Forest_fire

November 11, 2024

[ ]:
```python
import numpy as np

def complementary_CDF(f, f_max):
    """
    Function to return the complementary cumulative distribution function.

    Parameters
    ==========
    f : Sequence of values (as they occur, non necessarily sorted).
    f_max : Integer. Maximum possible value for the values in f.
    """

    num_events = len(f)
    s = np.sort(np.array(f)) / f_max  # Sort f in ascending order.
    c = np.array(np.arange(num_events, 0, -1)) / (num_events)  # Descending.

    c_CDF = c
    s_rel = s

    return c_CDF, s_rel
```

[ ]:
```python
def grow_trees(forest, p):
    """
    Function to pgrow new trees in the forest.

    Parameters
    ==========
    forest : 2-dimensional array.
    p : Probability for a tree to be generated in an empty cell.
    """

    Ni, Nj = forest.shape  # Dimensions of the forest.

    new_trees = np.random.rand(Ni, Nj)

    new_trees_indices = np.where(new_trees <= p)
    forest[new_trees_indices] = 1
```

```
    return forest
```

```
[ ]: def propagate_fire(forest, i0, j0):
    """
    Function to propagate the fire on a populated forest.

    Parameters
    ==========
    forest : 2-dimensional array.
    i0 : First index of the cell where the fire occurs.
    j0 : Second index of the cell where the fire occurs.
    """

    Ni, Nj = forest.shape  # Dimensions of the forest.

    fs = 0  # Initialize fire size.

    if forest[i0, j0] == 1:
        active_i = [i0]  # Initialize the list.
        active_j = [j0]  # Tnitialize the list.
        forest[i0, j0] = -1  # Sets the tree on fire.
        fs += 1  # Update fire size.

        while len(active_i) > 0:
            next_i = []
            next_j = []
            for n in np.arange(len(active_i)):
                # Coordinates of cell up.
                i = (active_i[n] + 1) % Ni
                j = active_j[n]
                # Check status
                if forest[i, j] == 1:
                    next_i.append(i)  # Add to list.
                    next_j.append(j)  # Add to list.
                    forest[i, j] = -1  # Sets the current tree on fire.
                    fs += 1  # Update fire size.

                # Coordinates of cell down.
                i = (active_i[n] - 1) % Ni
                j = active_j[n]
                # Check status
                if forest[i, j] == 1:
                    next_i.append(i)  # Add to list.
                    next_j.append(j)  # Add to list.
                    forest[i, j] = -1  # Sets the current tree on fire.
                    fs += 1  # Update fire size.
```

```python
                # Coordinates of cell left.
                i = active_i[n]
                j = (active_j[n] - 1) % Nj
                # Check status
                if forest[i, j] == 1:
                    next_i.append(i)  # Add to list.
                    next_j.append(j)  # Add to list.
                    forest[i, j] = -1  # Sets the current tree on fire.
                    fs += 1  # Update fire size.

                # Coordinates of cell right.
                i = active_i[n]
                j = (active_j[n] + 1) % Nj
                # Check status
                if forest[i, j] == 1:
                    next_i.append(i)  # Add to list.
                    next_j.append(j)  # Add to list.
                    forest[i, j] = -1  # Sets the current tree on fire.
                    fs += 1  # Update fire size.

            active_i = next_i
            active_j = next_j

    return fs, forest
```

[ ]:

```python
#main.py

import matplotlib.pyplot as plt
from grow_trees import grow_trees
from propagate_fire import propagate_fire
from complementary_CDF import complementary_CDF


N_values = [16, 32, 64, 128, 256, 512, 1024]
p = 0.01  # prob. of fire propagating
f = 0.2 # prob. of one tree fired ( lightning occurs)
repeats = 10


alpha_results = []
target_num_fires = 300
num_fires = 0

for N in N_values:

    if N == 1024:
        repeats = 2
```

```python
alpha_results_for_N = []
r = 0 # count repeat times  for debug

for _ in range(repeats):

    forest = np.zeros([N, N])  # Empty forest.
    fire_size = []  # Empty list of fire sizes.
    fire_history = []  # Empty list of fire history.

    num_fires = 0
    while num_fires < target_num_fires:

        forest = grow_trees(forest, p)  # Grow new trees.
        Ni, Nj = forest.shape
        p_lightning = np.random.rand()

        if p_lightning < f:  # Lightning occurs.
            i0 = np.random.randint(Ni)
            j0 = np.random.randint(Nj)

            fs, forest = propagate_fire(forest, i0, j0) # fs = firesize

            if fs > 0:
                fire_size.append(fs)
                num_fires += 1

            fire_history.append(fs)

        else:
            fire_history.append(0)

        forest[np.where(forest == -1)] = 0

    print(f'N = {N}', f'Target of {target_num_fires} fire events reached')

    c_CDF, s_rel = complementary_CDF(fire_size, forest.size)
    min_rel_size = 1e-3
    max_rel_size = 1e-1

    is_min = np.searchsorted(s_rel, min_rel_size)
    is_max = np.searchsorted(s_rel, max_rel_size)

    # Note!!! The linear dependence is between the logarithms
    fit_result = np.polyfit(np.log(s_rel[is_min:is_max]),
                np.log(c_CDF[is_min:is_max]), 1)
    beta = fit_result[0]
```

```
        alpha = 1 - beta
        alpha_results_for_N.append(alpha)
        r  = r + 1
        print(f'Repeat times = {r}')

    alpha_mean = np.mean(alpha_results_for_N )
    alpha_std = np.std(alpha_results_for_N )
    alpha_results.append((alpha_mean, alpha_std))
    print(f'After {r} times repeat, empirical cCDF has an exponent alpha = {alpha_results[-1]}')


# # Note loglog plot!
# plt.loglog(s_rel, c_CDF, ".-", color='k', markersize=5, linewidth=0.5)
# plt.title('Empirical cCDF')
# plt.xlabel('relative size')
# plt.ylabel('c CDF')
# plt.show()

inv_N = 1 / np.array(N_values)
alpha_means = [result[0] for result in alpha_results]
alpha_errors = [result[1] for result in alpha_results]

# Q1 Extrapolate results to 1/N : 0 -->> find fit function
inv_N_fit = inv_N[:7]
alpha_means_fit = alpha_means[:7]
fit_result = np.polyfit(inv_N_fit, alpha_means_fit, 1)
a, b = fit_result
fit_line = a * inv_N + b

xticks_positions = inv_N
xticks_labels = [f'$\\frac{{1}}{{{N}}}$' for N in N_values]  #   LaTeX

plt.figure(figsize=(12, 6))
plt.plot(inv_N, fit_line, 'k--', label=f'Linear fit:   = {a:.4f} (1/N) + {b:.4f}')  #  line of fit funcion
plt.errorbar(inv_N, alpha_means, yerr=alpha_errors, fmt='o', color='cyan', ecolor='black', capsize=5, label='Alpha values with error bars')
plt.xticks(xticks_positions, xticks_labels)
plt.xticks(xticks_positions, xticks_labels, rotation=45, ha='right', fontsize=10)

plt.xlabel('1/N')
plt.ylabel('Exponent  ')
plt.title('Dependence of Power-law Exponent on 1/N')
plt.grid(True)
plt.legend()
```

```python
plt.tight_layout()
plt.savefig('power_law_exponent_vs_inv_N.png', format='png', dpi=300)
plt.show()
```