Supplemental Materials: Counting the Eigenvalues of the Laplace Operator on Some Convex Domains

Here we include relevant snippets of code for the Eigenvalue Counting Function (ECF) computation discussed in the main article, as well as further detail concerning some of the algorithms discussed.

To begin, we formalize the algorithms used to fund the zeros of the Bessel functions and their derivatives. As discussed, it relies on the local root-finding method fsolve, which relies on giving initial conditions. To find suitable initial conditions, we travel along the x-axis in defined steps until we find k such that $J_\nu(k) \cdot J_\nu(k + \text{Step}) < 0$, which implies $J_\nu(x)$ crossed the x-axis in that step and there exists a zero in that interval.

Algorithm S1 Finding the Dirichlet Zeros of the Bessel function

Input Order ν, Upper boundary kMax
Step 1: init k = ν
Step 2: init zeros as an empty list
Step 3: init $Step = 0.5$
Step 4:
While k < kMax

- Find $y_1 = J_\nu(k)$
- Find $y_2 = J_\nu(k + Step)$
- **If** $y_1 \cdot y_2 < 0$
    - Using fsolve to find $J_\nu(z) == 0$
    - **If** z < kMax and z > ν
        - add z to zeros
        - update k = k + $Step$
- **Else** update k = k + $Step$

Output List of all Dirichlet zeros of the ν's order Bessel function in (ν,kMax)

Algorithm S2 Finding the Neumann Zeros of the Bessel function

Input Order ν, Upper boundary kMax
Step 1: init k = ν
Step 2: init zeros as an empty list
Step 3: init $Step = 0.5$
Step 4:
While k < kMax

- Find $y_1' = J_\nu'(k)$
- Find $y_2' = J_\nu'(k + Step)$
- **If** $y_1' \cdot y_2' \leq 0$
  - Using fsolve to find $J_\nu'(z) == 0$
  - **If** z < kMax and z > ν
    - add z to zeros
    - update k = k + $Step$
- **Else** update k = k + $Step$

Output List of all Neumann zeros of the ν's order Bessel function in (ν,kMax)

What follows are screenshots of the functions used to compute ECFs. The Python matplotlib, numpy, scipy, and math libraries are all used in this code.

Square Code:

```
def largestSquare(num):
    rooty = math.sqrt(num)
    rooty_flr = math.floor(rooty)
    square = int(rooty_flr ** 2)
    return square
```

Figure S1: This finds the largest perfect square lower than a given input

```
def squareDirichletEigenvalues(kMax, s):
    biggestSquare = largestSquare(kMax**2)
    eigenvalues = {0:0}
    for m in range (1, int(math.sqrt(biggestSquare))):
        for n in range (1, int(math.sqrt(biggestSquare))):
            candidate = math.sqrt((m**2 + n**2)*((PI**2)/s**2))

            if candidate > kMax:
                break
            if candidate not in eigenvalues.keys():
                eigenvalues[candidate] = 1
            else:
                eigenvalues[candidate] += 1

    eigenvalues = sorted(eigenvalues.items())
    return eigenvalues
```

```
def squareNeumannEigenvalues(kMax, s):
    biggestSquare = largestSquare(kMax**2)
    eigenvalues = {}
    for m in range (0, int(math.sqrt(biggestSquare))):
        for n in range (0, int(math.sqrt(biggestSquare))):
            candidate = math.sqrt((m**2 + n**2)*((PI**2)/s**2))

            if candidate > kMax:
                break
            if candidate not in eigenvalues.keys():
                eigenvalues[candidate] = 1
            else:
                eigenvalues[candidate] += 1

    eigenvalues = sorted(eigenvalues.items())
    return eigenvalues
```

Figure S2: These use algorithm 1 to find the ECF of a square with sidelength s

```
def plotting(all_eigen_lst, all_eigen_lst_2, kMax, figstart = 0):

    length = len(all_eigen_lst)
    x = np.array([all_eigen_lst[i][0] for i in range(length)])
    y = np.array([all_eigen_lst[i][1] for i in range(length)])
    y_sum = np.array([y[:i].sum() for i in range(1,length+1)])
    #print(y_sum)

    length_2 = len(all_eigen_lst_2)
    x_2 = np.array([all_eigen_lst_2[i][0] for i in range(length_2)])
    y_2 = np.array([all_eigen_lst_2[i][1] for i in range(length_2)])
    y_2_sum = np.array([y_2[:i].sum() for i in range(1,length_2+1)])
    #print(x_2)

    x_ = np.linspace(0, kMax, 1000)
    y_ = x_ ** 2 / 4

    plt.step(x, y_sum, "blue", where = "post", linestyle='solid',label = "Dirichlet")
    plt.step(x_2, y_2_sum, "red", where = "post", linestyle='solid',label = "Neumann")

    plt.plot(x_, y_, color = "black",label = "$k^2/4$")
    plt.xlabel(r"$k$")
    plt.ylabel(r'$\mathcal{N} (k)$')
    plt.xlim(figstart, kMax)
    # plt.ylim(0, )
    # plt.title("Eigenvalue Counting Functions and Weyl Polynomial for Square of side length $\pi$")
    plt.legend()
    # plt.savefig(f"square_plot_test_{strSTEP}.png")
    plt.show()
```

Figure S3: The plotting() function plots the Neumann and Dirichlet ECFs along with the leading Weyl term for a square with sidelength $\sqrt{\pi}$.

Disk Code:

```
PI = math.pi
STEP = 0.5
strSTEP = str(STEP)
DERIVE_ORDER = 1
```

```
def bessel(x, nu):
    return scipy.special.jv(nu, x)
```

```
def besselDerivative(x, nu):
    return scipy.special.jvp(nu, x, DERIVE_ORDER)
```

Figure S4: Beyond initializing our parameters, scipy fsolve only works on the first argument of a function, so we needed to make new functions to solve for roots in the spatial variable x.

```
def besselZeros(lower, upper, nu):

    k = nu
    zeros = []
    while k < upper:

        y1 = scipy.special.jv(nu, k)
        y2 = scipy.special.jv(nu, k + STEP)

        if y1 * y2 < 0 :
            zero = scipy.optimize.fsolve(bessel, k, args = (nu))
            if zero < upper and zero > nu:
                zeros.append(zero)
            k = k + STEP
        else:
            k = k + STEP
    return zeros
```

Figure S5: The function besselZeros() uses algorithm S1 to find the roots of the rank ν Bessel function on the interval [lower, upper]

```
def besselDerivativeZeros(lower, upper, nu):

    k = nu
    zeros = []
    while k < upper:

        y1 = scipy.special.jvp(nu, k, DERIVE_ORDER)
        y2 = scipy.special.jvp(nu, k + STEP, DERIVE_ORDER)

        if y1 * y2 <= 0 :
            zero = scipy.optimize.fsolve(besselDerivative, k, args = (nu))
            if zero < upper and zero > nu:
                zeros.append(zero)
            k = k + STEP
        else:
            k = k + STEP
    return zeros
```

Figure S6: The function besselDerivativeZeros() uses algorithm S2 to find the roots of the rank ν Bessel function derivative on the interval [lower, upper]

```
def besselEigenvalues(kMax):

    nMax = math.ceil(kMax)
    eigenvalues = {}
    for nu in range(0, nMax):
        Arr = besselZeros(0, kMax, nu)
        #if len(Arr) != 0 :
        eigenvalues[nu] = Arr
    return eigenvalues
```

```
def besselDerivativeEigenvalues(kMax):

    nMax = math.ceil(kMax)
    eigenvalues = {}
    for nu in range(0, nMax):
        Arr = besselDerivativeZeros(0, kMax, nu)
        #if len(Arr) != 0 :
        eigenvalues[nu] = Arr
    return eigenvalues
```

Figure S7: Here we use algorithm 2 to collect the Disk eigenvalues

```
def besselEigenPairs(all_eigens,kMax):
    all_eigen_lst = [(0,0)]
    for j in range(len(all_eigens[0])):
        all_eigen_lst.append((all_eigens[0][j][0],1))
    for i in range(1,int(kMax)):
        for j in range(len(all_eigens[i])):
            all_eigen_lst.append((all_eigens[i][j][0],2))
    all_eigen_lst = sorted(all_eigen_lst, key = lambda x: x[0])
    return all_eigen_lst
```

```
def besselDerivativeEigenPairs(all_eigens,kMax):
    all_eigen_lst = [(0,1)]
    for j in range(len(all_eigens[0])):
        all_eigen_lst.append((all_eigens[0][j][0],1))
    for i in range(1,int(kMax)):
        for j in range(len(all_eigens[i])):
            all_eigen_lst.append((all_eigens[i][j][0],2))
    all_eigen_lst = sorted(all_eigen_lst, key = lambda x: x[0])
    return all_eigen_lst
```

Figure S8: Here we initialize the multiplicity of the disk eigenvalues based on rank

```
def regressionTest(all_eigen_lst,all_eigen_lst_2):

    test_list_D = np.array([])
    test_list_N = np.array([])
    for n in range(9):
        test_list_D = np.append(test_list_D,scipy.special.jn_zeros(n, 4))
        test_list_N = np.append(test_list_N,scipy.special.jnp_zeros(n, 4))
    test_list_D = list(np.sort(test_list_D))[:15]
    test_list_N = list(np.sort(test_list_N))[:15]


    x_D = [all_eigen_lst[i+1][0] for i in range(len(test_list_D))]
    x_N = [all_eigen_lst_2[i+1][0] for i in range(len(test_list_N))]

    for i in range(len(test_list_D)):
        if abs(test_list_D[i] - x_D[i]) > 1e-6:
            print(f"Dirichlet test failed,\n test value:{test_list_D[i]}, calculate value:{x_D[i]}")
            return
        if abs(test_list_N[i] - x_N[i]) > 1e-6:
            print(f"Neumann test failed,\n test value:{test_list_N[i]}, calculate value:{x_N[i]}")
            return
    print("Test passed")
    return
```

Figure S9: This function is a regression test we used to compare our lower rank results with the scipy built-in Bessel zeros functions. This function can be used to test any future Bessel function zero function that is developed.

```
def plotting(all_eigen_lst, all_eigen_lst_2, kMax, figstart = 0):


    length = len(all_eigen_lst)
    x = np.array([all_eigen_lst[i][0] for i in range(length)])
    y = np.array([all_eigen_lst[i][1] for i in range(length)])
    #print(x)
    #print(y)
    y_sum = np.array([y[:i].sum() for i in range(1,length+1)])
    #print(y_sum)

    length_2 = len(all_eigen_lst_2)
    x_2 = np.array([all_eigen_lst_2[i][0] for i in range(length_2)])
    y_2 = np.array([all_eigen_lst_2[i][1] for i in range(length_2)])
    #print(x)
    #print(y_2)
    y_2_sum = np.array([y_2[:i].sum() for i in range(1,length_2+1)])
    #print(y_2_sum)

    x_ = np.linspace(0, kMax, 1000)
    y_ = x_ ** 2 / 4

    plt.step(x, y_sum, "blue", where = "post", linestyle='solid',label = "Dirichlet")
    plt.step(x_2, y_2_sum, "red", where = "post", linestyle='solid',label = "Neumann")

    plt.plot(x_, y_, color = "black",label = "$k^2/4$")
    plt.xlabel(r"k")
    plt.ylabel(r'$\mathcal{N} (k)$')
    plt.xlim(figstart, kMax)
    #plt.ylim(0, )
    # plt.title(f"Eigenvalue Counting Functions and Leading Weyl Term for the Unit Disk")
    plt.legend()
    plt.savefig(f"plot_test_{int(kMax)}.png")
    plt.show()
```

Figure S10: This plotting function plots the computed ECFs and leading Weyl term for the Unit Disk

Cube Code:

```
def cubeDirichletEigenvalues(kMax, s):
    biggestSquare = largestSquare(kMax**2)
    eigenvalues = {0:0}
    for m in range (1, int(math.sqrt(biggestSquare))):
        for n in range (1, int(math.sqrt(biggestSquare))):
            for p in range (1, int(math.sqrt(biggestSquare))):
                candidate = math.sqrt((m**2 + n**2 + p**2)*((PI**2)/s**2))

                if candidate > kMax:
                    break
                if candidate not in eigenvalues.keys():
                    eigenvalues[candidate] = 1
                else:
                    eigenvalues[candidate] += 1

    eigenvalues = sorted(eigenvalues.items())
    return eigenvalues
```

Figure S11: This function, cubeDirichletEigenvalues() uses a variation on 1 to find the Dirichlet eigenvalues of a cube with sidelength s

```
def cubeNeumannEigenvalues(kMax, s):
    biggestSquare = largestSquare(kMax**2)
    eigenvalues = {}
    for m in range (0, int(math.sqrt(biggestSquare))):
        for n in range (0, int(math.sqrt(biggestSquare))):
            for p in range (0, int(math.sqrt(biggestSquare))):
                candidate = math.sqrt((m**2 + n**2 + p**2)*((PI**2)/s**2))

                if candidate > kMax:
                    break
                if candidate not in eigenvalues.keys():
                    eigenvalues[candidate] = 1
                else:
                    eigenvalues[candidate] += 1

    eigenvalues = sorted(eigenvalues.items())
    return eigenvalues
```

Figure S12: This function, cubeNeumannEigenvalues() uses a variation on 1 to find the Neumann eigenvalues of a cube with sidelength s

Ball Code:

```
def besselSpherical(x, nu):
    return scipy.special.spherical_jn(nu, x)
```

```
def besselSphericalZeros(lower, upper, nu):

    k = nu
    zeros = []
    while k < upper:

        y1 = scipy.special.spherical_jn(nu, k)
        y2 = scipy.special.spherical_jn(nu, k + STEP)

        if y1 * y2 <= 0 :
            zero = scipy.optimize.fsolve(besselSpherical, k, args = (nu))
            if zero < upper and zero > nu:
                zeros.append(zero)
            k = k + STEP
        else:
            k = k + STEP
    return zeros
```

Figure S13: Here we used a variation on the function in S5 for finding the roots of Spherical Bessel functions

```
def besselSphericalEigenvalues(kMax):

    nMax = math.ceil(kMax)
    eigenvalues = {}
    # Now we have a problem here?
    for nu in range(0, nMax):
        Arr = besselSphericalZeros(0, kMax, nu)
        #if len(Arr) != 0 :
        eigenvalues[nu] = Arr
    return eigenvalues
```

```
def besselSphericalEigenPairs(all_eigens,kMax):

    all_eigen_lst = [(0,0)]

    for i in range(0,int(kMax)):
        for j in range(len(all_eigens[i])):
            all_eigen_lst.append((all_eigens[i][j][0], 2 * i + 1))

    all_eigen_lst = sorted(all_eigen_lst, key = lambda x: x[0])

    return all_eigen_lst
```

Figure S14: These two functions use algorithm 3 to compute the Dirichlet ECF of the Sphere.

```
def besselSphericalDerivative(x, nu):
    return scipy.special.spherical_jn(nu, x, True)
```

```
def besselSphericalDerivativeZeros(lower, upper, nu):

    k = nu
    zeros = []
    while k < upper:

        y1 = scipy.special.spherical_jn(nu, k, True)
        y2 = scipy.special.spherical_jn(nu, k + STEP, True)

        if y1 * y2 <= 0 :
            zero = scipy.optimize.fsolve(besselSphericalDerivative, k, args = (nu))
            if zero < upper and zero > nu:
                zeros.append(zero)
            k = k + STEP
        else:
            k = k + STEP
    return zeros
```

Figure S15: Here we used a variation on the function in S6 for finding the roots of Spherical Bessel functions

```
def besselSphericalDerivativeEigenvalues(kMax):

    nMax = math.ceil(kMax)
    eigenvalues = {}
    # Now we have a problem here?
    for nu in range(0, nMax):
        Arr = besselSphericalDerivativeZeros(0, kMax, nu)
        #if len(Arr) != 0 :
        eigenvalues[nu] = Arr
    return eigenvalues
```

```
def besselSphericalDerivativeEigenPairs(all_eigens,kMax):

    all_eigen_lst = [(0,1)]

    for i in range(0,int(kMax)):
        for j in range(len(all_eigens[i])):
            all_eigen_lst.append((all_eigens[i][j][0], 2 * i + 1))

    all_eigen_lst = sorted(all_eigen_lst, key = lambda x: x[0])

    return all_eigen_lst
```

Figure S16: These two functions use a slight variation on algorithm 3 to compute the Neumann ECF of the Sphere.

```
def plotting_3D(all_eigen_lst, all_eigen_lst_2, kMax, figstart = 0):


    length = len(all_eigen_lst)
    x = np.array([all_eigen_lst[i][0] for i in range(length)])
    y = np.array([all_eigen_lst[i][1] for i in range(length)])
    #print(x)
    #print(y)
    y_sum = np.array([y[:i].sum() for i in range(1,length+1)])
    #print(y_sum)

    length_2 = len(all_eigen_lst_2)
    x_2 = np.array([all_eigen_lst_2[i][0] for i in range(length_2)])
    y_2 = np.array([all_eigen_lst_2[i][1] for i in range(length_2)])
    #print(x)
    #print(y_2)
    y_2_sum = np.array([y_2[:i].sum() for i in range(1,length_2+1)])
    #print(y_2_sum)

    x_ = np.linspace(0, kMax, 1000)
    y_ = 2 * (x_ ** 3) / (9 * PI)

    plt.step(x, y_sum, "blue", where = "post", linestyle='solid',label = "Dirichlet")
    plt.step(x_2, y_2_sum, "red", where = "post", linestyle='solid',label = "Neumann")

    plt.plot(x_, y_, color = "black",label = "$ 2k^3 / 9\pi $")
    plt.xlabel(r"$k$")
    plt.ylabel(r'$\mathcal{N} (k)$')
    plt.xlim(figstart, kMax)
    plt.ylim(0, )
    plt.legend()
    #plt.title("Eigenvalue Counting Functions and Weyl Polynomial for Spherical")
    plt.savefig(f"plot_test_{strSTEP}.png")
    plt.show()
```

Figure S17: This plotting() function plots the ECFs and leading Weyl term for the Unit Sphere.

Cylinder:

```
def DirichletIntervalEigenvalues(kMax,a):
    nMax = math.ceil(kMax * a / PI)
    n_list = [i * PI/a for i in range(1,nMax)]
    # print(n_list)
    return n_list
```

```
def NeumannIntervalEigenvalues(kMax,a):
    nMax = math.ceil(kMax * a / PI)
    n_list = [i * PI/a for i in range(0,nMax)]
    # print(n_list)
    return n_list
```

Figure S18: These functions compute the eigenvalues of the interval [0,a].

```
def besselEigenvalues(kMax, r):

    nMax = math.ceil(r * kMax)
    eigenvalues = {}
    for nu in range(0, nMax):
        Arr = besselZeros(0, r * kMax, nu)
        Arr_1 = [Arr[i]/r for i in range(len(Arr))]
        eigenvalues[nu] = Arr_1
    return eigenvalues
```

```
def besselDerivativeEigenvalues(kMax, r):

    nMax = math.ceil(r * kMax)
    eigenvalues = {}
    for nu in range(0, nMax):
        Arr = besselDerivativeZeros(0, r * kMax, nu)
        Arr_1 = [Arr[i]/r for i in range(len(Arr))]
        eigenvalues[nu] = Arr_1
    return eigenvalues
```

Figure S19: These functions compute the Eigenvalues of the Disk with radius r, using the functions in figures S5-S5.

```
def besselEigenPairs(all_eigens,kMax):
    # print(all_eigens)
    all_eigen_lst = []
    for i in range(int(kMax)):
      if i in all_eigens.keys():
        for j in range(len(all_eigens[i])):
          if (i == 0):
              all_eigen_lst.append((all_eigens[i][j][0],1))
          else:
              all_eigen_lst.append((all_eigens[i][j][0],2))
    all_eigen_lst = sorted(all_eigen_lst, key = lambda x: x[0])
    # print(all_eigen_lst)
    return all_eigen_lst
```

```
def besselDerivativeEigenPairs(all_eigens,kMax):
    all_eigen_lst = [(0,1)]
    for i in range(int(kMax)):
      if i in all_eigens.keys():
        for j in range(len(all_eigens[i])):
          if (i == 0):
              all_eigen_lst.append((all_eigens[i][j][0],1))
          else:
              all_eigen_lst.append((all_eigens[i][j][0],2))
    all_eigen_lst = sorted(all_eigen_lst, key = lambda x: x[0])
    # print(all_eigen_lst)
    return all_eigen_lst
```

Figure S20: These functions initialize the multiplicity of the Bessel zeros based on the Bessel function rank.

```
def fullDirichletEigenvalues(kMax,all_eigens_lst,n_list):
    eigenvalues = {0:0}
    for i in range(len(all_eigens_lst)):
        for n in n_list:
            candidate = math.sqrt((all_eigens_lst[i][0])**2 + n**2)

            if candidate > kMax:
                break
            if candidate not in eigenvalues.keys():
                eigenvalues[candidate] =  all_eigens_lst[i][1]
            else:
                eigenvalues[candidate] += all_eigens_lst[i][1]

    eigenvalues = sorted(eigenvalues.items())
    # print(eigenvalues)
    return eigenvalues
```

```
def fullNeumannEigenvalues(kMax,all_eigens_lst,n_list):
    eigenvalues = {}
    for i in range(len(all_eigens_lst)):
        for n in n_list:
            candidate = math.sqrt((all_eigens_lst[i][0])**2 + n**2)

            if candidate > kMax:
                break
            if candidate not in eigenvalues.keys():
                eigenvalues[candidate] = all_eigens_lst[i][1]
            else:
                eigenvalues[candidate] += all_eigens_lst[i][1]

    eigenvalues = sorted(eigenvalues.items())
    # print(eigenvalues)
    return eigenvalues
```

Figure S21: These functions combine the interval and disk eigenvalues using algorithm 4 to generate the ECFs of the Cylinder.

```
def plotting(all_eigen_lst, all_eigen_lst_2, kMax, figstart = 0):

    length = len(all_eigen_lst)
    x = np.array([all_eigen_lst[i][0] for i in range(length)])
    y = np.array([all_eigen_lst[i][1] for i in range(length)])
    #print(x)
    #print(y)
    y_sum = np.array([y[:i].sum() for i in range(1,length+1)])
    #print(y_sum)

    length_2 = len(all_eigen_lst_2)
    x_2 = np.array([all_eigen_lst_2[i][0] for i in range(length_2)])
    y_2 = np.array([all_eigen_lst_2[i][1] for i in range(length_2)])
    #print(x)
    #print(y_2)
    y_2_sum = np.array([y_2[:i].sum() for i in range(1,length_2+1)])
    #print(y_2_sum)

    x_ = np.linspace(0, kMax, 1000)
    y_ = (x_ ** 3)/(6 * PI**2)

    plt.step(x, y_sum, "blue", where = "post", linestyle='solid',label = "Dirichlet")
    plt.step(x_2, y_2_sum, "red", where = "post", linestyle='solid',label = "Neumann")

    plt.plot(x_, y_, color = "black",label = "Weyl Polynomial")
    plt.xlabel(r"$\lambda$")
    plt.ylabel(r'$\mathcal{N} (\lambda)$')
    plt.xlim(figstart, kMax)
    #plt.ylim(0, )
    plt.title(f"Eigenvalue Counting Functions and Weyl Polynomial for Cylinder")
    plt.legend()
    plt.savefig(f"plot_test_{int(kMax)}.png")
    plt.show()
```

Figure S22: This plotting() function plots the ECFs and leading Weyl term for a cylinder with unit volume.

```
kMax = float(input("Input your kMax here:"))
r = 0.40071301
a = 1/(PI * r ** 2)

all_j = besselEigenvalues(kMax, r)
all_j_lst = besselEigenPairs(all_j, kMax)
all_k_lst = DirichletIntervalEigenvalues(kMax,a)
all_eigen_lst = fullDirichletEigenvalues(kMax,all_j_lst,all_k_lst)

all_j_N = besselDerivativeEigenvalues(kMax, r)
all_j_N_lst = besselDerivativeEigenPairs(all_j_N, kMax)
all_k_N_lst = NeumannIntervalEigenvalues(kMax,a)
all_eigen_lst_N = fullNeumannEigenvalues(kMax,all_j_N_lst,all_k_N_lst)

print(f"Dirichlet:The smallest eigen: {all_eigen_lst[1]}, the largest eigen: {all_eigen_lst[-1]}")
print(f"Dirichlet:There are {sum(x[1] for x in all_eigen_lst[1:])} eigens (with multiplicity) in total.")

print(f"Neumann:The smallest eigen: {all_eigen_lst_N[0]}, the largest eigen: {all_eigen_lst_N[-1]}")
print(f"Neumann:There are {sum(x[1] for x in all_eigen_lst_N)} eigens (with multiplicity) in total.")
print(f"lambda ^ 3 / (6 PI ^ 2) is { (math.ceil(kMax) ** 3)/(6 * PI**2) } ")
```

Figure S23: Here we compute the ECFs for the cylinder C_1 using the functions in figures S18-S22

```
kMax = float(input("Input your kMax here:"))
r = 0.713154
a = 1/(PI * r ** 2)
print(a)

all_j = besselEigenvalues(kMax, r)
all_j_lst = besselEigenPairs(all_j, kMax)
all_k_lst = DirichletIntervalEigenvalues(kMax,a)
all_eigen_lst = fullDirichletEigenvalues(kMax,all_j_lst,all_k_lst)

all_j_N = besselDerivativeEigenvalues(kMax, r)
all_j_N_lst = besselDerivativeEigenPairs(all_j_N, kMax)
all_k_N_lst = NeumannIntervalEigenvalues(kMax,a)
all_eigen_lst_N = fullNeumannEigenvalues(kMax,all_j_N_lst,all_k_N_lst)

print(f"Dirichlet:The smallest eigen: {all_eigen_lst[1]}, the largest eigen: {all_eigen_lst[-1]}")
print(f"Dirichlet:There are {sum(x[1] for x in all_eigen_lst[1:])} eigens (with multiplicity) in total.")

print(f"Neumann:The smallest eigen: {all_eigen_lst_N[0]}, the largest eigen: {all_eigen_lst_N[-1]}")
print(f"Neumann:There are {sum(x[1] for x in all_eigen_lst_N)} eigens (with multiplicity) in total.")
print(f"lambda ^ 3 / (6 PI ^ 2) is { (math.ceil(kMax) ** 3)/(6 * PI**2) } ")
```

Figure S24: Here we compute the ECFs for the cylinder C_2 using the functions in figures S18-S22