

```
In [ ]: import networkx as nx
import numpy as np
import matplotlib.pyplot as plt
from collections import Counter, defaultdict
```

# 1. ER phase transitions

## a) ER graph component sizes

```
In [ ]: def analyze_er_graph_components(n, p_range, graphs_per_p=20):
    means, sds = [], []
    for p in p_range:
        print('Analyzing ER graphs for p={}'.format(p))
        comp_sizes = []
        for i in range(graphs_per_p):
            G = nx.generators.fast_gnp_random_graph(n, p, seed=i)
            comp_sizes.append(max([len(comp) for comp in nx.connected_components(G)]))
        means.append(np.mean(comp_sizes))
        sds.append(np.std(comp_sizes))

    return np.array(means), np.array(sds)
```

```
In [ ]: n = 5000
p_range = np.logspace(-5, -2, num=20)
means, sds = analyze_er_graph_components(n, p_range)
```

```
Analyzing ER graphs for p=1e-05
Analyzing ER graphs for p=1.438449888287663e-05
Analyzing ER graphs for p=2.06913808111479e-05
Analyzing ER graphs for p=2.9763514416313192e-05
Analyzing ER graphs for p=4.281332398719396e-05
Analyzing ER graphs for p=6.158482110660267e-05
Analyzing ER graphs for p=8.858667904100833e-05
Analyzing ER graphs for p=0.00012742749857031334
Analyzing ER graphs for p=0.00018329807108324357
Analyzing ER graphs for p=0.00026366508987303583
Analyzing ER graphs for p=0.000379269019073225
Analyzing ER graphs for p=0.0005455594781168515
```

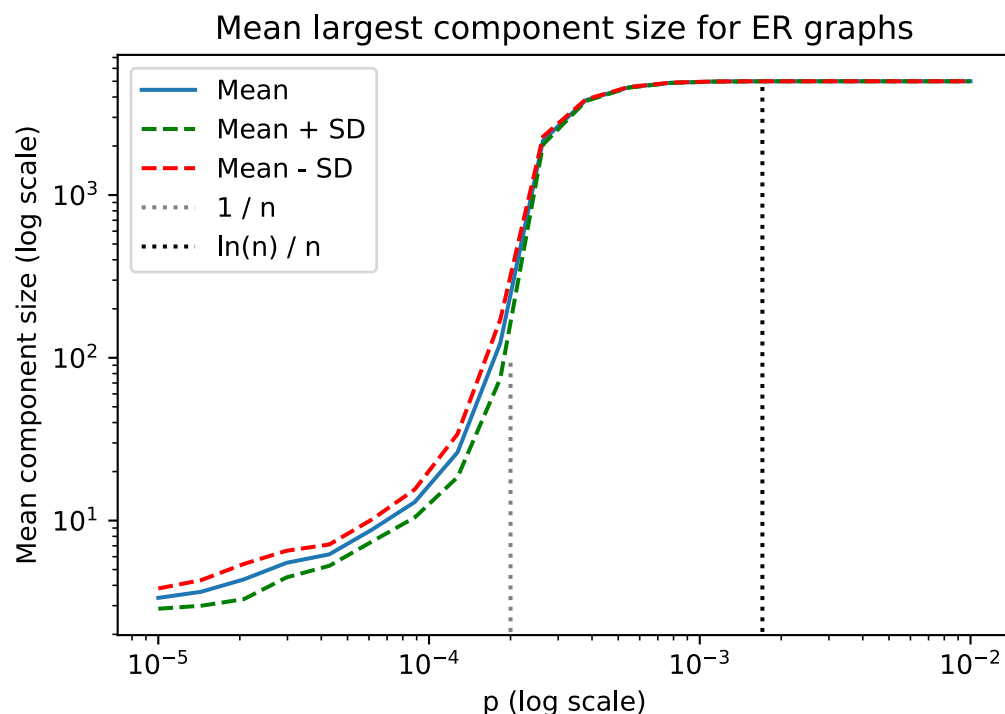
Analyzing ER graphs for  $p=0.0007847599703514606$   
 Analyzing ER graphs for  $p=0.0011288378916846883$   
 Analyzing ER graphs for  $p=0.001623776739188721$   
 Analyzing ER graphs for  $p=0.002335721469090121$   
 Analyzing ER graphs for  $p=0.003359818286283781$   
 Analyzing ER graphs for  $p=0.004832930238571752$   
 Analyzing ER graphs for  $p=0.0069519279617756054$   
 Analyzing ER graphs for  $p=0.01$

```
In [ ]: def plot_with_sds(x, y, y_sds, n=5_000):
    plt.plot(x, y)
    plt.plot(x, y - y_sds, c='green', linestyle='dashed')
    plt.plot(x, y + y_sds, c='red', linestyle='dashed')

    # plot vertical lines at 1 / n and ln(n) / n
    plt.vlines(1 / n, 0, 100, linestyle='dotted', colors='gray')
    plt.vlines(np.log(n) / n, 0, 4900, linestyle='dotted', colors='black')

    plt.yscale('log')
    plt.xscale('log')
    plt.title('Mean largest component size for ER graphs')
    plt.xlabel('p (log scale)')
    plt.ylabel('Mean component size (log scale)')
    plt.legend(['Mean', 'Mean + SD', 'Mean - SD', '1 / n', 'ln(n) / n'])
```

```
In [ ]: plot_with_sds(p_range, means, sds)
```



## b) Interesting values of $p$

From class, we have that when  $p = \lambda / n$ ,  $\lambda > 1$  makes the largest component size converge to  $n$ , while  $\lambda < 1$  makes it converge to  $\ln(n)$ . Here,  $\lambda = 1$  represents a point at which the largest component size is transitioning rapidly, as it has the graph's peak slope. Also, I notice that as  $\lambda$  shrinks less than 1, the size remains low (near  $\ln(n) \approx 8$ ), while as  $\lambda$  grows above 1, the size approaches  $n$ .

Also, we have that when  $p = \lambda \ln(n) / n$ ,  $\lambda < 1$  makes the graph's probability of being connected converge to 0, while  $\lambda > 1$  makes it converge to 1. Here, to the left of  $\ln(n) / n$  (i.e.  $\lambda > 1$ ), the mean component size has not yet converged to  $n$ , but to the right, the mean is equal to  $n$  and the standard deviation is 0, meaning that, as expected, all sampled graphs converge to being connected.

## 2. Fitting power-law exponents

### b) Pareto sampling

In [ ]:

```
def sample_pareto(alpha=1, x_0=1, size=100_000):
    # sample uniformly in [0, 1] and apply the pareto function to each sample
    pareto_func = np.vectorize(lambda u: x_0 / (u**(1 / alpha)))
    u_samples = np.random.sample(size=size)
    return pareto_func(u_samples)
```

```
In [ ]: def compute_pdf(samples):
    # round samples to nearest integer
    sample_ints = np rint(samples).astype(int)

    # count occurrences of each integer
    cts = Counter(sample_ints)
    x = np.array(sorted(cts.keys()))
    px_x = np.array([cts[i] for i in sorted(cts.keys())]) / len(sample_ints)

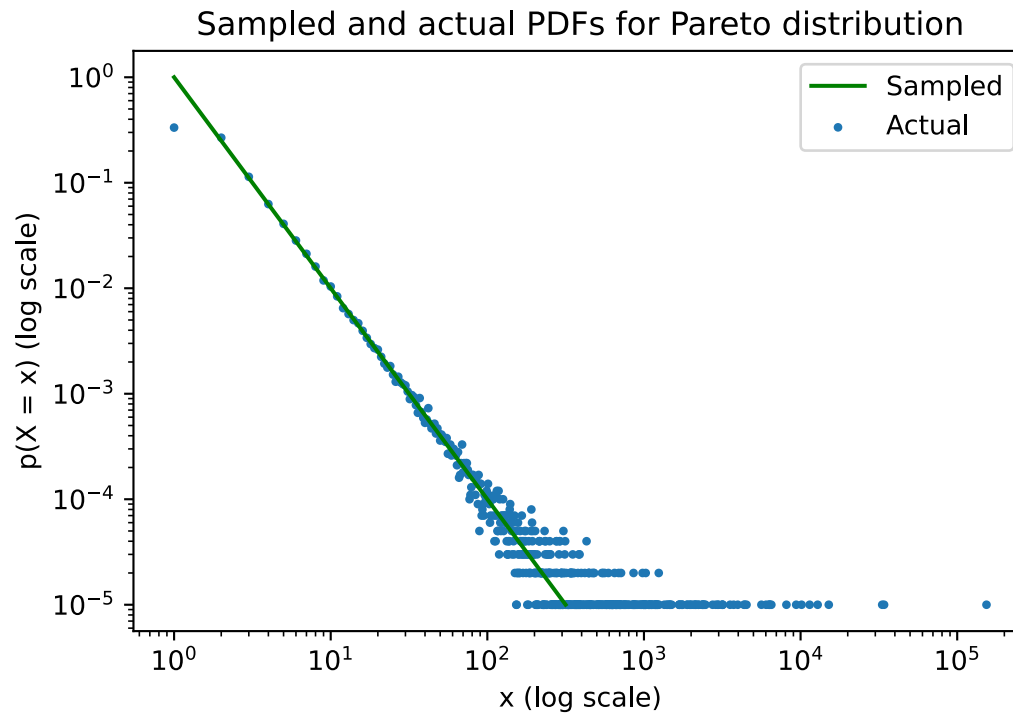
    return x, px_x
```

```
In [ ]: def compute_ccdf(x, px_x):
    # compute running total from right to left
    s = 0
    fbarx_x = np.zeros(x.shape)
    for i in range(len(x) - 1, -1, -1):
        s += px_x[i]
        fbarx_x[i] = s

    return x, fbarx_x
```

```
In [ ]: def plot_observations(x, y, alpha=1):
    plt.scatter(x, y, s=5)
    plt.xscale('log')
    plt.yscale('log')
    y_expected = 1 / x ** (alpha + 1)
    plt.plot(x[y_expected >= 1e-5], y_expected[y_expected >= 1e-5], c='green')
    plt.title('Sampled and actual PDFs for Pareto distribution')
    plt.legend(['Sampled', 'Actual'])
    plt.xlabel('x (log scale)')
    plt.ylabel('p(X = x) (log scale)')
```

```
In [ ]: plot_observations(x, px_x)
```



## d) Estimating alpha

```
In [ ]: def estimate_alpha_pdf_ls(samples):
    x, p_x = compute_pdf(samples)

    x_log = np.log(x)
    y_log = np.log(p_x)

    alpha_pred = -np.polyfit(x_log, y_log, deg=1)[0]
    return alpha_pred
```

```
In [ ]: def estimate_alpha_ccdf_ls(samples):
    x, px_x = compute_pdf(samples)
    x, fbarx_x = compute_ccdf(x, px_x)

    x_log = np.log(x)
    y_log = np.log(fbarx_x)

    # CCDF exponent is (alpha - 1)
```

```
alpha_pred = -np.polyfit(x_log, y_log, deg=1)[0] + 1

return alpha_pred
```

```
In [ ]: def estimate_alpha_mle(samples, x_0=1):
        n = len(samples)

        # from c), MLE is alpha = n / (sum_i(Log(x_i)) - (n * Log(x_0)))
        return n / (np.log(samples).sum() - (n * np.log(x_0)))
```

```
In [ ]: def eval_alpha_estimate(estimate_alpha, name, samples=100):
        preds = []
        for _ in range(samples):
            samples = sample_pareto()
            preds.append(estimate_alpha(samples))
        m, sd = np.mean(preds), np.std(preds)
        print('{}:'.format(name))
        print('Mean: {}, SD: {}'.format(m, sd))
```

```
In [ ]: eval_alpha_estimate(estimate_alpha_pdf_ls, 'Least-squares on PDF');
```

```
Least-squares on PDF:
Mean: 0.9587292458008777, SD: 0.048126100747554156
```

```
In [ ]: eval_alpha_estimate(estimate_alpha_ccdf_ls, 'Least-squares on CCDF');
```

```
Least-squares on CCDF:
Mean: 1.9955482093481538, SD: 0.03403874599398693
```

```
In [ ]: eval_alpha_estimate(estimate_alpha_mle, 'Log-likelihood maximization');
```

Least-squares regression on the CCDF gives a very poor result, while least-squares on the PDF and the log-likelihood MLE both give results somewhat close to the true value. The MLE almost exactly approximates true value, while the PDF least-squares is off by a bit more; also, the MLE has near-0 standard deviation, so since it is also the least biased, it is clearly the best estimate.

note: my numbers don't line up with Dr. Segarra's for i and ii, but he looked through my code during office hours and said that what I'm doing seems right, so we're not sure why my results are different. I plotted the PDF, CCDF, and fitted lines on a log-log graph below, if that helps at all.

```
In [ ]: x, p_x = compute_pdf(sample_pareto())
x, fbarx_x = compute_ccdf(x, p_x)

x_log = np.log(x)

m, c = np.polyfit(x_log, np.log(p_x), deg=1)

plt.plot(x_log, [a * m + c for a in x_log])
plt.scatter(x_log, np.log(p_x), s=5)

m, c = np.polyfit(x_log, np.log(fbarx_x), deg=1)

plt.plot(x_log, [a * m + c for a in x_log])
plt.scatter(x_log, np.log(fbarx_x), s=5)
```

```
Out[ ]: <matplotlib.collections.PathCollection at 0x7f18a828ee50>
```

