Biochemical Engineering Lab
Prof. Dr. P. Arosio
D CHAB

Biochemical and Polymer Reaction Engineering: Exercise 3

by Eduard Meier

This Jupyter Notebooks contains a presentation of the applied equations, all code (main script and various functions) used to solve the exercise as well as the resulting plots. It belongs to the course "Biochemical and Polymer Reaction Engineering", held by Prof. Arosio in the autumn semester 2022. References to equations inside the lecture script are indicated on the left side of the corresponding equations and refer to the equation number in the lecture script.

1 Equations

1.1 Part a)

The aggregation rate constants for diffusion-limited cluster aggregation (DLCA) are calculated according to

$$(5.19) \qquad \beta_{ij}^{DLCA} = \frac{2}{3}\frac{k_\mathrm{B}T}{\eta}\left(i^{1/d_\mathrm{f}} + j^{1/d_\mathrm{f}}\right)\left(i^{-1/d_\mathrm{f}} + j^{-1/d_\mathrm{f}}\right) \tag{1}$$

for a non-dimensional mass of $i, j = 1 - 100$.

1.2 Part b)

To describe the cluster formation up to a dimensionless mass of $i = 100$, the Smoluchowski equation

$$(5.8) \qquad \frac{\mathrm{d}N_k}{\mathrm{d}t} = \frac{1}{2}\sum_{i=1}^{k-1}\beta_{i,k-i}N_iN_{k-i} - N_k\sum_{i=1}^{\infty}\beta_{ik}N_i \tag{2}$$

is solved numerically, where N_k corresponds to the number concentration of aggregates of size k.

The total number of clusters (of all sizes) is given by summation (from 1 to ∞ or 100 in our case) at each point in time:

$$(5.9) \qquad N_{tot}(t) = \sum_{i=1}^{\infty} N_i(t) \tag{3}$$

1.3 Part c)

The aggregation rate constant for reaction limited cluster aggregation (RLCA) can be found from the respective DLCA rate constant using the Fuchs stability ratio:

$$(5.31) \qquad \beta_{ij}^{RLCA}(W) = \frac{\beta_{ij}^{DLCA}}{W}(ij)^{\lambda} \quad \text{with} \;\; \lambda = \frac{d_f - 1}{d_f} \tag{4}$$

The optimal Fuchs stability ratio, W_opt, is found via non-linear regression of the system of ODEs presented above, in which the RLCA aggregation rate constant, β_{ij}^{RLCA}, is used.

1.4 Part d)

The Fuchs stability ratio can be found via numerically integrating the following expression:

$$(5.26) \qquad W = 2a\int_{2a}^{\infty}\exp\left(\frac{V_T}{k_\mathrm{B}T}\right)\frac{\mathrm{d}r}{r^2} \tag{5}$$

The total potential, V_T, is given by the sum of an attractive term, V_A, and a repulsive term, V_R, given by

$$(\mathrm{Ex}) \qquad V_A = -\frac{A_H}{6}\left[\frac{2}{l^2-4} + \frac{2}{l^2} + \ln\left(1 - \frac{4}{l^2}\right)\right] \tag{6}$$

$$(\mathrm{Ex}) \qquad V_R = \frac{4\pi\epsilon_0\epsilon_r a\psi_0^2}{l}\ln\left(1 + \exp\left(-\kappa\alpha(l-2)\right)\right) \;. \tag{7}$$

Finally, the Debye-Hückel parameter, κ, is calculated from the ionic strength, I

$$(4.19 \text{ ff.}) \qquad I = \frac{1}{2}\sum_i c_i z_i^2 \tag{8}$$

$$(4.19) \qquad \kappa = 3.29\sqrt{I}\;[\mathrm{nm}^{-1}] \tag{9}$$

where c_i and z_i denote the concentration (in mol/L) and valency of ion species i, which are given in the assignment (valency for sodium chloride is one). In addition, we have $l = r/a$.

2. Plots and Discussion

Part a)

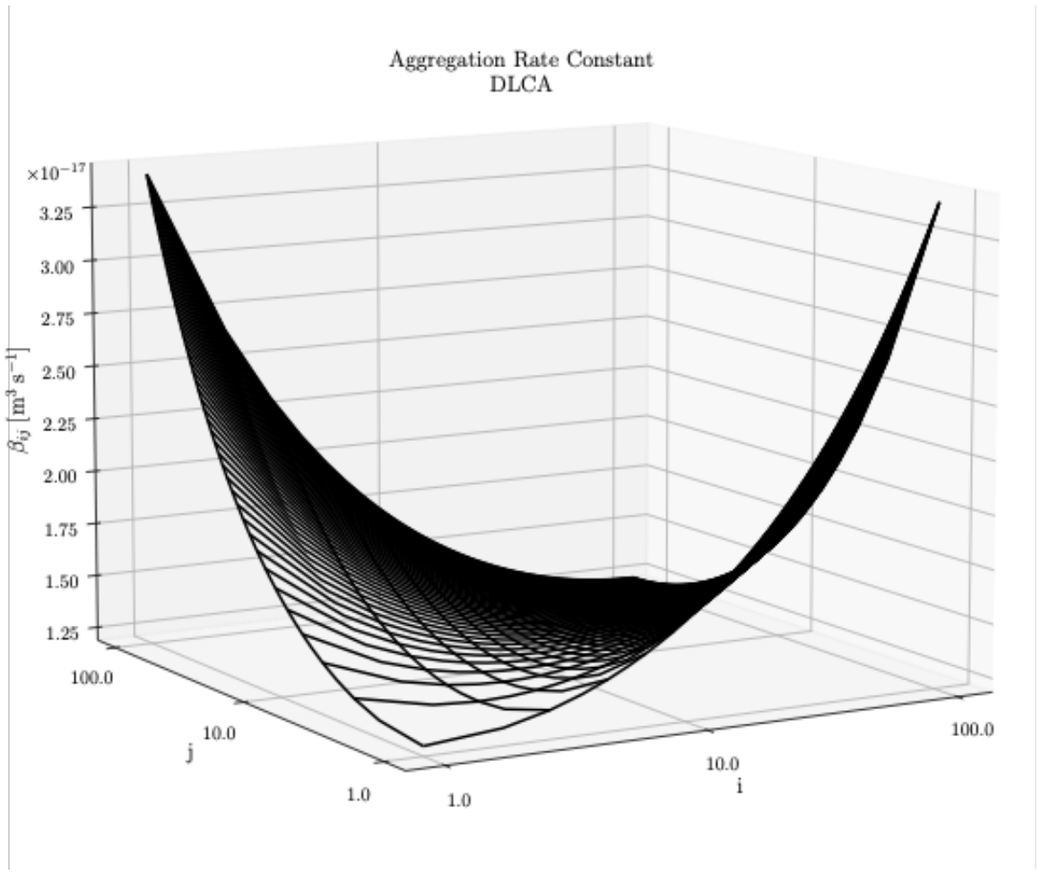


Figure 1: Aggregation rate constants for clusters of dimensionless masses between 1 and 100 in the diffusion-limited regime.

The visualization of the functional dependence of the aggregation rate constant in the diffusion-limited regime on the dimensionless masses, i, j, shows the expected behaviour, namely that high rates are expected for the aggregation of clusters of very different masses ($\sim$ sizes), while clusters of similar masses ($\sim$ sizes) significantly reduced aggregation rates.

While large clusters have a large surface area available for collision, their diffusion coefficients are relatively low. On the contrary, small clusters diffuse quickly (large diffusion coefficients), but their surface area is smaller and thus the probability of a collision is smaller. Combining two clusters from both extremes (very large and very small) therefore combines the high surface area with fast diffusion and in consequence leads to high aggregation rate constants.

Part b)

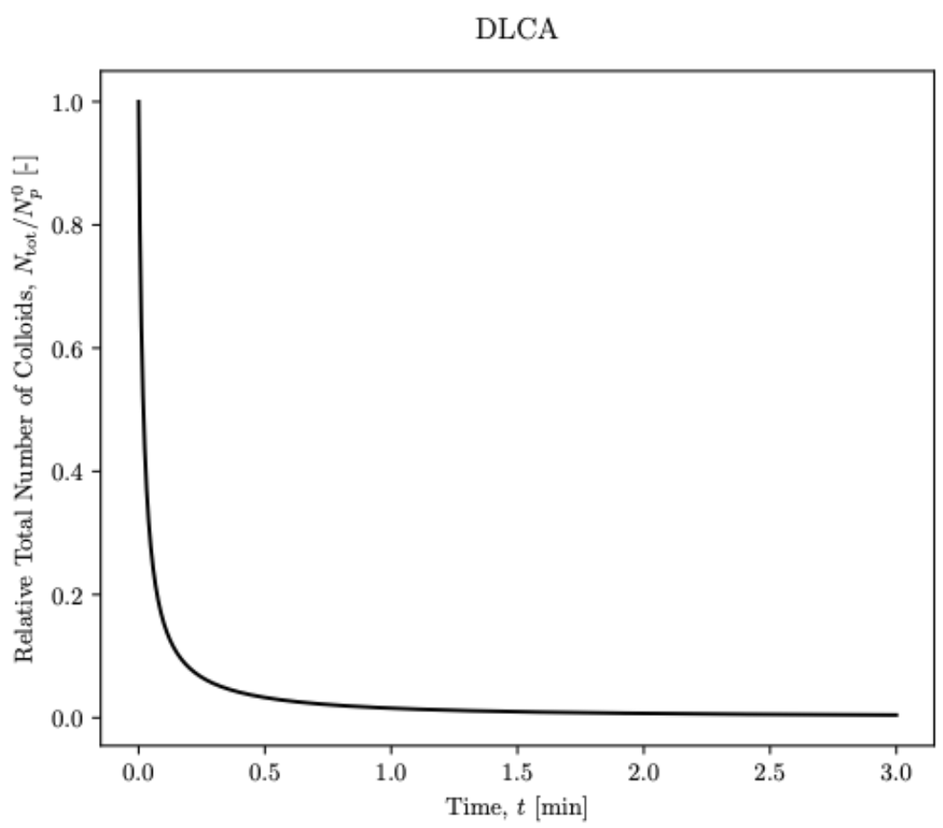


Figure 2: Total relative number of colloids versus time considering dimensionless masses between 1 and 100 in the diffusion-limited regime.

The above figure shows the evolution of the total number of particles (of all sizes) with time. It decreases rapidly and within the first few hours, most of particles have aggregated. For shorter timescales, one would observe a decreasing aggregation velocity with time, which can be attributed to the diminished diffusion coefficient of large clusters and thus despite their larger surface area, fewer collision events are observed.

Part c)

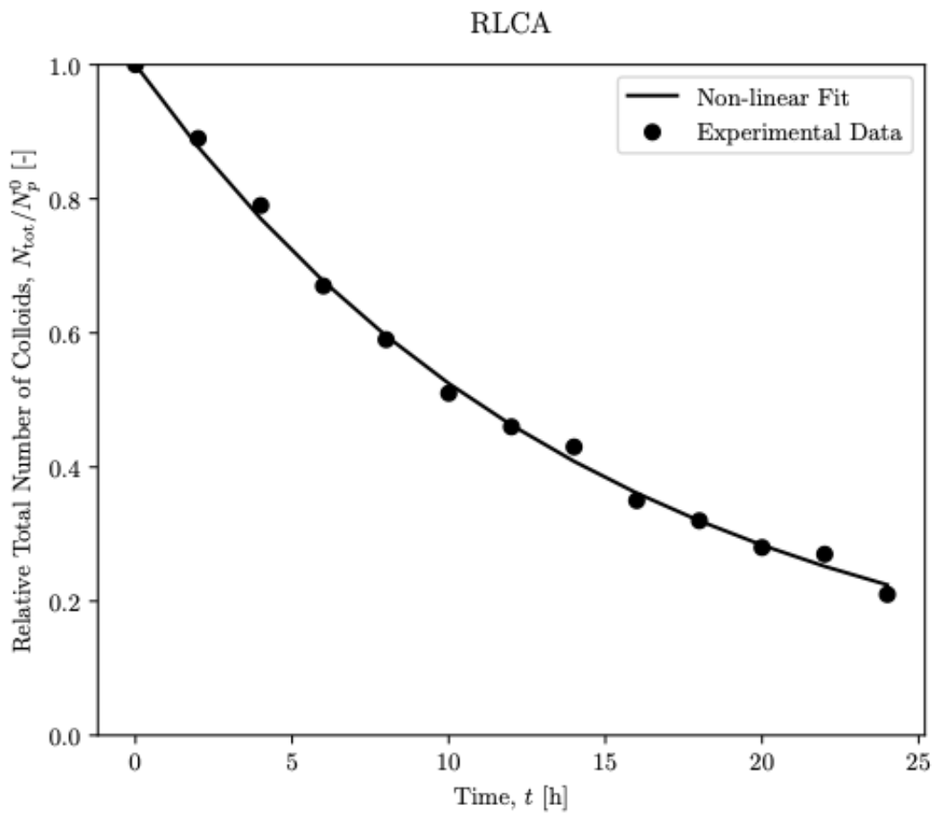


Figure 3: Total relative number of colloids versus time considering dimensionless masses between 1 and 100 in the reaction-limited regime.

A non-linear fit of the simulated relative colloid numbers to the experimentally observed values was conducted, in which the value of the Fuchs stability ratio was fitted. The optimal value was found to be $W_{\text{opt}} = 47'342$, resulting in a reasonable fit. Comparing the RLCA results with the previous simulation results for DLCA, we see that aggregation is significantly slower (more than 10 times) than in the DLCA regime.

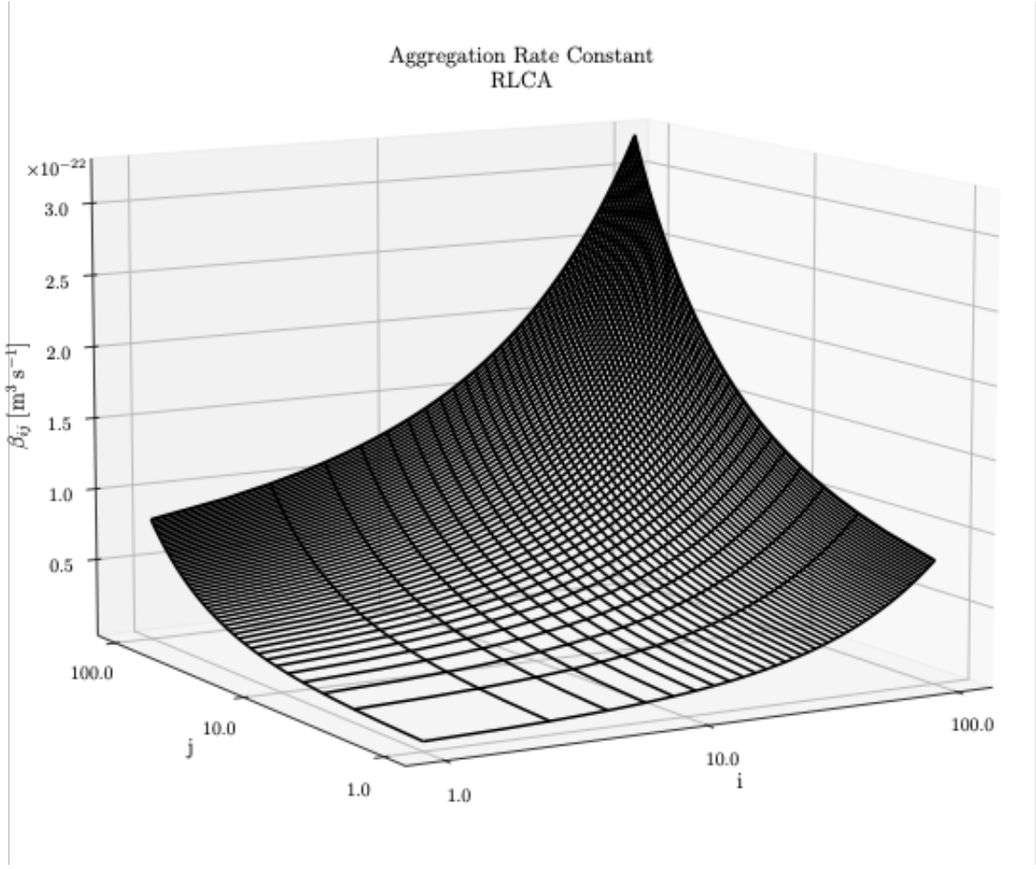


Figure 4: Aggregation rate constants for clusters of dimensionless masses between 1 and 100 in the reaction-limited regime.

The resulting RLCA aggregation rate constants were subsequently calculated with the optimal Fuchs stability ratio and the results are displayed in the figure above. The differences to Figure 1 can be explained as follows: in a reaction limited regime, the penalty larger particles face in terms of their diffusion coefficient does not impact the agglomeration rate. Without the penalty, it follows that the larger the particles are, which collide, the higher the rate constant due to their increased surface area. Consequently, the highest rate constant is observed in the back corner of the plot, which is given by the collision of the largest particles.

Part d)

Substituting r by l for the numerical integration, making use of $l = r/a$, a Fuchs stability ratio of $W = 130'040$ was calculated, which seems reasonable. However, the obtained Fuchs stability ratio is significantly higher than the experimentally fitted value, which might stem from inaccuracies of the analytical expressions for the (attractive and repulsive) potentials used in the integration. The higher value of the Fuchs stability ratio implies that the rate constants in the diffusion-limited regime are higher than in the reaction-limited regime.

Part e)

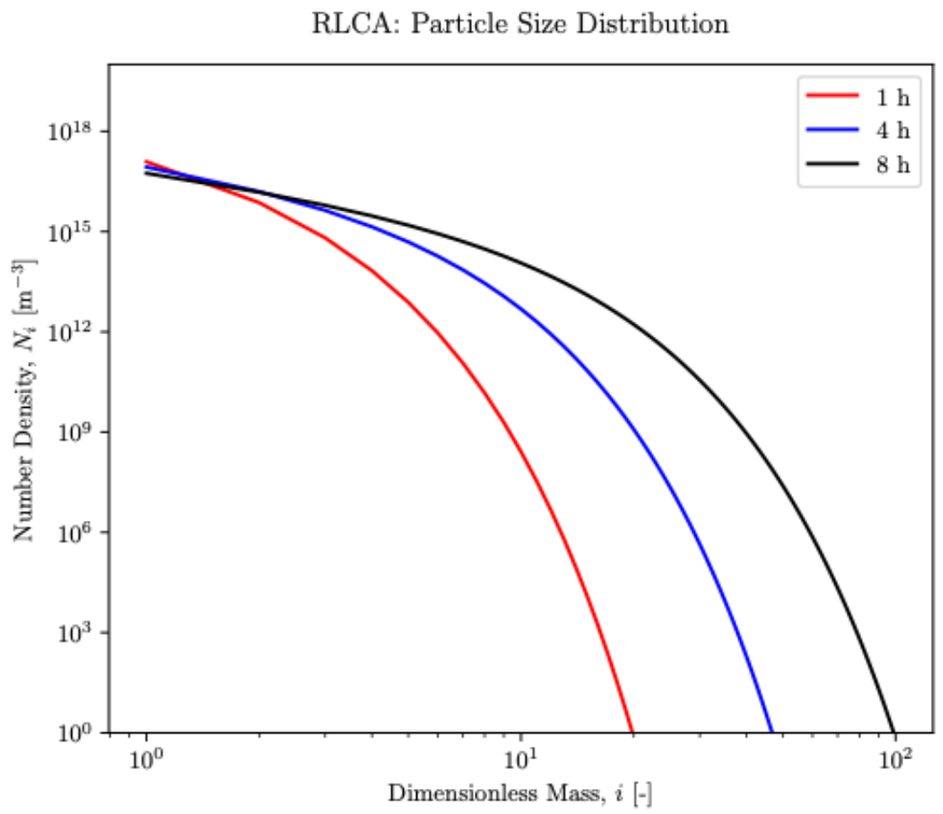


Figure 8: Particle size distributions after different times considering dimensionless masses between 1 and 100 in the reaction-limited regime.

The above results were obtained using the experimentally fitted value of the Fuchs stability ratio. It is clearly visible that the used model for simulation (with the restricted maximal non-dimensional mass of 100) cannot be used to simulate aggregation up to very large masses/sizes. Furthermore, at some point in time, the agglomeration will not be in the reaction-limited regime anymore and the few large clusters remaining will be diffusion-limited.

Python Code

Import of the relevant libraries.

In [1]:
```
# Import libraries and other preliminaries

import numpy as np                    # Import numerical python library
import matplotlib as mpl              # Import python mathematical plotting library
import matplotlib.pyplot as plt       # Import python mathematical plotting library
import math
from mpl_toolkits.mplot3d import Axes3D
from scipy.integrate import solve_ivp # Import solver for (system of coupled) ODEs
from scipy.optimize import curve_fit  # Import non-linear fitting function
from scipy.integrate import quad      # Import numerical integration function
from PyPDF2 import PdfWriter, PdfReader, PdfMerger # Import library for editing pdf-files
import sys

# Use LaTeX font in plots:
plt.rcParams.update({
    'text.usetex': True,
    'font.family': 'serif',
    'font.serif': ['Computern Modern Roman'],
})

from matplotlib import ticker
```

Function to crop a pdf on the fly.

In [2]:
```
def crop_pdf(Name, LeftCrop, RightCrop, BottomCrop, TopCrop):
    """
    Function to crop a pdf with the given name in the current working directory
    """
    reader = PdfReader(Name+'.pdf')
    writer = PdfWriter()

    for page in reader.pages:

        lower_left_new_x_coordinate = page.cropbox.lower_left[0] + LeftCrop
        lower_left_new_y_coordinate = page.cropbox.lower_left[1] + BottomCrop

        lower_right_new_x_coordinate = page.cropbox.lower_right[0] - RightCrop
        lower_right_new_y_coordinate = page.cropbox.lower_right[1] + BottomCrop

        upper_left_new_x_coordinate = page.cropbox.upper_left[0] + LeftCrop
        upper_left_new_y_coordinate = page.cropbox.upper_left[1] - TopCrop

        upper_right_new_x_coordinate = page.cropbox.upper_right[0] - RightCrop
        upper_right_new_y_coordinate = page.cropbox.upper_right[1] - TopCrop

        page.mediabox.lower_right = (lower_right_new_x_coordinate, lower_right_new_y_coordinate)
        page.mediabox.lower_left = (lower_left_new_x_coordinate, lower_left_new_y_coordinate)
        page.mediabox.upper_right = (upper_right_new_x_coordinate, upper_right_new_y_coordinate)
        page.mediabox.upper_left = (upper_left_new_x_coordinate, upper_left_new_y_coordinate)

        writer.add_page(page)

    with open(Name+'Crop.pdf','wb') as fp:
        writer.write(fp)
```

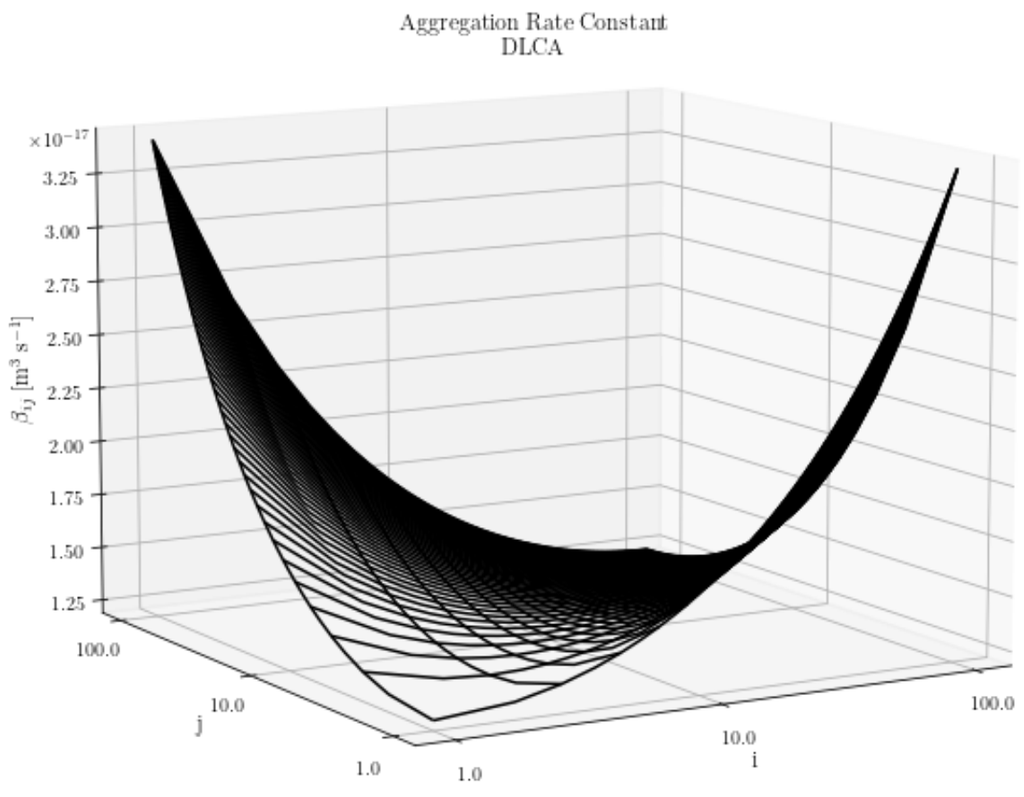
Part a)

In [3]:
```
def beta(k_B, T, eta, d_f, i, j):

    bij = np.zeros((len(i),len(j)))

    for idxi, elmnti in enumerate(i):
        for idxj, elmntj in enumerate(j):
            bij[idxi,idxj] = 2/3 * k_B*T/eta * (elmnti**(1/d_f)+elmntj**(1/d_f))*(elmnti**(-1/d_f)+elmntj**(-1/d_f))
    return bij
```

In [4]:
```
def beta_DLCA(k_B, T, eta, d_f, i, j):

    return 2/3 * k_B*T/eta * (i**(1/d_f)+j**(1/d_f))*(i**(-1/d_f)+j**(-1/d_f))
```

In [5]:
```
def TaskA():
    # Define parameters
    k_B = 1.380649e-23 # Boltzmann constant [J/K]
    T = 298.15         # Room temperature (25 deg C) [K]
    eta = 0.89e-3      # Dynamic viscosity [Pa s]
    d_f = 2.1          # Fractal dimension [-]

    # Define dim.less mass arrays
    i = np.linspace(1, 100, 100)
    j = np.linspace(1, 100, 100)

    # Create meshgrid
    I, J = np.meshgrid(i, j)

    # Calculate aggregation rate constants
    betaij = beta(k_B, T, eta, d_f, i, j)
    betaij2 = beta_DLCA(k_B, T, eta, d_f, I, J)

    # Set up a figure
    fig = plt.figure(figsize=(10,10))

    # Set up the axes for the first plot
    ax = fig.add_subplot(1, 1, 1, projection='3d')

    # Plot the data
    ax.plot_wireframe(np.log10(I), np.log10(J), betaij2, color='black')

    # Set log tick labels
    ax.set_xticks([0, 1, 2])
    ax.set_xticklabels([1e0, 1e1, 1e2])
    ax.set_yticks([0, 1, 2])
    ax.set_yticklabels([1e0, 1e1, 1e2])

    # Set x- and y-labels
    ax.zaxis.set_rotate_label(False)  # disable automatic rotation
    ax.set_xlabel(r'i',rotation=0, fontsize=12)
    ax.set_ylabel(r'j',rotation=0, fontsize=12)
    ax.set_zlabel(r'$\beta_{ij} \; [\mathrm{m}^3 \; \mathrm{s}^{-1}]$',rotation=90, fontsize=12)

    ax.set_title('Aggregation Rate Constant \n DLCA',y=0.94, pad=-14)

    ax.view_init(10, 240)

    formatter = ticker.ScalarFormatter(useMathText=True)
    formatter.set_scientific(True)
    formatter.set_powerlimits((-1,1))
    ax.zaxis.set_major_formatter(formatter)

    # Change orientation of offset text (10^-17)
    ax.zaxis.get_offset_text().set_visible(False)
    exponent = int('{:.2e}'.format(np.min(betaij2)).split('e')[1])
    ax.text(ax.get_xlim()[1]*(-0.2), ax.get_ylim()[1], ax.get_zlim()[1],
            '$\\times\\mathdefault{10^{%d}}\\mathdefault{}$' % exponent)

    # Save plot
    filename = 'Plot_BPRE_Ex3_1'
    plt.savefig(filename+'.pdf', bbox_inches='tight')

    # Crop plot and save again
    crop_pdf(filename, 0, 0, 50, 50)
```

In [6]:
```
# Execute task a)
TaskA()
```



Part b)

In [7]:

```python
def odesystem(t, y, MinMass, MaxMass, kNumber, betaik):
    """
    Return a right-hand side (RHS) vector of the system of ODEs to be solved.
    ODEs are written in the following form: dy/dt = f(y, t) = RHS
    The following system of coupled ODEs is considered:
    dy/dt = dNk/dt = ... for all k's
    and thus y = [N1, N2, N3, ..., Nk]
    """

    # Rename array of variables
    N = y

    # Initialize empty arrays
    dNk_dt = np.zeros(kNumber)
    sum1 = np.zeros(kNumber)
    sum2 = np.zeros(kNumber)
    prod1 = np.zeros(kNumber)
    prod2 = np.zeros(kNumber)
    odesys = np.zeros(kNumber)

    # Compute the terms in the Smoluchowski differential equation
    # Remark: the value for N1 is stored at index 0 of the N[k] vector etc.

    for k in range(kNumber):  # Range starts at 0 and ends at kNumber - 1 (total = kNumber)

        # First sum
        for i in range(k):  # Range starts at 0 and ends at k - 1

            sum1[k] += betaik[i,k-i-1]*N[i]*N[k-i-1]
            # Remark: in terms where the difference of the indices is computed, an
            #         aditional subtraction is required as the subtracted index i starts at 0
            #         instead of 1 and thus a shift by 1 unit has to be added!


        # Second sum
        for i in range(kNumber):

            sum2[k] += betaik[i,k]*N[i]

        # First product
        prod1[k] = 1/2 * sum1[k]

        # Second product
        prod2[k] = N[k] * sum2[k]

    # Compute complete system of ODEs
    dNk_dt = prod1 - prod2

    return dNk_dt
```

In [8]:

```
def solve_ode_DLCA(Param):
    """
    Solve the system of coupled ODEs
    """
    global k_B, T, eta, d_f

    # Read-out parameters and initial conditions
    MinMass = Param[0]
    MaxMass = Param[1]
    kNumber = Param[2]
    tMax = Param[3]
    tStep = Param[4]
    N_0 = Param[5]

    # Define dim.less mass arrays
    i = np.linspace(MinMass, MaxMass, kNumber)
    k = np.linspace(MinMass, MaxMass, kNumber)

    # Create meshgrid
    I, K = np.meshgrid(i, k)

    # Calculate aggregation rate constants
    betaik = beta_DLCA(k_B, T, eta, d_f, I, K)

    # Define initial condition vector
    y0 = N_0

    # Define time limits
    tlimits = [0, tMax]
    trange = np.arange(0, tMax+tStep, tStep)

    # Define additional ODE parameters
    Param_ODE = [MinMass, MaxMass, kNumber, betaik]

    # Solve system of ODEs (use Runge-Kutta 4-5th order for numerical integration)
    Sol = solve_ivp(odesystem, tlimits, y0, args=Param_ODE, method='RK45', t_eval=trange)
    t = Sol.t
    Nk = Sol.y

    # Calculate sum for each time
    N_tot = np.zeros(len(t))
    N_tot = np.sum(Nk, axis=0)

    # Calculate relative number
    N_rel = np.zeros(len(t))
    N_rel = N_tot / N1_0

    return N_rel, t
```

In [9]:

```
def TaskB():
    """
    Solve and plot the system of ODEs for DLCA regime
    """
    # Define parameters
    global k_B, T, eta, d_f
    k_B = 1.380649e-23 # Boltzmann constant [J/K]
    T = 298.15         # Room temperature (25 deg C) [K]
    eta = 0.89e-3      # Dynamic viscosity [Pa s]
    d_f = 2.1          # Fractal dimension [-]

    # Minimal and maximal considered mass
    MinMass = 1        # Minimal dim.-less mass [-]
    MaxMass = 100      # Maximal dim.-less mass [-]

    # Calculate total number of considered non-dim. masses (only considering integer steps)
    kNumber = MaxMass-MinMass+1

    # Maximal time for integration
    tMax = 3*60          # Maximal time [s]
    tStep = 0.01         # Time step size [s]

    # Initial condition
    global N1_0
    N1_0 = 1.4e17           # Initial number of primary particles (i.e. mass/size = 1) [m^-3]
    N_0 = np.zeros(kNumber) # Array of inital numbers of all sizes
    N_0[0] = N1_0           # Add size of primary particles

    # Parameter array
    Param = [MinMass, MaxMass, kNumber, tMax, tStep, N_0]

    # Create and solve system of ODEs
    N_rel, t = solve_ode_DLCA(Param)

    # Plot figure
    fig, axs = plt.subplots(nrows=1, ncols=1, figsize = (6,5)) # Create figure with one plot
    plot = axs.plot(t/60, N_rel, color='black')      # Create plot of N_tot [m^-3] vs. t [s]
    axs.set_xlabel('Time, $t$ [min]')   # Add x-axis label to plot
    axs.set_ylabel(r'Relative Total Number of Colloids, $N_{\mathrm{tot}}/N_p^0$ [-]')  # Add y-axis label to plot

    axs.set_title('DLCA',y=1.1, pad=-14)

    # Save plot
    filename = 'Plot_BPRE_Ex3_2'
    plt.savefig(filename+'.pdf', bbox_inches='tight')
```

In [10]:

```
# Execute task b)
#TaskB()
```

Part c)

In [11]:
```
def beta_RLCA(beta_DLCA, W, i, j, lamb):
    return beta_DLCA/W * (i*j)**lamb
```

In [12]:
```
def solve_ode_RLCA(Param, W):
    """
    Solve the system of coupled ODEs for RLCA regime
    """
    global k_B, T, eta, d_f, lamb

    # Read-out parameters and initial conditions
    MinMass = Param[0]
    MaxMass = Param[1]
    kNumber = Param[2]
    tMax = Param[3]
    tStep = Param[4]
    N_0 = Param[5]

    # Define dim.less mass arrays
    i = np.linspace(MinMass, MaxMass, kNumber)
    k = np.linspace(MinMass, MaxMass, kNumber)

    # Create meshgrid
    I, K = np.meshgrid(i, k)

    # Calculate aggregation rate constants
    betaik0 = beta_DLCA(k_B, T, eta, d_f, I, K)
    betaik = beta_RLCA(betaik0, W, I, K, lamb)

    # Define initial condition vector
    y0 = N_0

    # Define time limits
    tlimits = [0, tMax]
    trange = np.arange(0, tMax+1, tStep)

    # Define additional ODE parameters
    Param_ODE = [MinMass, MaxMass, kNumber, betaik]

    # Solve system of ODEs (use Runge-Kutta 4-5th order for numerical integration)
    Sol= solve_ivp(odesystem, tlimits, y0, args=Param_ODE, method='RK45', t_eval=trange)
    t = Sol.t
    Nk = Sol.y

    # Calculate sum for each time
    N_tot = np.zeros(len(t))
    N_tot = np.sum(Nk, axis=0)

    # Calculate relative number
    N_rel = np.zeros(len(t))
    N_rel = N_tot / N1_0

    return N_rel, t
```

In [13]:
```
def TaskC():

    #### Part 1 #### Non-linear least-squares regression to find optimal W

    # Define parameters
    global k_B, T, eta, d_f, lamb, MinMass, MaxMass, kNumber, tMax, tStep, N_0, N1_0
    k_B = 1.380649e-23 # Boltzmann constant [J/K]
    T = 298.15         # Room temperature (25 deg C) [K]
    eta = 0.89e-3      # Dynamic viscosity [Pa s]
    d_f = 2.1          # Fractal dimension [-]
    W0 = 0.5e5         # Guess for the Fuchs stability ratio

    # Calculate lambda [-]
    lamb = (d_f-1)/d_f

    # Minimal and maximal considered mass
    MinMass = 1        # Minimal dim.-less mass [-]
    MaxMass = 100      # Maximal dim.-less mass [-]

    # Calculate total number of considered non-dim. masses (only considering integer steps)
    kNumber = MaxMass-MinMass+1

    # Maximal time for integration
    tMax = 24*3600      # Maximal time [s]
    tStep = 2*3600      # Time step [s]

    # Initial condition
    N1_0 = 1.4e17           # Initial number of primary particles (i.e. mass/size = 1) [m^-3]
    N_0 = np.zeros(kNumber) # Array of inital numbers of all sizes
    N_0[0] = N1_0           # Add size of primary particles

    # Parameter array
    Param = [MinMass, MaxMass, kNumber, tMax, tStep, N_0]

    # Experimental data
    N_rel_exp = [1.00, 0.89, 0.79, 0.67, 0.59, 0.51, 0.46, 0.43, 0.35, 0.32, 0.28, 0.27, 0.21]
    t_exp = np.arange(0, 24+1, 2)   # Add 1 since np.arange excludes stop value

    # Define function used for regression
    def model_nls(t_exp, W):
        """
        Function required for the scipy.optimization.curve_fit function for non-linear regression
        """
```

```
    N_rel, t = solve_ode_RLCA(Param, W)
    return N_rel

# Non-linear least squares regression
Wopt, Wcov = curve_fit(model_nls, t_exp, N_rel_exp, W0)

print('The optimal W was found to be:', Wopt)

# Create and solve system of ODEs
N_rel, t = solve_ode_RLCA(Param, Wopt)

# Plot figure
fig, axs = plt.subplots(nrows=1, ncols=1, figsize = (6,5))  # Create figure with one plot
plot = axs.plot(t/3600, N_rel, color='black', label='Non-linear Fit')               # Create plot of N_tot [m^-3] vs. t [s]
expplot = axs.scatter(t_exp, N_rel_exp, color='black', label='Experimental Data')      # Create plot of N_tot [m^-3] vs. t [s]
axs.set_xlabel('Time, $t$ [h]')   # Add x-axis label to plot
axs.set_ylabel(r'Relative Total Number of Colloids, $N_{\mathrm{tot}}/N_p^0$ [-]')  # Add y-axis label to plot
axs.set_ylim([0, 1])

axs.set_title('RLCA',y=1.1, pad=-14)
axs.legend()

# Save plot
filename = 'Plot_BPRE_Ex3_3-1'
plt.savefig(filename+'.pdf', bbox_inches='tight')


#### Part 2 #### Plotting of the RLCA aggregation rate constant

# Define dim.less mass arrays
i = np.linspace(1, 100, 100)
j = np.linspace(1, 100, 100)

# Create meshgrid
I, J = np.meshgrid(i, j)

# Calculate aggregation rate constants
betaij_DLCA = beta_DLCA(k_B, T, eta, d_f, I, J)

betaij_RLCA = beta_RLCA(betaij_DLCA, Wopt, I, J, lamb)

# Set up a figure
fig = plt.figure(figsize=(10,10))

# Set up the axes for the first plot
ax = fig.add_subplot(1, 1, 1, projection='3d')

# Plot the data
ax.plot_wireframe(np.log10(I), np.log10(J), betaij_RLCA, color='black')

# Set log tick labels
ax.set_xticks([0, 1, 2])
ax.set_xticklabels([1e0, 1e1, 1e2])
ax.set_yticks([0, 1, 2])
ax.set_yticklabels([1e0, 1e1, 1e2])

# Set x- and y-labels
ax.zaxis.set_rotate_label(False)  # disable automatic rotation
ax.set_xlabel(r'i',rotation=0, fontsize=12)
ax.set_ylabel(r'j',rotation=0, fontsize=12)
ax.set_zlabel(r'$\beta_{ij} \; [\mathrm{m}^3 \; \mathrm{s}^{-1}]$',rotation=90, fontsize=12)

ax.set_title('Aggregation Rate Constant \n RLCA',y=0.94, pad=-14)

ax.view_init(10, 240)

formatter = ticker.ScalarFormatter(useMathText=True)
formatter.set_scientific(True)
formatter.set_powerlimits((-1,1))
ax.zaxis.set_major_formatter(formatter)

    # Change orientation of offset text (10^-17)
    ax.zaxis.get_offset_text().set_visible(False)
    exponent = int('{:.2e}'.format(np.min(betaij_RLCA)).split('e')[1])
    ax.text(ax.get_xlim()[1]*(-0.2), ax.get_ylim()[1], ax.get_zlim()[1],
            '$\\times\\mathdefault{10^{%d}}\\mathdefault{}$' % exponent)

    # Save plot
    filename = 'Plot_BPRE_Ex3_3-2'
    plt.savefig(filename+'.pdf', bbox_inches='tight')

    # Crop plot and save again
    crop_pdf(filename, 0, 0, 50, 50)
```

In [14]:

```
# Execute task c)
#TaskC()
```

Part d)

In [15]:

```
def VA(l, A_H):
    return - A_H/6 * np.add(np.add(2/(np.power(l,2)-4), 2/np.power(l,2)), np.log(1-4/np.power(l,2)))
```

In [16]:

```
def VR(l, a, eps_0, eps_r, psi_0, kappa):
    return 4*np.pi*eps_0*eps_r*a*psi_0**2/l * np.log(1+np.exp(-kappa*a*(l-2)))
```

In [17]:
```
def VT(l, a, A_H, eps_0, eps_r, psi_0, kappa):
    V_A = VA(l, A_H)
    V_R = VR(l, a, eps_0, eps_r, psi_0, kappa)
    return V_A+V_R
```

In [18]:
```
def Ion(c_i, z_i):
    return 1/2 * np.sum(c_i * np.power(z_i,2))
```

In [19]:
```
def kap(I):
    return 3.29 * np.sqrt(I) * 1e9
```

In [20]:
```
def W_Integrand(l, Param):
    k_B, T, A_H, eps_0, eps_r, a, psi_0, kappa = Param
    return np.exp( VT(l, a, A_H, eps_0, eps_r, psi_0, kappa) / (k_B*T) ) * 1/(a*np.power(l,2))
```

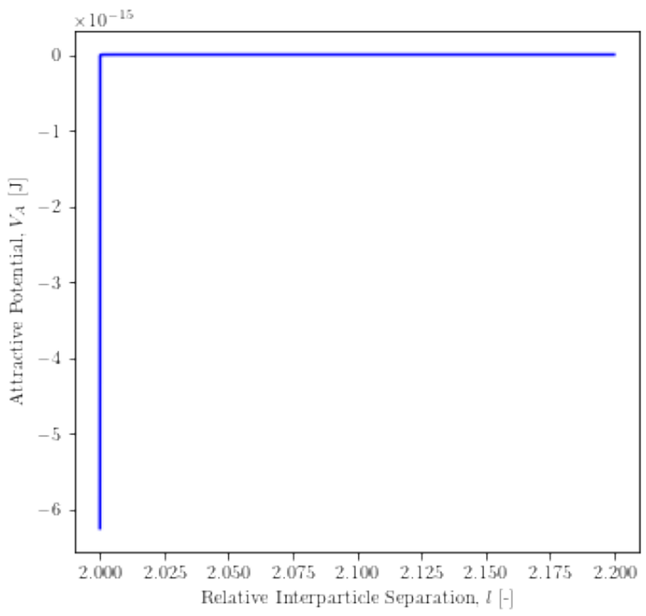
In [21]:
```
def W(Integral, a):
    return 2*a*Integral
```

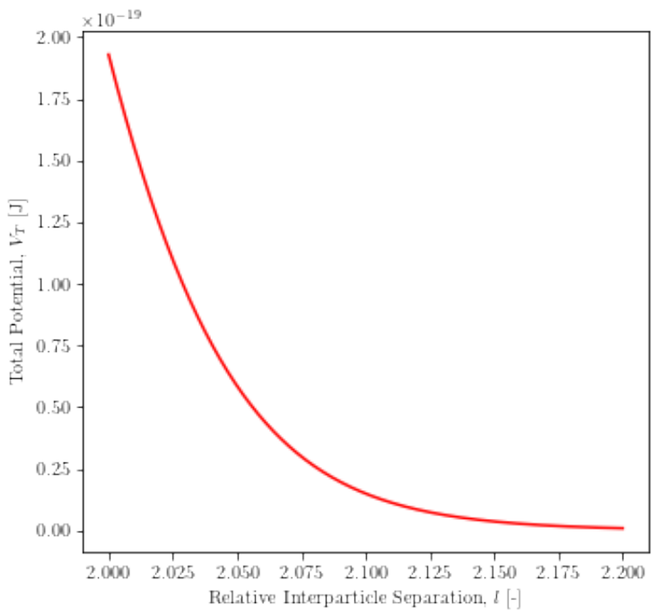
In [22]:
```
def TaskD():
    # Define parameters
    k_B = 1.380649e-23      # Boltzmann constant [J/K]
    T = 25+273.15           # Temperature [K]
    A_H = 1.5e-20           # Hamacker constant [J]
    eps_0 = 8.85e-12        # Dielectric constant [F/m]
    eps_r = 80.1            # Some additional electric factor [-]
    a = 100e-9              # Radius of primary particle [m]
    psi_0 = 25e-3           # Potential [V]
    d_f = 2.1               # Fractal dimension [-]
    c_i = np.ones(2)*7.5e-3 # Concentrations (NaCl) [mol/L]
    z_i = np.ones(2)        # Valencies (NaCl) [-]

    # Define evaluated limits for l (= r/a hence the interparticle distance over the radius of the primary particle)
    lmin = 2
    lmax = 2.2
    lnum = 1000000

    # Calculate ionic strength and kappa
    I = Ion(c_i, z_i)
    kappa = kap(I)

    # Define parameter array
    global Param
    Param = [k_B, T, A_H, eps_0, eps_r, a, psi_0, kappa]

    # Numerically solve integrand for Fuchs stability ratio
    Integral, Err = quad(W_Integrand, lmin, np.inf, args=Param)

    # Calculate Fuchs stability ratio
    Wcalc = W(Integral, a)

    print('The value found for the Fuchs stability ratio is:', Wcalc)

    # Create plots to check potentials and integrand
    lrange = np.linspace(lmin, lmax, lnum)
    V_A = VA(lrange, A_H)
    V_R = VR(lrange, a, eps_0, eps_r, psi_0, kappa)
    V_T = VT(lrange, a, A_H, eps_0, eps_r, psi_0, kappa)
    Integrand = W_Integrand(lrange, Param)

    # Plot potentials
    fig, axs = plt.subplots(nrows=1, ncols=3, figsize = (18,5))  # Create figure with one plot
    plot1 = axs[0].plot(lrange, V_A, 'b')
    plot2 = axs[1].plot(lrange, V_R, 'r')
    plot3 = axs[2].plot(lrange, V_T, 'r')

    axs[0].set_xlabel(r'Relative Interparticle Separation, $l$ [-]')
    axs[0].set_ylabel(r'Attractive Potential, $V_A$ [J]')

    axs[1].set_xlabel(r'Relative Interparticle Separation, $l$ [-]')
    axs[1].set_ylabel(r'Repulsive Potential, $V_R$ [J]')

    axs[1].set_xlabel(r'Relative Interparticle Separation, $l$ [-]')
    axs[1].set_ylabel(r'Total Potential, $V_T$ [J]')

    # Save plot
    filename = 'Plot_BPRE_Ex3_4-1'
    plt.savefig(filename+'.pdf', bbox_inches='tight')

    # Plot integrand
    fig, axs = plt.subplots(nrows=1, ncols=1, figsize = (6,5))  # Create figure with one plot
    plot1 = axs.plot(lrange, Integrand, 'black')

    axs.set_xlabel(r'Relative Interparticle Separation, $l$ [-]')
    axs.set_ylabel(r'Integrand, $ \mathrm{exp} \left( \frac{V_T}{k_\mathrm{B}T} \right) \frac{1}{r^2}$ [m$^{-2}$]')

    # Save plot
    filename = 'Plot_BPRE_Ex3_4-2'
    plt.savefig(filename+'.pdf', bbox_inches='tight')
```

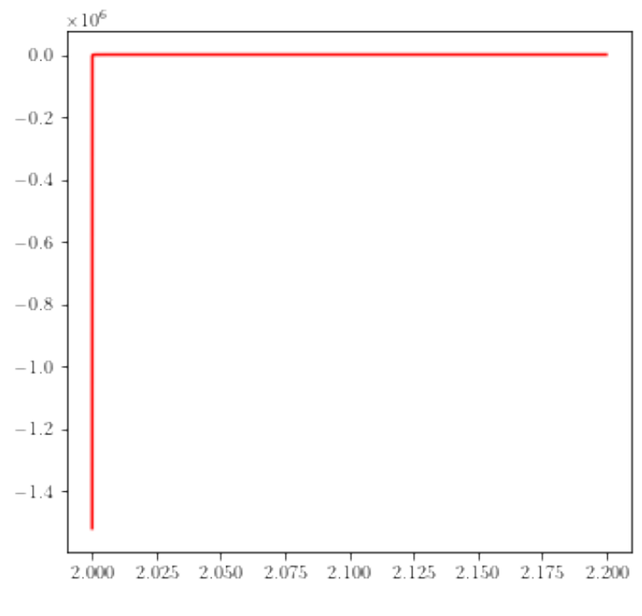
In [23]:
```
# Execute task d)
TaskD()
```

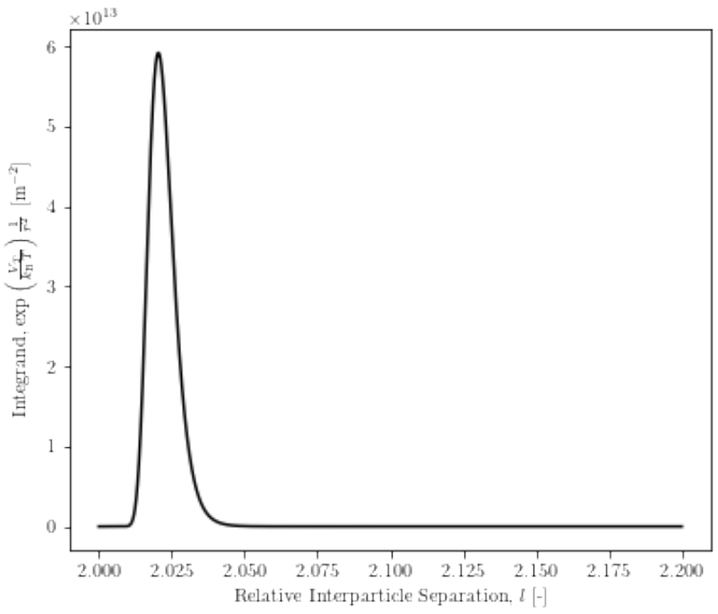
```
The value found for the Fuchs stability ratio is: 130039.539272088
```

```
/var/folders/b3/hlft3w2x583166xmfn5vcnpr0000gn/T/ipykernel_10028/884504962.py:2: RuntimeWarning: divide by zero encountered in true_divide
  return - A_H/6 * np.add(np.add(2/(np.power(l,2)-4), 2/np.power(l,2)), np.log(1-4/np.power(l,2)))
/var/folders/b3/hlft3w2x583166xmfn5vcnpr0000gn/T/ipykernel_10028/884504962.py:2: RuntimeWarning: divide by zero encountered in log
  return - A_H/6 * np.add(np.add(2/(np.power(l,2)-4), 2/np.power(l,2)), np.log(1-4/np.power(l,2)))
/var/folders/b3/hlft3w2x583166xmfn5vcnpr0000gn/T/ipykernel_10028/884504962.py:2: RuntimeWarning: invalid value encountered in add
  return - A_H/6 * np.add(np.add(2/(np.power(l,2)-4), 2/np.power(l,2)), np.log(1-4/np.power(l,2)))
```









Part e)

In [24]:

```python
def solve_ode_RLCA_dist(Param, W):
    """
    Solve the system of coupled ODEs for RLCA regime
    """
    global k_B, T, eta, d_f, lamb

    # Read-out parameters and initial conditions
    MinMass = Param[0]
    MaxMass = Param[1]
    kNumber = Param[2]
    tMax = Param[3]
    tStep = Param[4]
    N_0 = Param[5]

    # Define dim.less mass arrays
    i = np.linspace(MinMass, MaxMass, kNumber)
    k = np.linspace(MinMass, MaxMass, kNumber)

    # Create meshgrid
    I, K = np.meshgrid(i, k)

    # Calculate aggregation rate constants
    betaik0 = beta_DLCA(k_B, T, eta, d_f, I, K)
    betaik = beta_RLCA(betaik0, W, I, K, lamb)

    # Define initial condition vector
    y0 = N_0

    # Define time limits
    tlimits = [0, tMax]
    trange = np.arange(0, tMax+1, tStep)

    # Define additional ODE parameters
    Param_ODE = [MinMass, MaxMass, kNumber, betaik]

    # Solve system of ODEs (use Runge-Kutta 4-5th order for numerical integration)
    Sol= solve_ivp(odesystem, tlimits, y0, args=Param_ODE, method='RK45', t_eval=trange)
    t = Sol.t
    Nk = Sol.y

    return Nk, t, i
```

In [25]:
```
def TaskE():

    # Define parameters
    global k_B, T, eta, d_f, lamb, MinMass, MaxMass, kNumber, tMax, tStep, N_0, N1_0
    k_B = 1.380649e-23 # Boltzmann constant [J/K]
    T = 298.15         # Room temperature (25 deg C) [K]
    eta = 0.89e-3      # Dynamic viscosity [Pa s]
    d_f = 2.1          # Fractal dimension [-]
    Wopt = 0.4734e5    # Optimal value for the Fuchs stability ratio

    # Calculate lambda [-]
    lamb = (d_f-1)/d_f

    # Minimal and maximal considered mass
    MinMass = 1        # Minimal dim.-less mass [-]
    MaxMass = 100      # Maximal dim.-less mass [-]

    # Calculate total number of considered non-dim. masses (only considering integer steps)
    kNumber = MaxMass-MinMass+1

    # Maximal time for integration
    tMax = 8*3600       # Maximal time [s]
    tStep = 1*3600      # Time step [s]

    # Initial condition
    N1_0 = 1.4e17           # Initial number of primary particles (i.e. mass/size = 1) [m^-3]
    N_0 = np.zeros(kNumber) # Array of inital numbers of all sizes
    N_0[0] = N1_0           # Add size of primary particles

    # Parameter array
    Param = [MinMass, MaxMass, kNumber, tMax, tStep, N_0]


    # Create and solve system of ODEs
    Nk, t, i = solve_ode_RLCA_dist(Param, Wopt)

    # Extract distribution at different times
    Nk_1h = Nk[:,1]
    Nk_4h = Nk[:,4]
    Nk_8h = Nk[:,8]

    # Plot integrand
    fig, axs = plt.subplots(nrows=1, ncols=1, figsize = (6,5))  # Create figure with one plot
    plot1 = axs.loglog(i, Nk_1h, 'red', label='1 h')
    plot2 = axs.loglog(i, Nk_4h, 'blue', label='4 h')
    plot3 = axs.loglog(i, Nk_8h, 'black', label='8 h')


    axs.set_xlabel(r'Dimensionless Mass, $i$ [-]')
    axs.set_ylabel(r'Number Density, $N_i$ [m$^{-3}$]')
    axs.set_ylim([1e0, 1e20])

    axs.set_title('RLCA: Particle Size Distribution',y=1.1, pad=-14)
    axs.legend()

    # Save plot
    filename = 'Plot_BPRE_Ex3_5-1'
    plt.savefig(filename+'.pdf', bbox_inches='tight')
```

In [26]:
```
# Execute task e)
TaskE()
```