res_external

July 8, 2025

[3]:
```
load('fractal_functions.sage')
%display latex
```

1 Compute the Resistance Across Boundary Vertices of H_n

[24]:
```
def compute_single_resistance(graph, v1, v2):
    """
    Computes the effective resistance between two specific vertices v1 and v2
    efficiently using a conjugate gradient solver.

    Args:
        graph: A SageMath graph with edge labels as conductances.
        v1, v2: The labels of the two vertices.

    Returns:scipy.sparse
        float: The effective resistance R(v1, v2).
    """
    N = graph.order()
    vertices = list(graph.vertices())
    v_map = {vertex: i for i, vertex in enumerate(vertices)}

    if v1 not in v_map or v2 not in v_map:
        raise ValueError("One or both vertices not in the graph.")

    i1, i2 = v_map[v1], v_map[v2]

    # --- Step 1: Build the sparse Laplacian Matrix ---
    # Using a sparse matrix is crucial for performance with large graphs.
    # LIL (List of Lists) format is efficient for construction.
    L = lil_matrix((N, N), dtype=float)

    for u, v, conductance in graph.edges(labels=True):
        if conductance is None or conductance <= 1e-9:
            continue

        i, j = v_map[u], v_map[v]
```

```
        if i != j:
            L[i, j] = -conductance
            L[j, i] = -conductance
            L[i, i] += conductance
            L[j, j] += conductance
        else:
            L[i, i] += conductance

    # Convert to CSR (Compressed Sparse Row) format for fast arithmetic
    L_csr = L.tocsr()

    # --- Step 2: Set up and solve the linear system ---

    # Create the current injection vector b
    b = np.zeros(N)
    b[i1] = 1.0
    b[i2] = -1.0

# Pin the last node (remove its row and column)
    pinned_idx = N - 1  # pinning the last vertex
    mask = np.arange(N) != pinned_idx

    L_reduced = L_csr[mask][:, mask]
    b_reduced = b[mask]

    # Solve the reduced system
    f_reduced = spsolve(L_reduced, b_reduced)

    # Reconstruct full potential vector (insert pinned node voltage=0)
    f_full = np.zeros(N)
    f_full[mask] = f_reduced
    f_full[pinned_idx] = 0.0  # pinned node voltage

    # --- Step 3: The resistance is the voltage difference ---
    resistance = f_full[i1] - f_full[i2]

    return resistance
```

[28]:
```
# --- EXAMPLE USAGE ---
st1 = time.time()
n = 7

# 1. Create the graph
st2 = time.time()
my_large_graph = weighted_tails_d(n)
en1 = time.time()
```

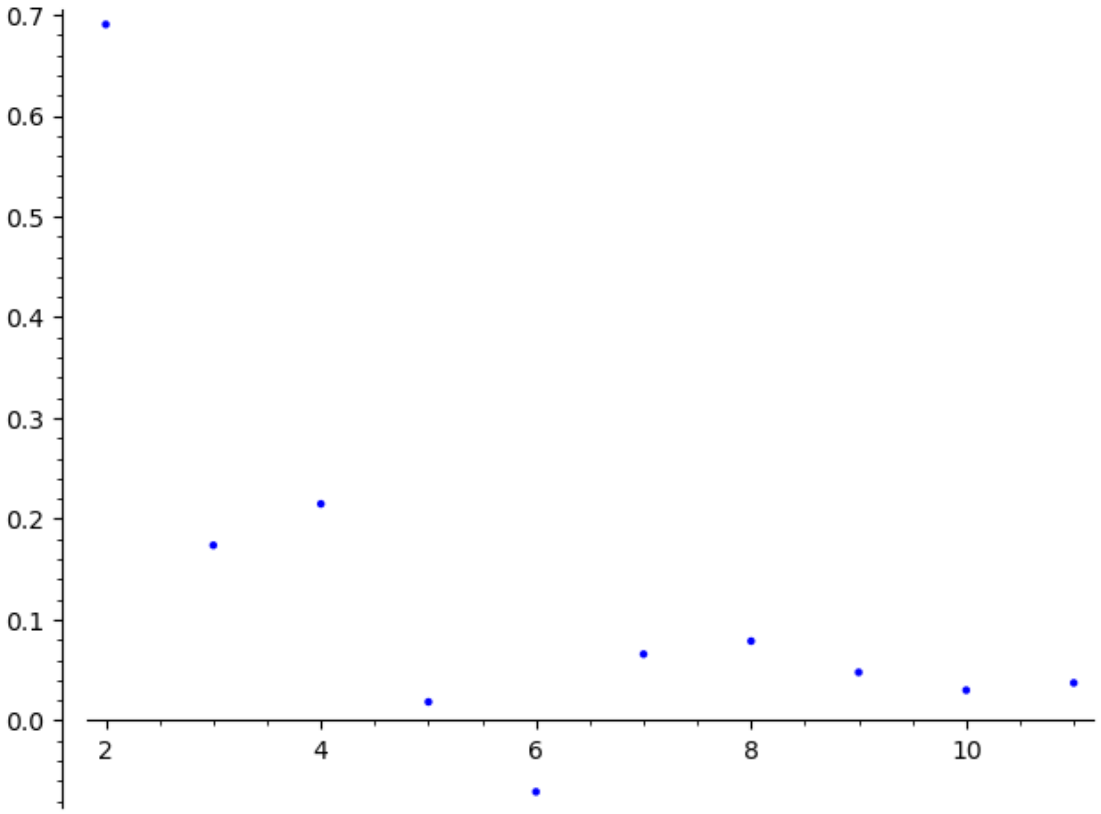
```
print(f'Time to make the graph: {en1 - st1}')
# 2. Choose two vertices to compute the resistance between.
v_a = 'b0'
v_b = 'b1'

# 3. Compute the single resistance value
R_ab = compute_single_resistance(my_large_graph, v_a, v_b)

print(f"\ncomputed resistance for H_{n}:")
print(f"R({v_a}, {v_b}) = {R_ab:.6f}")
en2 = time.time()
print(f"full time: {en2 - st1}")
print(f"System Solve time: {(en2 - st1) - (en1 - st1)}")
```

```
Time to make the graph: 0.38072633743286133

computed resistance for H_7:
R(b0, b1) = 1.000000
full time: 0.5402765274047852
System Solve time: 0.15955018997192383
```

[47]:
```
b_res = []
v_a = 'b0'
v_b = 'b1'
v_c = 'ext_1'
v_d = 'ext_2'

for n in range(2,12):
    g = weighted_tails_r(n)
    R_ab = compute_single_resistance(g, v_a, v_b)
    b_res.append(1 - R_ab)
```

[48]:
```
list_plot([(i+2, b_res[i]) for i in range(len(b_res))])
```

[48]:



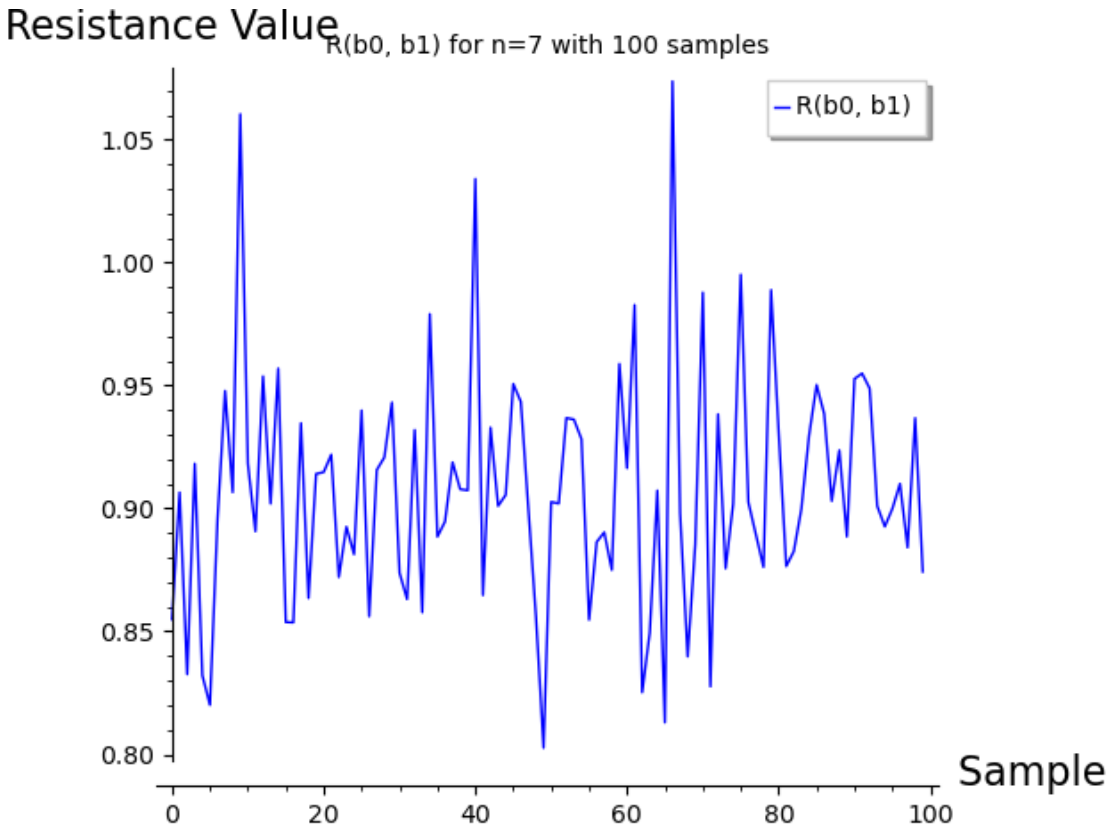
2 Variance of R_n (With Bootstrapping)

[85]:
```
# define number of initial samples
n_samples = 100 # good for up to n=7?

# define the largest graph level to sample from
n = 7

# create the graph outside the loop, initialize resistance list
g = H_tails(n); lambda_n = ll_vals[n-1]

res_list = []

# choose boundary points
v_a = 'b0'
v_b = 'b1'

for i in range(n_samples):
    # weight the graph
```

```
    # 2. Assign a random population to each vertex (representing a cell in the␣
↪pre-amalgamated SG).
    #    We add 1 to the Poisson result to ensure the population is always >= 1.
    cell_populations = {v: pois(lambda_n) + 1 for v in g.vertices()}

    d1 = [v for v in g.vertices() if g.degree(v) == 1]

    # 3. Iterate through edges and assign weights based on endpoint populations.
    for u, v, _ in g.edges():
        # Look up the pre-assigned populations
        pop_u = cell_populations[u]
        pop_v = cell_populations[v]

        # Calculate resistance
        r_uv = 1 / (pop_u * pop_v)

        if u in d1 or v in d1:
            r_uv /= 2

        g.set_edge_label(u, v, 1/r_uv)

    # compute the single resistance value, append to list
    R_ab = compute_single_resistance(g, v_a, v_b)

    res_list.append(R_ab)
```

[86]:
```
list_plot(res_list, plotjoined=True,
          title=f"R({v_a}, {v_b}) for n={n} with {n_samples} samples",
          legend_label=f"R({v_a}, {v_b})",
          axes_labels=["Sample Index", "Resistance Value"])
```

[86]:



[87]:
```
res_arr = np.array(res_list)
# use scipy.stats.bootstrap to compute the confidence interval for the variance␣
  ↪of the resistance at level n
from scipy.stats import bootstrap
boot = bootstrap((res_arr,), np.var, confidence_level=0.95)
ci = boot.confidence_interval
print(f"95% CI for variance of R({v_a}, {v_b}) at n={n}: ({ci[0]}, {ci[1]})")
```

```
95% CI for variance of R(b0, b1) at n=7: (0.0016394468068399084,
0.0034461216038283514)
```

[84]:
```
len(boot.bootstrap_distribution)
```

[84]:
```
9999
```