self.K), sigma = 8)
    for k in range(self.p_array.shape[0]):
        for i in range(self.c):
            for j in range(self.maxM):
                if self.p_array[k,i*self.maxM+j] < self.M:
                    bs = self.p_array[k,i*self.maxM+j]//self.maxM
                    dx2 =
np.square((p_rx[k//self.maxM//self.n_x][k//self.maxM%self.n_x][k%self.maxM]-
p_tx[bs//self.n_x][bs%self.n_x]))
                    dy2 =
np.square((p_ry[k//self.maxM//self.n_x][k//self.maxM%self.n_x][k%self.maxM]-
p_ty[bs//self.n_x][bs%self.n_x]))
                    distance = np.sqrt(dx2 + dy2)
                    dis[k,i*self.maxM+j] = distance
    path_loss = lognormal*pow(10., -(120.9 + 37.6*np.log10(dis))/10.)
    return path_loss
```

The selected text defines the generate_path_loss method of the Env_cellular class. This method calculates the path loss between users and base stations. The method first initializes several arrays of zeros: p_tx, p_ty, p_rx, and p_ry. It also generates two arrays of random values: dis_rx and phi_rx. These arrays represent the distance and angle between users and their serving base station.

The method then uses nested loops over the indices i, j, and k to fill the previously initialized arrays with values calculated using the random values and several other intermediate variables.

The method then initializes two more arrays: dis with all elements equal to a large value (1e10) and lognormal with log-normally distributed random values. It uses another set of nested loops over the indices k, i, and j to fill the dis array with values calculated using the previously defined arrays and several other intermediate variables.

Finally, the method calculates the path loss as an element-wise product between the lognormal array and a power of 10 raised to a negative linear combination of constants and the logarithm of the dis array. It returns this path loss array.

```python
def calculate_rate(self, P):
    '''
    Calculate C[t]
    1.H2[t]
    2.p[t]
    '''
    maxC = 1000.
    H2 = self.H2_set[:,:,self.count]
    p_extend = np.concatenate([P, np.zeros((1), dtype=dtype)], axis=0)
    p_matrix = p_extend[self.p_array]
    path_main = H2[:,0] * p_matrix[:,0]
    path_inter = np.sum(H2[:,1:] * p_matrix[:,1:], axis=1)
    sinr = np.minimum(path_main / (path_inter + self.sigma2), maxC)    #capped sinr
    rate = self.W * np.log2(1. + sinr)

    sinr_norm_inv = H2[:,1:] / np.tile(H2[:,0:1], [1,self.K-1])
    sinr_norm_inv = np.log2(1. + sinr_norm_inv)   # log representation
    rate_extend = np.concatenate([rate, np.zeros((1), dtype=dtype)], axis=0)
    rate_matrix = rate_extend[self.p_array]
    '''
    Calculate reward, sum-rate
    '''
    sum_rate = np.mean(rate)
    reward_rate = rate + np.sum(rate_matrix, axis=1)
    return p_matrix, rate_matrix, reward_rate, sum_rate
```

The selected text defines the calculate_rate method of the Env_cellular class. This method calculates the rates, reward rates, and sum rate for a given transmit power allocation. The method takes an array P as an argument and first sets a maximum value for the capped SINR (maxC) and selects the current “slice” of self.H2_set along the third dimension. It then concatenates this array with an array of zeros and uses it to index into self.p_array. The method then calculates the main and interference paths using these values and element-wise operations. It calculates the SINR as the element-wise minimum between the ratio of these two arrays and maxC. The rate is then calculated as a weighted logarithm base 2 of one plus this SINR.

The method then calculates the inverse of the normalized SINR by dividing all but the first column of H2 element-wise by its first column and taking the logarithm base 2 of one plus this result. It also concatenates the rate array with an array of zeros and uses it to index into self.p_array. The method then calculates the sum rate as the mean of the rate array and the reward rate as the sum of this array along its second dimension.

Finally, it returns all these calculated values: p_matrix, rate_matrix, reward_rate, and sum_rate.

```
def generate_next_state(self, H2, p_matrix, rate_matrix):
    '''
    Generate state for actor
    ranking
    state including:
    1.sinr_norm_inv[t+1]   [M,C]  sinr_norm_inv
    2.p[t]         [M,C+1]  p_matrix
    3.C[t]         [M,C+1]  rate_matrix  optional
    '''
    sinr_norm_inv = H2[:,1:] / np.tile(H2[:,0:1], [1,self.K-1])
    sinr_norm_inv = np.log2(1. + sinr_norm_inv)   # log representation
    indices1 = np.tile(np.expand_dims(np.linspace(0, p_matrix.shape[0]-1,
num=p_matrix.shape[0], dtype=np.int32), axis=1),[1,self.C])
    indices2 = np.argsort(sinr_norm_inv, axis = 1)[:,-self.C:]
    sinr_norm_inv = sinr_norm_inv[indices1, indices2]
    p_last = np.hstack([p_matrix[:,0:1], p_matrix[indices1, indices2+1]])
    rate_last = np.hstack([rate_matrix[:,0:1], rate_matrix[indices1, indices2+1]])

#        s_actor_next = np.hstack([sinr_norm_inv, p_last])
    s_actor_next = np.hstack([sinr_norm_inv, p_last, rate_last])
    '''
    Generate state for critic
    '''
    s_critic_next = H2
    return s_actor_next, s_critic_next
```

The selected text defines the generate_next_state method of the Env_cellular class. This method generates the next state of the environment for both the actor and critic. The method takes three arguments: H2, p_matrix, and rate_matrix. These represent the current channel gains, transmit power levels, and rates respectively.

The method first calculates the inverse of the normalized signal-to-interference-plus-noise ratio (SINR) by dividing all but the first column of H2 element-wise by its first column and taking the logarithm base 2 of one plus this result. It then sorts this array along the second dimension and selects the last self.C columns. The method also creates two new arrays: p_last and rate_last. The former is created by horizontally stacking the first column of p_matrix with a “slice” of itself along the second dimension. The latter is created in a similar way using rate_matrix.

The method then horizontally stacks these three arrays to obtain s_actor_next, which represents the next state for the actor. It also sets s_critic_next to be equal to H2, which represents the next state for the critic. Finally, it returns both s_actor_next and s_critic_next.

```python
def reset(self):
    self.count = 0
    self.H2_set = self.generate_H_set()
    P = np.zeros([self.M], dtype=dtype)

    p_matrix, rate_matrix, _, _ = self.calculate_rate(P)
    H2 = self.H2_set[:,:,self.count]
    s_actor, s_critic = self.generate_next_state(H2, p_matrix, rate_matrix)
    return s_actor, s_critic

def step(self, P):
    p_matrix, rate_matrix, reward_rate, sum_rate = self.calculate_rate(P)
    self.count = self.count + 1
    H2_next = self.H2_set[:,:,self.count]
    s_actor_next, s_critic_next = self.generate_next_state(H2_next, p_matrix, rate_matrix)
    return s_actor_next, s_critic_next, reward_rate, sum_rate

def calculate_sumrate(self, P):
    maxC = 1000.
    H2 = self.H2_set[:,:,self.count]
    p_extend = np.concatenate([P, np.zeros((1), dtype=dtype)], axis=0)
    p_matrix = p_extend[self.p_array]
    path_main = H2[:,0] * p_matrix[:,0]
    path_inter = np.sum(H2[:,1:] * p_matrix[:,1:], axis=1)
    sinr = np.minimum(path_main / (path_inter + self.sigma2), maxC)    #capped sinr
    rate = self.W * np.log2(1. + sinr)
    sum_rate = np.mean(rate)
    return sum_rate

def step__(self, P):
    reward_rate = list()
    for p in P:
        reward_rate.append(self.calculate_sumrate(p))
    self.count = self.count + 1
    H2_next = self.H2_set[:,:,self.count]
    return H2_next, reward_rate

def reset__(self):
    self.count = 0
    self.H2_set = self.generate_H_set()
    H2 = self.H2_set[:,:,self.count]
    return H2
```

The selected text defines several methods of the Env_cellular class: reset, step, calculate_sumrate, step_, and reset_. The reset method resets the environment to its initial state. It sets the attribute self.count to zero, calls the generate_H_set method to generate a new set of channel gains, and initializes an array P of zeros. It then calls the calculate_rate method with this array as an argument and uses its return values along with the first “slice” of self.H2_set along the third dimension to call the generate_next_state method. The return values of this method are then returned by the reset method.

The step method takes an array P as an argument and advances the environment by one time step. It calls the calculate_rate method with this array as an argument and uses its return values to update the attributes self.count and H2_next. It then calls the generate_next_state method with these updated values and returns its return values along with the reward rate and sum rate calculated by the calculate_rate method.

The calculate_sumrate method takes an array P as an argument and calculates the sum rate for this transmit power allocation. It first sets a maximum value for the capped SINR (maxC) and selects the current “slice” of self.H2_set along the third dimension. It then concatenates this array with an array of zeros and uses it to index into self.p_array. The method then calculates the main and interference paths using these values and element-wise operations. It calculates the SINR as the element-wise minimum between the ratio of these two arrays and maxC. The rate is then calculated as a weighted logarithm base 2 of one plus this SINR. Finally, it returns the mean of this rate array.