Parameter Estimation for Breakage and Coalescence Kernels in a Population Balance Framework

Sindre Bakke Øyen

Chemical Engineering and Biotechnology
Submission date: June 2018
Supervisor: Brian Arthur Grimes, IKP

Norwegian University of Science and Technology
Department of Chemical Engineering

Summary

In this work, a zero-dimensional model in the population balance framework has been developed for an oil-in-water emulsion in a batch continuously stirred tank reactor in order to calibrate four parameters to experimental data: two parameters for breakage and two parameters for coalescence. Three different regression approaches were employed: sum of squared errors for the distributions (dSSE), weighted sum of squared errors for the distributions (wdSSE) and sum of squared errors for the statistical mean (mSSE). The non-linear regressions required good initial guesses, and therefore an appropriate parameter space was charted in two directions, keeping the breakage parameters constant. The residual three-dimensional surface for the dSSE had a long, twisting valley with a clear minimum. However, it also had large areas of flat gradients. Given a sufficiently good initial guess, gradient-based optimizers should therefore be applicable to the problem. The same applied for the wdSSE, however, the valley was narrower and shallower. The mSSE had multiple local minima, but the same valley was also observed here.

The previous findings for fixed breakage parameters were promising, and consequently, the parameter search was augmented to four directions, resulting in a disproportional increase in computational complexity. A coarse grid with lower resolution was therefore used for an initial charting. Thereafter, the areas of flat gradients were cut out, and the surfaces were refined with a higher grid point density. The best solutions found for both the coarse and the refined grid were then provided as initial guesses to a Levenberg-Marquardt optimization routine from the GNU Scientific Library. A sensitivity analysis was performed on the optimal parameter combination found for the coarse grid by perturbing the initial guess. It was discovered that the optimal solutions found were also perturbed from the nominal case. The findings were ambiguous and inconclusive. For future work, the internal parameters of the routine should be adjusted in order to ensure desirable convergence properties, and to ensure that the global minimum is found given that a point of descension is provided.

Since measurement devices are expensive, it was also desirable to find a minimum number of measurements required to obtain the same optimal parameter combination. The best solutions found for both the coarse grid and the refined grid were subject to the experiment, and for the dSSE, both initial guesses needed about 30 measurements of the distribution in time for all parameters but one to stabilize. The objective function value per degree of freedom for the wdSSE stabilized at 20 measurements in time for both initial guesses, however, it produced oscillatory behaviors for three parameters for the coarse initial guess and two parameters for the refined initial guess. If the internal parameters of the optimization routine is to be optimized in order to ensure consistent convergence, this experiment should also be repeated to investigate whether the oscillations are connected to the objective function formulation or internally to the optimization routine employed.

Finally, the optimal parameter combination found for all three regression approaches were checked for validity by comparing their dynamical behaviors, their steady state locations and their temporal evolutions of the mean to the experimental measurements. It was found that the dSSE and the wdSSE had very similar behaviors, and since the wdSSE was inferior in every other way discovered, its future use is discouraged. The dynamical behavior of the two was quicker than the experimental observations. At first, the distribution widened, pushing the peak of the distribution down, before it narrowed and pushed the peak up again. The findings contradict the experimental data. Since the model failed to account for the production of the long tail for small droplets, the distribution could not widen at the end to push the peak down again. Therefore the modeled dynamical behavior could not represent the observations properly. The modeled distribution reached steady state somewhat early, but the agreement with the experimental steady state was satisfactory, and both the mode of the distribution and the pronounced gradient at the end due to coalescence were approximately correct. The mSSE, on the other hand, narrowed and kept rising until it reached steady state. For the same reasoning as before, the distribution could not widen again by producing small droplets, and therefore it failed to settle down at the correct peak value. The dynamical behavior was too fast, and the steady state location was unsatisfactory. The tracking of the mean was slightly better for this regression approach, but not good enough to outweigh its discrepancies with respect to the dynamical and steady state behavior. Allover, the dSSE performed better than the other two, making it the best regression approach of the three employed.

Sammendrag

I denne oppgaven ble en null-dimensjonal modell utviklet i populasjonsbalanserammeverket for en olje-i-vann-emulsjon i en batch kontinuerlig røretankreaktor for å kalibrere fire parametre til eksperimentelle data: to parametre for bobleoppbrytning og to parametre for koalesens. Tre forskjellige fremgangsmåter for regresjon ble benyttet: minste kvadraters metode, vektet minste kvadraters metode og minste kvadraters metode på gjennomsnittet. Den ikke-lineære regresjonen trengte gode initiale gjett, og et passende parameterrom ble derfor kartlagt for fastsatte oppbrytningsparametre. Den tre-dimensjonale overflaten for minste kvadraters metode hadde en lang, svingende dal med et tydelig minimum i bunnen. Det var imidlertid også store områder med flate gradienter som kunne være vanskelig å navigere. Gitt et godt gjett bør fortsatt gradient-baserte algoritmer klare å finne veien til minimumet. Det samme gjaldt for vektet minste kvadraters metode, men her var dalen smalere og grunnere. Minste kvadraters metode på gjennomsnittet hadde flere lokale minima, men dalen ble også observert her.

De tidligere funnene for fastsatte oppbrytningsparametre så lovende ut, og derfor ble parametersøket utvidet til fire retninger, noe som resulterte i en disproporsjonal økning i beregningskompleksitet. Et grovt nett med lavere oppløsning enn tidligere ble derfor benyttet for en initiell kartlegging. Deretter ble områder med flat gradient kappet vekk, og overflatene av interesse ble raffinert med høyere nettpunkttetthet. De beste løsningene funnet både for det grove og det raffinerte nettet ble brukt som initiale gjett til en Levenberg-Marquardt-optimialiseringsrutine i GNU Scientific Library. Den optimale parameterkombinasjonen funnet for det grove nettet ble deretter testet for sensitivitet ved å forskyve det initiale gjettet. Det ble oppdaget at den optimale løsningen også ble forskjøvet av dette. Funnene var tvetydige og mangelfulle. For fremtidig arbeid anbefales det at de interne parametrene i den benyttede rutinen justeres for å sikre ønskede konvergensegenskaper, og for å sikre at det globale minimumet blir funnet gitt at rutinen forsynes med et nedstigende initielt punkt.

Siden måleapparater er kostbare, var det ønskelig å finne et minimum antall målinger nødvendig for å få samme optimale parameterkombinasjon. De beste løsningene funnet både for det grove nettet og det raffinerte nettet gjennomgikk eksperimentet, og for begge de initiale gjettene trengte minste kvadraters metode ca. 30 målinger av fordelingen i tid for at alle bortsett fra én parameter stabiliserte. Målfunksjonsverdien for vektet minste kvadraters metode stabiliserte på 20 målinger i tid for både det grove og det raffinerte initiale gjettet, men denne regresjonsmetoden skapte oscillerende oppførsel for tre parametre for det grove nettets initiale gjett og to parametre for det raffinerte nettets initiale gjett. Hvis de interne parametrene i den benyttede rutinen optimaliseres for konsistent konvergens bør dette eksperimentet gjentas for å etterforske om den oscillerende oppførselen er knyttet til målfunksjonsformuleringen eller internt til den benyttede optimaliseringsrutinen.

Til sist ble gyldigheten til de optimale parameterkombinasjonene testet for alle regresjonsstrategier ved å sammenlikne den dynamiske oppførselen, steady state posisjonen og tidsutviklingen til gjennomsnittet med eksperimentelle målinger. Det ble oppdaget at minste kvadraters metode og vektet minste kvadraters metode hadde veldig lik oppførsel, og siden vektet minste kvadraters metode var underlegen på alle andre måter er dens fremtidige bruk frarådet. Den dynamiske utviklingen til de to regresjonsstrategiene var raskere enn de eksperimentelle observasjonene. Først ble fordelingen bredere, noe som presset toppen ned, deretter ble fordelingen smalere, noe som presset toppen opp igjen. Dette er motsatt av hva som ble observert eksperimentelt. Siden modellen ikke tok hensyn til produksjonen av den lange halen for små dråpestørrelser kunne ikke fordelingen bli bredere i slutten for å dytte toppen ned igjen. Derfor kunne ikke den modellerte dynamikken reprodusere observasjonene korrekt. Den modellerte fordelingen når steady state noe tidlig, men det er god korrespondanse mellom modellert steady state og målt steady state, og både typetallet og den sterke kanten på høyre side på grunn av koalesens ble reprodusert rett. Minste kvadraters metode av gjennomsnittet, på den annen side, ble smalere i starten og fortsatte å stige helt til den nådde steady state. Ved samme argumentasjon som tidligere klarte ikke fordelingen å bli bredere ved å produsere små dråper, slik at toppen gikk ned igjen. Den dynamiske utviklingen var for rask og steady state posisjonen var ikke tilfredsstillende. Sporingen av gjennomsnittet, derimot, var litt bedre for denne regresjonsstrategien enn for de to andre, men ikke så mye bedre at den oppveier den manglende overensstemmelsen til den dynamiske og steady state oppførselen. Alt i alt var minste kvadraters metode den regresjonsstrategien som presterte best av de tre, noe som gjør den til den foretrukne regresjonsstragien.

Preface

This thesis is submitted as part of a Master's degree in collaboration with the Department of Chemical Engineering, NTNU, and SUBPRO - Subsea Production and Processing. The work is a continuation from the specialization project during the fall of 2017, a project that was funded by SUBPRO. The thesis contains work on experimental data calibration and parameter estimation on the zero-dimensional population balance equation developed for an oil-in-water emulsion in a batch continuously stirred tank reactor.

The author would like to express his gratitude to Associate Professor Brian Arthur Grimes for his exceptional assistance and patience along the way. He has been very helpful throughout the last year. The author would also like to thank Seok Ki Moon and Marcin Dudek for providing the author with experimental data. Last, but not least the author is grateful to SUBPRO for their financial support covering his attendance at the Population Balance Modeling Conference in Ghent, Belgium, May 2018.

Special thanks to Brian, Dag and Anja for proof-reading the thesis and being helpful throughout the process. Your comments, feedback and support are appreciated.

Trondheim, June 2018 Sindre Bakke Øyen

Table of Contents

List of Tables

List of Figures

List of Program Code

Abbreviations

B	Bottom
BC	Boundary condition
BVP	Boundary value problem
CSTR	Continuously stirred tank reactor
DQMOM	Direct quadrature method of moments
DSD	Droplet size distribution
dSSE	Sum of squared errors on the distributions
E	East
FVM	Finite volume method
GSL	GNU Scientific Library
H.O.T	Higher order terms
LSM	Least squares method
MOM	Method of moments
mSSE	Sum of squared errors on the mean
N	North
ODE	Ordinary differential equation
OFV	Objective function value
PBE	Population balance equation
PBM	Population balance modeling
PDE	Partial differential equation
PIDE	Partial integro-differential equation
QMOM	Quadrature method of moments
RHS	Right hand side
S	South
SMOM	Standard method of moments
SSE	Sum of squared errors
SV	Subvolume
T	Top
VDD	Volume density distribution
W	West
WRM	Weighted residual methods
wdSSE	Weighted sum of squared errors on the distributions

List of Latin Symbols

Symbol	Definition	Unit
A	Area under curve	-
a	Recurrence coefficient for Jacobi polynomials	-
a_C	Coalescence frequency	s^{-1}
B	Daughter redistribution function for ξ and ξ'	m^{-1}
b	Recurrence coefficient for Jacobi polynomials	-
b	Discretized source vector	-
B_B	Birth breakage	$\mathrm{m}^{-3}\,\mathbf{x}^{-1}\,\mathrm{s}^{-1}$
B_C	Birth coalescence	$\mathrm{m}^{-3}\,\mathbf{x}^{-1}\,\mathrm{s}^{-1}$
b_Γ	Source vector on Γ	-
b_Ω	Source vector on Ω	-
b^p	Source problem vector	-
c	Recurrence coefficient for Jacobi polynomials	-
d	Diameter	m
D_B	Death breakage	$\mathrm{m}^{-3}\,\mathbf{x}^{-1}\,\mathrm{s}^{-1}$
D_C	Death coalescence	$\mathrm{m}^{-3}\,\mathbf{x}^{-1}\,\mathrm{s}^{-1}$
e	Unit vector	-
F	Density function	-
f	Arbitrary function	-
f	Discretized function vector	-
f_Γ	Arbitrary boundary source function	-
f_n	Number density distribution	$\mathrm{m}^{-3}\,\mathbf{x}^{-1}$
$f_n^{(2)}$	Pair number density distribution	$\mathrm{m}^{-3}\,\mathbf{x}^{-1}$
f_v	Volume density distribution	m^{-1}
f_v^{exp}	Experimentally measured volume density distribution	m^{-1}
f_v^{num}	Numerically modeled volume density distribution	m^{-1}
G	Breakage frequency for dimensionless radius	s^{-1}
g	Arbitrary source function	-
g	Breakage frequency	s^{-1}
h	Step size	-
I	Definite integral	-
J	Objective function value	m^{-2} or $\mu\mathrm{m}^{2}$
K	Effective rate of coalescence for dimensionless radii	$\mathrm{m}^{3}\,\mathrm{s}^{-1}$
k	Chicken factor	-
k_1	Pre-factor for breakage	-
k_2	Exponential factor for breakage birth	-
k_3	Pre-factor for coalescence	-
k_4	Exponential factor for coalescence	-

K_{BB}	Extracted kernel for birth breakage	-
K_{BC}	Extracted kernel for birth coalescence	-
K_{DB}	Extracted kernel for death breakage	-
K_{DC}	Extracted kernel for death coalescence	-
$k_{b,1}$	Dynamic breakage parameter	-
$k_{b,2}$	Exponential breakage parameter	-
$k_{c,1}$	Dynamic coalescence parameter	-
$k_{c,2}$	Exponential coalescence parameter	-
N	Number of discretization points	-
N	Number of distributions chosen	-
n	Number of total residuals (N_pN_t)	-
N_p	Number of size classes measured	-
N_t	Number of measurements in time	-
P	Power supplied to emulsion by agitator	W
p	Number of parameters	-
p	Orthogonal polynomial	-
R	Residual vector	-
r	Radius	m
R_m	Maximum radius (characteristic length)	m
r_{eq}	Equidistant radius between droplets	m
r'	Radius of coalescing particle	m
r''	Radius of coalescing particle	m
S	Riemann sum	-
S	Source term	$\mathrm{m}^{-3}\,\mathbf{x}^{-1}\,\mathrm{s}^{-1}$
t	Time	s
t_c	Drainage time	s
t_f	Time of experiment (characteristic time)	s
t_i	Interaction time	s
u	Variable transformation of x	-
V	Volume	m^3
v	Eigenvector of $\mathbf{T}$	-
$V_{emulsion}$	Total volume in the emulsion	m^3
V_l	Volume of liquid emulsion in tank	m^3
V_m	Maximum volume	m^3
V_{oil}	Volume of oil in the emulsion	m^3
$V_{\mathbf{r}}$	Volume in external coordinate	m^3
$V_{\mathbf{x}}$	Volume in internal coordinate	$\mathbf{x}^3$
W	Weight function	-
w	Integral weight corresponding to some collocation point	-
w	Weight function for weighted sum of squared errors	-
x	Arbitrary independent coordinate	-
x_Ω	Collocation point vector on reference domain	-
w_Ω	Integral weight vector on reference domain	-
$\mathbf{A}$	Linear problem matrix on interior domain	-
$\mathbf{A}^p$	Linear problem matrix on interior and boundary domain	-

$\mathbf{B}$	Linear boundary matrix	-
$\mathbf{D}$	Golub-Welsch similarity transformation matrix	-
$\mathbf{g}$	Gravitational field	$\mathrm{m\,s^{-2}}$
$\mathbf{I}$	Identity matrix	-
$\mathbf{J}$	Golub-Welsch matrix	-
$\mathbf{J}$	Jacobian matrix	-
$\tilde{\mathbf{J}}$	Modified Golub-Welsch matrix for Gauss Lobatto grid	-
$\mathbf{M}$	Newton matrix	-
$\mathbf{r}$	External coordinate	$\mathrm{m^3}$
$\mathbf{T}$	Tridiagonal matrix	-
$\mathbf{v}$	Terminal velocity vector	$\mathrm{m\,s^{-1}}$
$\mathbf{v}_{rel,t,d,d'}$	Relative velocity between two particles	$\mathrm{m\,s^{-1}}$
$\mathbf{v_r}$	Space velocity	$\mathrm{m\,s^{-1}}$
$\mathbf{v_x}$	Phase velocity	$\mathbf{x}\,\mathrm{s^{-1}}$
$\mathbf{x}$	Internal coordinate	$\mathbf{x}$
$\mathbf{Y}$	Environment vector	-
$\mathbb{P}$	Set of partitions of independent coordinate	-
$\mathbb{Z}$	Set of all integers	-

List of Greek Symbols

Symbol	Definition	Unit
α	Basis coefficient	-
α	Dimensionless interpolated domain for birth coalescence	-
α	Jacobi polynomial parameter	-
β	Daughter redistribution function	$\mathbf{x}^{-1}$
β	Jacobi polynomial parameter	-
β	Parameter vector (decision variables)	-
Γ	Boundary domain	-
γ	Coefficient in **J**	-
γ	Dimensionless interpolated domain for birth breakage	-
γ	Variable order step size	-
Δ	Partition for sectional methods	-
δ	Dirac's delta function	-
δ	Number of particles coalescing	-
Δx	Change in variable x	-
ε	Turbulent energy dissipation rate	$\mathrm{m}^2\,\mathrm{s}^{-3}$
$\bar{\varepsilon}$	Spatially averaged turbulent energy dissipation rate	$\mathrm{m}^2\,\mathrm{s}^{-3}$
η	Vector to construct modified **J**	-
κ_C	Coalescence density	$\mathrm{m}^3\,\mathrm{s}^{-1}$
λ	Eigenvalue	-
μ	Vector to construct modified **J**	-
μ_0	Value of finite integral of W	-
μ_c	Continuous phase dynamic viscosity	$\mathrm{Pa\,s}$
μ_j	j-th moment of a function f	-
ν	Number of fragments born due to breakage	-
ξ	Dimensionless radius	-
ξ'	Dimensionless radius for mother particle	-
ξ''	Dimensionless radius for mother particle	-
ρ_c	Continuous phase density	$\mathrm{kg\,m}^{-3}$
ρ_d	Dispersed phase density	$\mathrm{kg\,m}^{-3}$
σ	Surface tension	$\mathrm{N\,m}^{-1}$
τ	Coefficient in **J**	-
τ	Dimensionless time	-
ϕ	Trial function	-
φ	Volume fraction of oil in the emulsion	-
ψ	Dimensionless volume density distribution	-
Ψ_E	Probability of coalescence for dimensionless radii	-
ψ_E	Probability of coalescence	-

Ω	Interior domain	-
Ω	Swept volume rate for dimensionless radii	-
ω	Subdomain of Ω	-
ω	Swept volume rate	$\mathrm{m}^3\,\mathrm{s}^{-1}$
ω	Weight function in weighted resiudal methods	-
$\omega_{\Gamma,I}$	Weight function on Γ at collocation point I	-
$\omega_{\Omega,I}$	Weight function on Ω at collocation point I	-
$\nabla_{\mathbf{x}}\cdot$	Phase divergence operator	$\mathbf{x}^{-1}$
$\nabla_{\mathbf{r}}\cdot$	Space divergence operator	m^{-1}
$\mathcal{B}$	Boundary linear function operator	-
$\mathcal{L}$	Linear function operator	-
ℓ	Lagrange interpolating polynomial	-
$\mathcal{O}$	Order of accuracy	-
$\mathcal{R}_{\Gamma}$	Residual on Γ	-
$\mathcal{R}_{\Omega}$	Residual on Ω	-

Chapter 1

Introduction

More than often, engineers are troubled with multi-phase systems where the product is dependent on the particles involved. For a fluid-liquid emulsion, for instance, liquid droplets or gas bubbles are dissolved in a continuous liquid phase, and the settling velocity, given by Stoke's law, is governed primarily by the droplet sizes. For hydrocyclones, the droplet size will determine whether or not the droplet escapes the centrifugal forces or not, and thus separation efficiency is directly correlated to the size of the droplet. In other systems, such as bioreactors, the size of the cells are among the factors that determine the production rate of a desired compound, but also the *population* of these cells are critical, as the the total production rate is dependent on the production of each individual cell in the presence of other cells in that particular environment. The droplet, bubble or particle surrounded by such an environment is commonly referred to as the dispersed phase and the environment is referred to as the continuous phase, both of which are important for the properties of the total system.

As noted for the bioreactor example, the population of the cells are important for the production rate. The classical transport equations applies to the behavior of single particles [1], which motivates the development of an equation that acts as a conservation of mass, volume (if applicable) or number of particles in the population of particles, droplets or bubbles. The equation is reasonably called a population balance equation, and in the last 25 years this field of research has grown almost quadratically, as seen in Figure 1.1. The increase may be related to the book written by Ramkrishna [1] in 2000, as there is little growth from 1993 to 2000. The increasing interest in the field might also be a natural consequence of an increasing level of complexity in modeling itself. Nevertheless, the topic provides a detailed description of a dispersed system of multivariate populations, and its use might be manifold, as seen above.

1.1 Motivation

Consider the gravity separator presented by Backi et al. [2] with an oil-in-water emulsion, seen in Figure 1.2, where a mixture of oil, water and gas is fed in. The water phase has

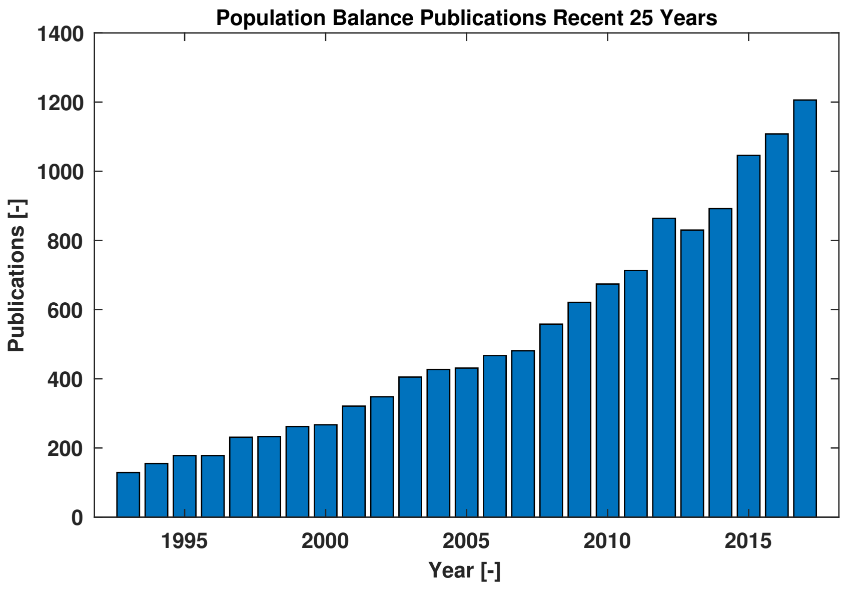


Figure 1.1: The bar plot shows an increasing interest in the field of population balance modeling. The data was downloaded from Webofknowledge May 21, 2018, with the keyword “population balance modeling”.

a higher density, and thus its continuous phase lies at the bottom. Oil droplets dispersed in the water phase, and water droplets dispersed in the oil phase, will travel according to Stoke’s law

$$\mathbf{v} = \frac{\mathbf{g}d^2(\rho_d - \rho_c)}{18\mu_c}, \tag{1.1}$$

where **v** is the terminal velocity vector, **g** is the gravitational field, d is the droplet diameter, ρ_d and ρ_c are the dispersed and continuous phase densities, respectively, and μ_c is the dynamic viscosity of the continuous phase. The terminal settling velocity determines how fast the droplets rise (or sink) through the continuous phase in which it travels. When it reaches the interface where the two immiscible liquids meet, it may merge with its own phase. The separation efficiency of the two immiscible liquids is therefore determined by the ability of the droplets to rise or sink, and this strongly depends on the droplet sizes, as seen in (1.1). To improve on this efficiency under fluctuating operating conditions, it is important to understand how the droplet size distribution (DSD) acts, as controlling the separation effectively translates to controlling the DSD.

The generic population balance equation (PBE) contains terms that result in a non-closed form of the equation itself, and those terms are usually referred to as *kernels*. These kernels are system-dependent and some of the parameters that occur are experimentally determined. Therefore experiments have to be conducted, and the experimental data has to be compared to model calculations. The comparison results in an iterative procedure of adjusting parameters and recalculating errors between measurements and calculations, called parameter estimation. The procedure is crucial to reproduce and predict realistic

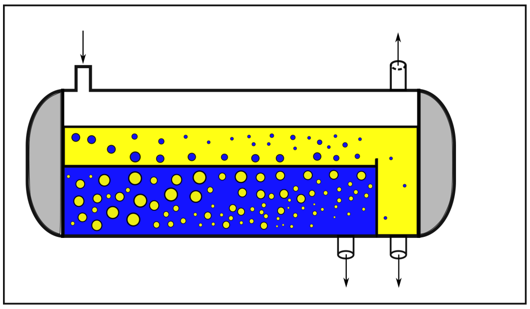


Figure 1.2: The gravity separator has an inlet in its upper left corner, where a mixture of oil, water and gas flows in. The liquids form two separate phases and are separated due to density differences and the weir that holds the water back. Gas leaves through the top right.

conditions, and for the model calculations to be useful.

1.2 Objective

As for the oil-in-water emulsion considered before, experimental data will be obtained for the system seen in Figure 1.3. The measurements will produce a DSD for each point in time, and a surface plot may be drawn for all measurements in that parallel. A model will be developed in the population balance framework and model regression techniques will be applied for recreating the experimental time evolution of the DSD. The techniques applied will be the sum of squared errors for each droplet size, the weighted sum of squared errors for each droplet size, and the sum of squared errors for the statistical mean for each temporal measurement. The goal of this work is to estimate the parameters previously mentioned under controlled circumstances to provide a model sufficiently good to predict inlet conditions for downstream units, such as the separator in Figure 1.2. If successful, this may remedy potentially propagating errors, increasing the overall performance of the plant.

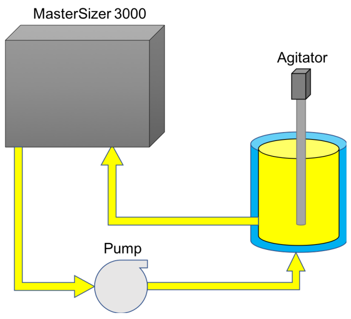


Figure 1.3: The experimental setup consists of a continuously stirred tank reactor, tubing to the measurement device, MasterSizer 3000, and tubing back to the CSTR through a pump.

If the model fitting is satisfactory, this also motivates for employing a similar model for pipe flow. The measurement devices under such conditions are typically expensive, and minimizing the cost of equipment may be a goal from the perspective of a process designer. The number of measurement devices is directly proportional to the cost, thus the overall cost would be minimized by, among other factors, minimizing the number of devices required. Therefore, the number of measurements required to arrive at the same parameter combinations will also be explored in this work, as it would determine the number of measurement devices required in the pipe.

Chapter 2

Mathematical Prerequisites

This chapter is dedicated to give a mathematical and theoretical background, as well as to provide all the tools needed to numerically solve differential equations and integrate functions over a domain Ω. The location of the grid points on which the function is evaluated, the derivative of the function and its integral can sometimes be decentralized in the sense that they are independent of each other. However, they can also be coupled in order to improve the numerical accuracy, as will be shown later on.

There exist many numerical schemes, and they all have their advantages and disadvantages. Some offer high local accuracy, but have high computational cost, some have low computational cost, but offer low local accuracy, and some have trouble for stiff systems in the sense that they cannot handle high gradients well. The numerical accuracy can be critical for systems where conservation properties must hold, while at other times it is essential that the computational complexity[1] is kept to a minimum. This all depends on the system at hand, and it would be wise to pick a numerical scheme that best suits the needs of the system.

Any system of equations that are to be solved should be presented on its linear form

$$\mathcal{L}f(x) = g(x) \quad \text{on } \Omega \tag{2.1a}$$
$$\mathcal{B}f(x) = f_\Gamma(x) \quad \text{on } \Gamma, \tag{2.1b}$$

where $\mathcal{L}$ is the linear operator, $f(x)$ is the function sought, $g(x)$ is the source term, Ω is the interior domain, $\mathcal{B}$ is the boundary operator, $f_\Gamma(x)$ is the function sought on the boundary and Γ is the boundary domain. The system has the form of any linear system of equations and can be solved on matrix form by left inversing

$$\mathbf{A}f = b \tag{2.2}$$

to obtain f. If the system of equations is non-linear, it can be linearized for an iterative procedure. For instance, this can be done by implementing Newton iteration [3, 4] or

[1]Computational complexity is here defined as the amount of resources required for running a program.

Picard iteration [3]. Newton linearization requires the solution of the linear system

$$\mathbf{M}[f^{n(m+1)} - f^{n(m)}] = -R, \tag{2.3}$$

where $f^{n(m)}$, is function n sought at iteration m, R is the residual and $\mathbf{M}$ is given by

$$\mathbf{M} = \mathbf{I} - \gamma \mathbf{J}. \tag{2.4}$$

$\mathbf{I}$ is the identity matrix, γ is the variable order step size and $\mathbf{J}$ is the Jacobian matrix. Newton iteration offers local q-quadratic convergence [5], however, it does not guarantee convergence. Picard iteration has a larger convergence radius, meaning that its initial value for iteration is less significant than for Newton iteration, however, it also has slower convergence, so there is a trade-off [6]. Picard iteration updates the function sought in the following manner

$$f_{n+1} = kf_{(n+1)'} + (1-k)f_n. \tag{2.5}$$

$f_{(n+1)'}$ is the temporary solution found after an iteration, and k is a value in the closed interval [0, 1] that can be viewed as a "chicken factor", in the sense that decreasing it will weight the old solution more than the new one. In other words, if the new solution takes too aggressive steps, this is constrained by lowering k. The value of f_n to be used in the next iteration is f_{n+1}.

2.1 Finite Differences and Simple Integration

Finite differences is one of the simplest schemes for differentiation and can be divided into three main categories: forward differences, backward differences and central differences. For a first order derivative and some step length h, both forward and backward differences have a local truncation error of $\mathcal{O}(h)$, while central differences has a local truncation error of $\mathcal{O}(h^2)$. This is easily verified by a simple Taylor expansion

$$f(x+h) = f(x) + h\frac{df(x)}{dx} + \text{H.O.T} \tag{2.6a}$$

$$f(x-h) = f(x) - h\frac{df(x)}{dx} + \text{H.O.T.} \tag{2.6b}$$

H.O.T are all higher order terms. Rearranging (2.6), replacing x and $x+h$ by x_i and x_{i+1} respectively, and truncating after the first derivative yields

$$\frac{df(x_i)}{dx} \approx \frac{f(x_{i+1}) - f(x_i)}{h}, \quad \text{forward} \tag{2.7a}$$

$$\frac{df(x_i)}{dx} \approx \frac{f(x_i) - f(x_{i-1})}{h}, \quad \text{backward,} \tag{2.7b}$$

where all terms of order two and higher were neglected. This gives the desired results of a local truncation error of $\mathcal{O}(h)$. Subtraction of (2.6a) and (2.6b) and truncating after the second derivative yields the central differences

$$\frac{df(x_i)}{dx} \approx \frac{f(x_{i+1}) - f(x_{i-1})}{2h}, \tag{2.8}$$

and a local truncation error of $\mathcal{O}(h^2)$ as claimed. Of course, the derivatives introduced in (2.7) and (2.8) can be used for any set of grid points $x = [x_1, x_2, ..., x_N]$, equidistant or not. However, for central differences the denominator would change slightly for non-equidistant grids. As seen, these derivatives only use information about the function sought in two neighboring points, see Figure 2.1. Finite difference schemes with higher order of truncation error also exist, but are not treated here. The second order derivative is deduced similarly to the first order ones and are given as

$$\frac{d^2 f(x_i)}{dx^2} \approx \frac{f(x_{i-1}) - 2f(x_i) + f(x_{i+1})}{h^2}, \quad \text{central} \tag{2.9a}$$

$$\frac{d^2 f(x_i)}{dx^2} \approx \frac{f(x_i) - 2f(x_{i+1}) + f(x_{i+2})}{h^2}, \quad \text{forward} \tag{2.9b}$$

$$\frac{d^2 f(x_i)}{dx^2} \approx \frac{f(x_{i-2}) - 2f(x_{i-1}) + f(x_i)}{h^2}, \quad \text{backward.} \tag{2.9c}$$

Applying the linear operator $\mathcal{L}$ from (2.1) on the central differences first derivative would yield the derivative matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\ -\frac{1}{h} & 0 & \frac{1}{h} & 0 & 0 & 0 & \dots & 0 \\ 0 & -\frac{1}{h} & 0 & \frac{1}{h} & 0 & 0 & \dots & 0 \\ 0 & 0 & -\frac{1}{h} & 0 & \frac{1}{h} & 0 & \dots & 0 \\ 0 & 0 & 0 & -\frac{1}{h} & 0 & \frac{1}{h} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & -\frac{1}{h} & 0 & \frac{1}{h} \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \end{bmatrix}, \tag{2.10}$$

while applying it to the second derivative would yield the derivative matrix

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \\ \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & 0 & 0 & \dots & 0 \\ 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & 0 & \dots & 0 \\ 0 & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & 0 & \dots & 0 \\ 0 & 0 & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} & \dots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{h^2} & -\frac{2}{h^2} & \frac{1}{h^2} \\ 0 & 0 & 0 & 0 & 0 & 0 & \dots & 0 \end{bmatrix}. \tag{2.11}$$

The boundary operator matrix $\mathcal{B}$ make sure that the boundary conditions (BCs) are satisfied. When the solution is specified at the boundary it is called a Dirichlet boundary condition, and when the derivative is specified at the boundary, it is called a Neumann boundary condition [7]. The boundary operator $\mathcal{B}$ for Dirichlet BCs is simply

$$\mathbf{B} = \begin{bmatrix} 1 & 0 & \dots & 0 \\ 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & 1 \end{bmatrix}. \tag{2.12}$$

The first row being different from 0 implies a left BC, i.e. at $x = x_0$, and the last row being different from 0 implies a right BC, i.e. at $x = x_N$. For the Neumann BCs, forward differences can be used at the left boundary and backwards differences may be used at the right boundary. Applying (2.7), the boundary operator applied to this kind of problem yields

$$\mathbf{B} = \begin{bmatrix} -\frac{1}{h} & \frac{1}{h} & 0 & \dots & 0 \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ 0 & 0 & \dots & -\frac{1}{h} & \frac{1}{h} \end{bmatrix}. \tag{2.13}$$

The right hand side (source) of the problem, $g(x)$, will specify the boundaries in its top and bottom element.

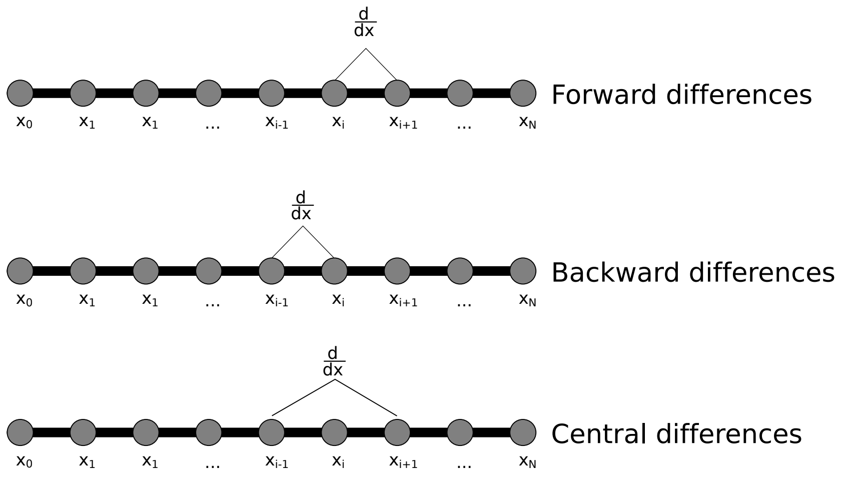

d/dx
Forward differences
x0 x1 x1 ... xi-1 xi xi+1 ... xN
d/dx
Backward differences
x0 x1 x1 ... xi-1 xi xi+1 ... xN
d/dx
Central differences
x0 x1 x1 ... xi-1 xi xi+1 ... xN

Figure 2.1: First and second order finite differences numerical schemes. Derivatives use information from two neighboring points at most.

In numerical mathematics, the term numerical integration consists of a collection of ways to evaluate a finite integral. Numerical integration is often referred to as numerical quadrature, or quadrature for short. One of the earliest references to the term was by David Gibb [8], and it has since received a lot of attention because of the growing need to evaluate complex integrals precisely. The simplest techniques for numerical integration is probably rectangle approximations, trapezoidal approximations and Simpson's method.

Consider the integral from left boundary a to right boundary b

$$I = \int_a^b f(x)dx, \tag{2.14}$$

and let $\mathbb{P}$ be the set of partitions of the integral above

$$\mathbb{P} = \{[x_0, x_1], [x_1, x_2], ..., [x_{N-2}, x_{N-1}], [x_{N-1}, x_N]\}. \tag{2.15}$$

For $a = x_0 < x_1 < ... < x_{N-1} < x_N = b$, the integral, I, can be approximated by applying the finite Riemann sum

$$I \approx S = \sum_{i=1}^{N} f(x_i)\Delta x_i, \tag{2.16}$$

where $\Delta x_i = x_i - x_{i-1}$. The Riemann sum becomes exact when $N \to \infty$, that is

$$I = \lim_{N\to\infty} \sum_{i=1}^{N} f(x_i)\Delta x_i, \tag{2.17}$$

but from a computational perspective this is not a result that can be used. As (2.16) suggests, the integral is approximated by rectangles of width $x_i - x_{i-1}$ and height $f(x_i)$, see Figure 2.2. For small N, the approximation can be questionable.

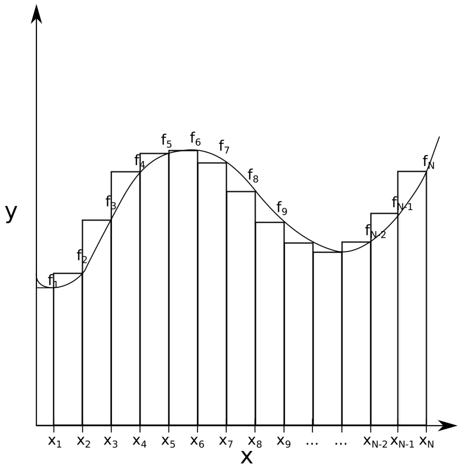


Figure 2.2: An integral approximation by a finite Riemann sum of length N. The area is approximated by rectangles, and for small N, the approximation can be questionable.

As seen above, the approximation by rectangles can be coarse when N is small. This can be remedied by using trapezoids instead of rectangles. The area of a trapezoid is

$$A = \frac{b+a}{2}h, \tag{2.18}$$

where b and a are lengths and h is the height. Applied to a function $f(x)$ and using the same indexing procedure as earlier, this translates to [9]

$$I \approx \sum_{i=1}^{N} A_i = \frac{f_i + f_{i-1}}{2}\Delta x_i, \tag{2.19}$$

where $\Delta x_i = x_i - x_{i-1}$. The trapezoidal approximation usually gives a smaller error than the rectangle approximation. For an illustration of the trapezoidal approximation of the quartic polynomial example used previously, see Figure 2.3.

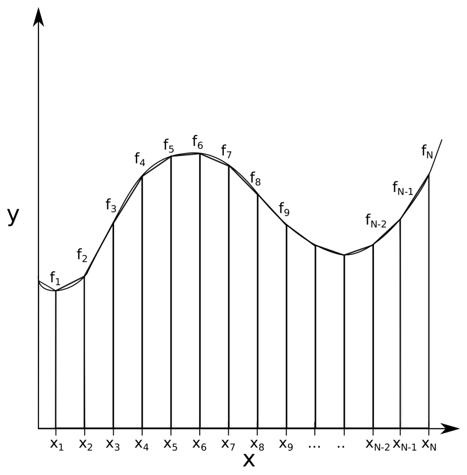


Figure 2.3: An integral approximation by N trapezoids. The error between the quadrature and the analytical solution is smaller here than for the Riemann sum.

If the error is still unsatisfactory, a quadratic polynomial can be constructed in order to interpolate between the three points (x_i, f_i), (x_{i+1}, f_{i+1}) and (x_{i+2}, f_{i+2}), and the area under this polynomial can be evaluated. The sum of these areas approximate the original function integral. Each polynomial constructed is on the form

$$f^q(x) = a + b(x - x_i) + c(x - x_{i+1})(x - x_i), \tag{2.20}$$

and for the polynomial to pass through the points claimed, it is required that

$$a = f_i \tag{2.21a}$$

$$b = \frac{f_{i+1} - f_i}{x_{i+1} - x_i} \tag{2.21b}$$

$$c = \frac{1}{x_{i+2} - x_{i+1}} \left(\frac{f_{i+2} - f_i}{x_{i+2} - x_i} - \frac{f_{i+1} - f_i}{x_{i+1} - x_i} \right). \tag{2.21c}$$

Inserting the coefficients from (2.21) into (2.20) yields an interpolating polynomial of degree two

$$f^q(x) = f_i + (f_{i+1} - f_i)\frac{x - x_i}{x_{i+1} - x_i} + \left(\frac{f_{i+2} - f_i}{x_{i+2} - x_i} - \frac{f_{i+1} - f_i}{x_{i+1} - x_i} \right) \frac{(x - x_{i+1})(x - x_i)}{x_{i+2} - x_{i+1}}. \tag{2.22}$$

In the case of an equidistant grid, see Figure 2.4, all points are evenly spaced, i.e. $h = \Delta x_i = x_i - x_{i-1}, i \in [1, N]$. For the quadratic interpolation, the area under each inter-

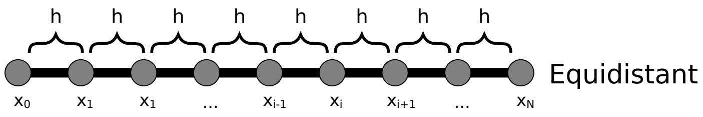


Figure 2.4: An equidistant grid in the independent coordinate. The grid spacing is $h = \Delta x_i = x_i - x_{i-1}, i \in [1, N]$.

polant is its integral

$$\begin{aligned}
\int_{x=x_i}^{x=x_{i+2}} f^q(x)dx &= \int_{x=x_i}^{x=x_i+2h} f^q(x)dx \\
&= \int_{u=0}^{u=2h} \Big(f_i + \frac{f_{i+1}-f_i}{h}u \\
&+ \left(\frac{f_{i+2}-f_i}{2h} - \frac{f_{i+1}-f_i}{h}\right)\frac{u(u-h)}{h}\Big)du \\
&= \int_{u=0}^{u=2h} \Big(f_i + \frac{f_{i+1}-f_i}{h}u \\
&+ \frac{1}{2h^2}\left(f_i - 2f_{i+1} + f_{i+2}u(u-h)\right)\Big)du \\
&= \frac{h}{3}\left(f_i + 4f_{i+1} + f_{i+2}\right),
\end{aligned} \tag{2.23}$$

where $u = x - x_0$ has been used as substitution. Repeating this integration for several subdomains $\omega \subset \Omega$, results in the total approximation

$$I \approx \frac{h}{3}\left(f_0 + 4f_1 + 2f_2 + 4f_3 + ... + 2f_{N-2} + 4f_{N-1} + f_N\right) \tag{2.24}$$

over Ω. The result is called Simpson's $1/3$ rule, named after the English mathematician Thomas Simpson. The formula is precise to an order of h^4, while that of trapezoids is precise to an order of h^3 [9]. This can be proved by a simple Taylor series expansion followed by integration. For oscillatory functions, or functions that are non-smooth over the interval, Simpson's rule can yield poor results. Even though Simpson's rule is relatively high order, high order only translates to high accuracy when the function is smooth, or when its polynomial expansion approximates the function well [10]. Thus it may give poor results if the polynomial approximation is unsatisfactory.

2.2 Weighted Residual Methods: Orthogonal Collocation

The weighted residual methods (WRM) is a family of numerical methods that focuses on finding the function value at fixed nodal points called collocation points. Therefore this family of methods is similar to that of the family of discrete methods. Consider (2.1) on its residual form [10]

$$\mathcal{R}_\Omega = \mathcal{L}f(x) - g(x) \quad \text{on } \Omega \tag{2.25a}$$

$$\mathcal{R}_\Gamma = \mathcal{B}f(x) - f_\Gamma(x) \quad \text{on } \Gamma. \tag{2.25b}$$

With respect to a certain weight function, ω, the methods try to drive the residual, $\mathcal{R}$, to zero over the entire domain, that is

$$\int_{\Omega} \omega_{\Omega,I} \mathcal{R}_{\Omega} d\Omega + \int_{\Gamma} \omega_{\Gamma,I} \mathcal{R}_{\Gamma} d\Gamma = 0, \quad \forall I = 0, 1, ..., N, \tag{2.26}$$

where N is the total number of collocation points. The WRM are all based on having represented the function, f, as a trial function expansion. This trial function expansion is a formulation of the sought function f as a series of trial functions $\phi_i(x)$ multiplied by basis coefficients α_i:

$$f(x) \approx f^N(x) = \sum_{i=0}^{N} \alpha_i \phi_i(x). \tag{2.27}$$

By choosing the trial functions as orthogonal polynomials such as Lagrange interpolating polynomials [11]

$$\ell_i(x) = \prod_{\substack{j=0 \\ j \neq i}}^{N} \frac{x - x_j}{x_i - x_j}, \tag{2.28}$$

the basis coefficients can be chosen as the function value, $f(x_i)$, itself at the collocation points, x_i. This is due to the property of Lagrange polynomials being defined as

$$\ell_j(x_i) = \begin{cases} 1, & \text{if } i = j \\ 0, & \text{if } i \neq j \end{cases}. \tag{2.29}$$

The Lagrange interpolating polynomials are all of the same degree, and for degree N, there are $N + 1$ polynomials that interpolate f [11]. The five polynomials of degree four are depicted in Figure 2.5. The numerical derivative of the function f can be found by using the polynomial expansion

$$\frac{df(x)}{dx} \approx \frac{d}{dx} \sum_{i=0}^{N} f(x_i) \ell_i(x) = \sum_{i=0}^{N} f(x_i) \frac{d\ell_i(x)}{dx} \tag{2.30}$$

The resulting derivative matrix is represented by

$$\mathbf{A} = \begin{bmatrix} 0 & 0 & 0 & \dots & 0 \\ \ell_0'(x_1) & \ell_1'(x_1) & \ell_2'(x_1) & \dots & \ell_N'(x_1) \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \ell_0'(x_{N-1}) & \ell_1'(x_{N-1}) & \ell_2'(x_{N-1}) & \dots & \ell_N'(x_{N-1}) \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix}, \tag{2.31}$$

and as seen it uses information from all of the derivatives at all grid points, see Figure 2.6. The matrix $\mathbf{A}$ is square and is comparable to those from finite differences in (2.10) and (2.11). The boundary derivative may be specified as in (2.13):

$$\mathbf{B} = \begin{bmatrix} \ell_0'(x_0) & \ell_1'(x_0) & \ell_2'(x_0) & \dots & \ell_N'(x_0) \\ 0 & 0 & 0 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ \ell_0'(x_N) & \ell_1'(x_N) & \ell_2'(x_N) & \dots & \ell_N'(x_N) \end{bmatrix}. \tag{2.32}$$

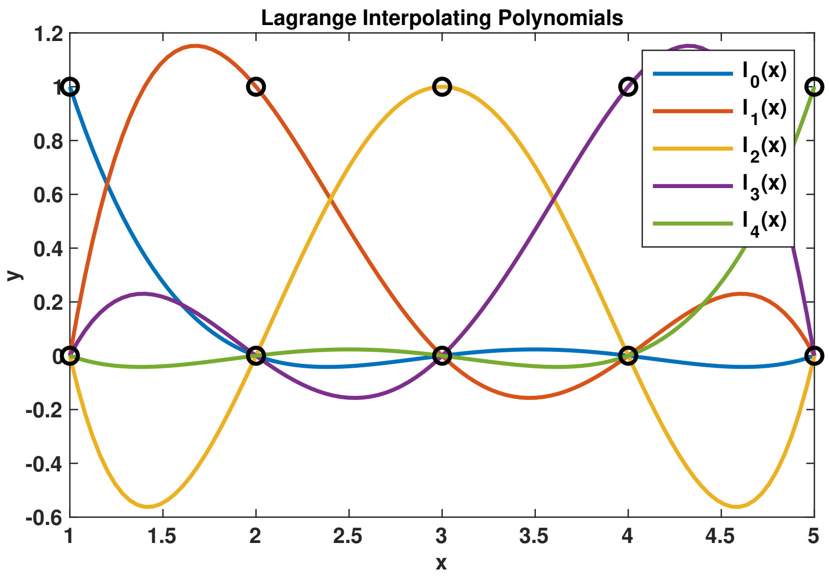


Figure 2.5: Lagrange interpolating polynomials. All polynomials take value of 0 in all nodes except when the node index and the polynomial index are equal. Then the function takes the value of 1.

The Dirichlet boundary conditional matrix $\mathbf{B}$ is the same as in (2.12).

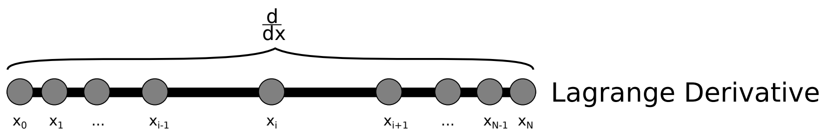


Figure 2.6: As the figure shows, all collocation points in the grid contribute to the derivative at $x = x_i$.

The selection of the collocation points, x_i, in the domain Ω are normally taken as the roots of Jacobi polynomials [12]. If this is the case, there are three different types of grids: Gauss, Gauss-Lobatto and Gauss-Radau. As seen in Figure 2.7, Gauss is without endpoints, Gauss-Lobatto is with both endpoints and Gauss-Radau is with one endpoint. The Jacobi polynomials are orthogonal, meaning

$$\int_a^b W(x)p_i(x)p_j(x)dx = 0, \quad i \neq j. \tag{2.33}$$

The weight function is dependent on which polynomials are chosen, but for the Jacobi polynomials

$$W(x) = (1-x)^{\alpha}(1+x)^{\beta}, \tag{2.34}$$

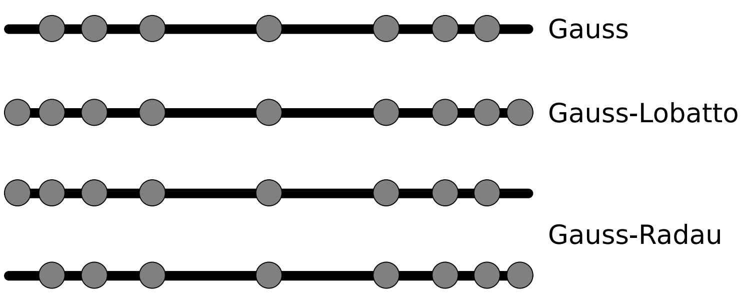


Figure 2.7: Gauss, Gauss-Lobatto and Gauss-Radau collocation points. Gauss is without endpoints, Gauss-Lobatto is with both endpoints, and Gauss-Radau is with one endpoint.

where α and β can be viewed as parameters for shifting the collocation points towards one or the other end of the domain. All orthogonal polynomials satisfy a three term recurrence relationship

$$\begin{aligned} p_{-1}(x) &= 0 \\ p_0(x) &= 1 \\ p_{i+1}(x) &= (a_i x + b_i)p_i(x) - c_i p_{i-1}(x), \quad -1 < i \in \mathbb{Z}. \end{aligned} \tag{2.35}$$

The coefficients a_i, b_i and c_i are given by

$$a_i = \frac{(2i+\alpha+\beta+1)(2i+\alpha+\beta+2)}{2(i+1)(i+\alpha+\beta+1)} \tag{2.36a}$$

$$b_i = \frac{(2i+\alpha+\beta+1)(\alpha^2-\beta^2)}{2(i+1)(i+\alpha+\beta+1)(2i+\alpha+\beta)} \tag{2.36b}$$

$$c_i = \frac{(i+\alpha)(i+\beta)(2i+\alpha+\beta+2)}{(i+1)(i+\alpha+\beta+1)(2i+\alpha+\beta)}. \tag{2.36c}$$

If any of the denominators in (2.36) take the value of zero, the coefficient itself takes the value of zero. For $\alpha = \beta$, the polynomials are called Gegenbauer polynomials, and among the most known are Chebyshev polynomials ($\alpha = \beta = -1/2$) and Legendre polynomials ($\alpha = \beta = 0$). For the latter, the weight of (2.34) takes the form

$$W(x) = 1, \quad -1 < x < 1, \tag{2.37}$$

and the recurrence coefficients simplify to

$$a_i = \frac{2i+1}{i+1} \tag{2.38a}$$

$$b_i = 0 \tag{2.38b}$$

$$c_i = \frac{i}{i+1}. \tag{2.38c}$$

Representing the coefficients from (2.36) on matrix notation

$$x\begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} = \begin{bmatrix} -\frac{b_1}{a_1} & \frac{1}{a_1} & 0 & \dots & 0 \\ \frac{c_2}{a_2} & -\frac{b_2}{a_2} & \frac{1}{a_2} & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \frac{c_{N-1}}{a_{N-1}} & -\frac{b_{N-1}}{a_{N-1}} & \frac{1}{a_{N-1}} \\ 0 & 0 & \dots & \frac{c_N}{a_N} & -\frac{b_N}{a_N} \end{bmatrix} \begin{bmatrix} p_0 \\ p_1 \\ \vdots \\ p_{N-2} \\ p_{N-1} \end{bmatrix} + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ \frac{p_N}{a_N} \end{bmatrix} \tag{2.39}$$

and evaluating $p_N(x)$ at its roots yields the eigenvalue-form of (2.39)

$$xp(x) = \mathbf{T}p(x), \tag{2.40}$$

where p is the vector of Jacobi polynomials and $\mathbf{T}$ is the tridiagonal matrix in (2.39). As noted, the last polynomial in the chain, p_N, is evaluated at its roots, which is why the last term of (2.39) vanished ($p_N(x_i) = 0 \ \forall i \in \{0, 1, ..., N\}$). This is only true if $\lambda_i p(\lambda_i) = \mathbf{T}p(\lambda_i)$, and consequently the problem of finding the collocation points reduces to finding the eigenvalues of $\mathbf{T}$. This procedure is called the Golub-Welsch algorithm, and it is faster than root-finding algorithms such as Newton-Raphson iteration [10, 13]. For a quicker numerical procedure, $\mathbf{T}$ may be transformed into the symmetric matrix

$$\mathbf{J} = \mathbf{D}\mathbf{T}\mathbf{D}^{-1} = \begin{bmatrix} -\frac{b_1}{a_1} & \left(\frac{c_2}{a_1 a_2}\right)^{1/2} & 0 & \dots & 0 \\ \left(\frac{c_2}{a_1 a_2}\right)^{1/2} & -\frac{b_2}{a_2} & \left(\frac{c_3}{a_2 a_3}\right)^{1/2} & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & \vdots \\ 0 & \dots & \left(\frac{c_{N-1}}{a_{N-2} a_{N-1}}\right)^{1/2} & -\frac{b_{N-1}}{a_{N-1}} & \left(\frac{c_N}{a_{N-1} a_N}\right)^{1/2} \\ 0 & 0 & \dots & \left(\frac{c_N}{a_{N-1} a_N}\right)^{1/2} & -\frac{b_N}{a_N} \end{bmatrix}. \tag{2.41}$$

As seen, the matrix is symmetric around its diagonal, and it can thus be simplified as such. Adopting the notation of Jakobsen [10],

$$\mathbf{J} = \begin{bmatrix} \tau_1 & \gamma_1 & 0 & \dots & 0 \\ \gamma_1 & \tau_2 & \gamma_2 & \dots & 0 \\ \vdots & \ddots & \ddots & \ddots & 0 \\ 0 & \dots & \gamma_{N-2} & \tau_{N-1} & \gamma_{N-1} \\ 0 & 0 & \dots & \gamma_{N-1} & \tau_N \end{bmatrix}, \tag{2.42}$$

where $\tau_i = -\frac{b_i}{a_i}$ and $\gamma_i = \left(\frac{c_{i+1}}{a_i a_{i+1}}\right)^{1/2}$. If eigenvalues, λ_i, and eigenvectors, v_i, of $\mathbf{T}$ are found from (2.40) and compared to those of (2.41), they are identical. Thus, the transformation in (2.41) has left the values of interest as they were. The integral weights that will allow for numerical quadratures are found by normalizing each eigenvector and multiplying its first element squared with the integral of the corresponding weight function, that is

$$w_i = \mu_0 v_{i,1}^2, \tag{2.43}$$

where

$$\mu_0 = \int_a^b W(x)dx. \tag{2.44}$$

For the Legendre polynomials, μ_0 takes the value of 2, since $W(x) = 1$, $a = -1$ and $b = 1$. The weights w_i are used in the evaluation of the integral of $f(x)$ [14]

$$\int_{-1}^{1} f(x)dx = \int_{-1}^{1} W(x)f(x)dx = \sum_{i=1}^{N} w_i f(x_i). \tag{2.45}$$

The above is valid for the Gauss quadrature, i.e. without endpoints. For boundary value problems (BVPs), Gauss-Lobatto grids should be used instead. Then $x_0 = -1$ and $x_N = 1$ must be enforced, and the matrix $\mathbf{J}$ must be modified. To enforce these criteria, the modified matrix $\tilde{\mathbf{J}}$ must enforce $\lambda_0 = -1$ and $\lambda_N = 1$. Thus a polynomial $p_{N+1}(x)$ is constructed so that

$$p_{N+1}(x = -1) = p_{N+1}(x = 1) = 0 \tag{2.46}$$

is satisfied. The recurrence relationship (2.35) can be written for this system as

$$\gamma_{N+1} p_{N+1}(x) = (x - \tau_{N+1}) p_N(x) - \gamma_N p_{N-1}(x). \tag{2.47}$$

This is seen by inspecting the last row of the matrix $\mathbf{J}$. The enforcing condition in (2.46) can be applied to (2.47) (note that the left hand side in the latter of the equations then equate to zero):

$$\tau_{N+1} p_N(x = -1) + \gamma_N p_{N-1}(x = -1) = -1 p_N(x = -1) \tag{2.48a}$$
$$\tau_{N+1} p_N(x = +1) + \gamma_N p_{N-1}(x = +1) = +1 p_N(x = +1). \tag{2.48b}$$

This takes the form of (2.40)

$$xp(x) = \mathbf{J}p(x) + \gamma_N p_N(x) e_N, \tag{2.49}$$

where e_N is the unit vector $(0, 0, ..., 1)$ of size N. This means that some vector η and μ can be set so that the following is satisfied

$$[\mathbf{J} - (-1)\mathbf{I}]\eta = e_N \tag{2.50a}$$
$$[\mathbf{J} - (+1)\mathbf{I}]\mu = e_N, \tag{2.50b}$$

which means that

$$\eta_i = -\frac{1}{\gamma_N} \frac{p_{i-1}(x = -1)}{p_N(x = -1)} \tag{2.51a}$$
$$\mu_i = -\frac{1}{\gamma_N} \frac{p_{i-1}(x = +1)}{p_N(x = +1)}. \tag{2.51b}$$

To find the complete modified matrix, $\tilde{\mathbf{J}}$, τ_{N+1} and γ_N must be found. For this, the equation set

$$\tau_{N+1} - \eta_N \gamma_N^2 = -1 \tag{2.52a}$$
$$\tau_{N+1} - \mu_N \gamma_N^2 = +1 \tag{2.52b}$$

must be solved. The eigenvalues and eigenvectors can then be computed from

$$\tilde{\mathbf{J}} = \begin{bmatrix} \mathbf{J} & \gamma_N e_N \\ \gamma_N e_N^T & \tau_{N+1} \end{bmatrix} \tag{2.53}$$

and the collocation points and quadrature weights can be computed as for the Gauss grid.

Now that both numerical integration by Gauss quadratures and numerical differentiation by Lagrange derivatives have been treated, once again consider (2.26). For the orthogonal collocation method the weight function, ω_I, is chosen as a Dirac delta $\delta(x - x_I)$, forcing the residual to zero at the collocation points. Other popular weighted residual methods include the least squares method (LSM), where the weight function is taken as the derivative of the residual with respect to its function value, $\omega_I = \frac{dR}{df_I}$. The LSM will not be treated here, but instead the weight function for the orthogonal collocation (OC) method is applied to (2.26)

$$\begin{aligned} \int_\Omega \mathcal{R}_\Omega(x, f_1, f_2, ..., f_N)\delta(x - x_I)d\Omega &= \mathcal{R}_\Omega(x, f_1, f_2, ..., f_N)|_{x=x_I} \\ &= \sum_{i=0}^{N} \mathcal{L} f_i \ell_i(x_I) - g(x_I) \\ &= 0. \end{aligned} \tag{2.54}$$

The definition of the residual was used and inserted above. As (2.54) states, polynomial ℓ_i is evaluated at all the collocation points x_I, resulting in a matrix structure. Also, the function values sought, f_i, are independent of the linear operator, $\mathcal{L}$, so the prepared system of equations can be written in accordance with (2.2) as

$$\begin{aligned} [\mathbf{A}]_{ij} &= \mathcal{L}\ell_j(x_i) \\ [f]_i &= f(x_i) \\ [b_\Omega]_i &= g(x_i). \end{aligned} \tag{2.55}$$

To enforce appropriate BCs, the matrix $\mathbf{A}$ and the vector b_Ω are modified in their first and last row. These rows are set to 0, and the boundary matrix $\mathbf{B}$ is constructed

$$\begin{aligned} [\mathbf{B}]_{ij} &= \begin{cases} \left.\begin{array}{ll} 1, & i = j = 0, N \\ 0, & i = 0, j \neq 0 \\ 0, & i = N, j \neq N \end{array}\right\} \text{Dirichlet BC} \\ \left.\begin{array}{ll} \ell_j'(x_0), & i = 0 \\ \ell_j'(x_N), & i = N \\ 0, & i \neq 0, N \end{array}\right\} \text{Neumann BC} \end{cases} \\ [b_\Gamma]_i &= \begin{cases} \psi_0, & i = 0 \\ \psi_N, & i = N \\ 0, & i \neq 0, N \end{cases} \end{aligned} \tag{2.56}$$

Obviously a mix of Dirichlet and Neumann BCs may also be specified if necessary. To construct the entire problem

$$\mathbf{A}^{\mathrm{p}} = \mathbf{A}' + \mathbf{B} \tag{2.57a}$$

$$[f]_i = f(x_i) \tag{2.57b}$$

$$b^{\mathrm{p}} = b'_{\Omega} + b_{\Gamma}, \tag{2.57c}$$

where $\mathbf{A}'$ and b'_{Ω} are the modified matrix $\mathbf{A}$ and vector b_{Ω} respectively. The problem is solved by left inversing $\mathbf{A}^{\mathrm{p}}$. Should the problem be non-linear, Newton iteration or Picard iteration can be applied as previously discussed.

2.3 Other Frequently Used Methods

The numerical methods frequently employed are normally split in two: those that focus on finding the function itself, and those that focus on finding its statistical moments of order j defined by [15]

$$\mu_j = \int_0^{\infty} x^j f(x)dx, \tag{2.58}$$

where $f(x)$ is the function whose moments are to be determined. The different methods that can be employed are optimized for its use, and a lengthy description of them is not the focus of this work, and will thus not be treated here. The different numerical methods will, however, be presented to the reader for a broader list of options.

2.3.1 Finding the Function

For the methods that focus on finding the function itself, the WRM has already been explained in detail. The sectional methods may be called zero order methods. The methods lump a portion of the independent coordinate together in a single cell (section) and this cell is represented by a zero-order polynomial. Consequently, each cell takes the form of a histogram or a bar diagram, and the function is represented by a set of bar diagrams [16, 17]. Each partition is represented by

$$\Delta_i = [x_{i+1/2} - x_{i-1/2}], \tag{2.59}$$

and the middle point of each partition, x_i, is called a pivot or grid point. The independent coordinate, x, is considered fixed throughout the entire pivot. The pivots can move throughout the time horizon or they can be fixed (stationary), giving rise to the names moving pivot [18] and fixed pivot methods [19], respectively. Each cell may also be approximated by a low order polynomial. The method of zero-order polynomials may resemble the form of Riemann sums depicted in Figure 2.2, whereas the low-order polynomial may resemble the form of trapezoids (Figure 2.3), Simpson's rule etc. For a density distribution, $f(x,t)$, the density, $F(t)$, can be found in each pivot by integrating the distribution from its left to its right boundary

$$F(t) = \int_{x_{i-1/2}}^{x_{i+1/2}} f(x,t)dx. \tag{2.60}$$

Among the most famous sectional methods is the fixed pivot technique developed by Kumar and Ramkrishna due to its generality and robustness, as shown by the studies of Kumar and Warnecke [20].

The finite volume method (FVM) is a multi-dimensional version of the finite differences scheme discussed previously. The idea is locate a control volume around the grid points and evaluate fluxes coming in and out of this finite volume. This method describes the net change in the function sought over finite volume elements, just as a partial differential equation (PDE) describes the net change over an infinitesimally small volume element [21]. Johnson [22] described the method for two dimensions with the x-y grid depicted in Figure 2.8. Each cell (control volume) is shown as a dashed square, the grid points are the filled circles and the unfilled circles are the boundary points containing the boundary condition. As shown, the grid points are placed in the center of the control volume, and they communicate with four neighboring grid points. For an arbitrary point, P, the communication points are denoted N, W, E and S. These names denote the cardinal directions *north*, *west*, *east* and *south*, respectively. For a three-dimensional FVM scheme, the point, P, communicates with the grid point on top of it, T, and the grid point beneath it, B, as well. The symbols denote *top* and *bottom*, respectively. Note that the FVM is also a sectional method in the sense that the cells are sections of the full domain.

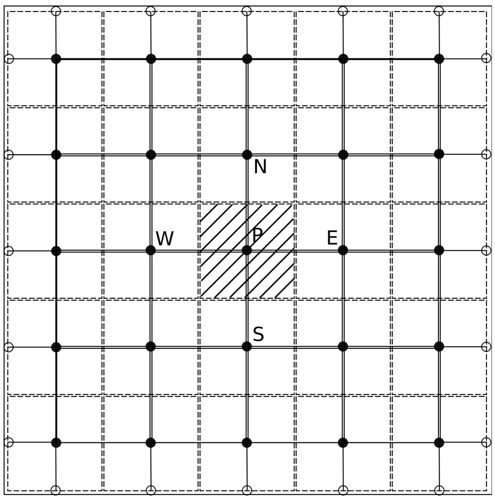


Figure 2.8: A finite volume numerical scheme. Each cell is marked by dashed lines, and each grid point itself is modeled by its fluxes of some quantity in and out of the cell. The filled circles indicate grid points and the unfilled ones are boundary points.

2.3.2 The Methods of Moments

For processes where the function itself is not of particular importance, the statistical moments defined in (2.58) might be sufficient. Consider a process only relying on the lower order moments such as the statistical mean and the standard deviation defined by the first and second order moment, respectively. Then it is not efficient to first find the function itself and then calculate the mean and standard deviation, but it may be computationally faster to do a moment transformation on the function and find the moments themselves instead. Should the function be needed at some point, it may be reconstructed by the method of maximum entropy, which reconstructs the function by equally weighting all possible distributions. As a result, the function is reconstructed in a least biased way, i.e. the reconstructed function has the highest probability of resembling the real distribution. See Cover [23] for a detailed explanation.

The methods of moments (MOM) are a family of numerical approaches on how these equations can be solved. Without going into detail on each of them, some of them will be listed here for comparison. The standard method of moments (SMOM) developed by Randolph and Larson [24] and Hulburt and Katz [25] solves the time-dependent moment transformation of the function sought for lower order moments. The method expands the j-th moment and expresses it in terms of its lower order moments, k, where $k \leq j$. The set of equations must be entirely independent of the function and may only be expressed as a function of these k-th order moments.

McGraw [26] proposed a method called the quadrature method of moments (QMOM) that has received much more attention in population balance modeling. This method is widely used, as it is both computationally efficient and has a broad range of applicable problems. The method includes the approximation of f by a series expansion of Dirac delta functions

$$f(x) \approx \sum_{i=1}^{N} w_i(x)\delta(x - x_i), \tag{2.61}$$

where the weights w_i are determined from specialized algorithms consisting of lower order moments. The weights are then applied and the moments are expressed by

$$\mu_j = \sum_{i=1}^{N} w_i(x)x_i^j. \tag{2.62}$$

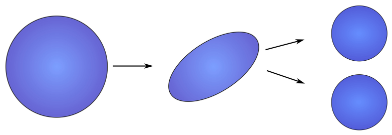
Another closely related MOM is the direct quadrature method of moments (DQMOM) developed by Marchisio and Fox [27] and Fan et al. [28], where the same procedure as McGraw suggested is applied, only for a product sum of Dirac deltas. Surely there are other MOMs as well, but to avoid a lengthy discussion of these, this section is truncated, and the reader is redirected to Jakobsen [10].

Chapter 3

Population Balance Modeling

As noted in Chapter 1, population balance modeling (PBM) has grown as a field of research, possibly by the influence of Ramkrishna [1]. He employed a local continuum mechanical framework in order to establish an approach of population balance modeling (PBM) frequently adopted. His PBE established a relation for countable particles and their temporal (time) evolution. The particles also depend on their location in physical space and their abstract property space. The former is defined in three dimensions, $\mathbf{r} = (r_1, r_2, r_3)$, and the coordinate system employed (cartesian, cylindrical or spherical) depend on the geometry of the system considered. The latter of them is defined by some inherently possessed quantities, $\mathbf{x} = (x_1, x_2, ..., x_m)$, where m is the number of quantities. The two spaces, $\mathbf{r}$ and $\mathbf{x}$, were referred to by Hulburt and Katz [25] as *external* and *internal* coordinates, respectively, and together they define the *state* of the particle. Furthermore, the particles are considered to depend on the environment variables (continuous phase for fluid-liquid dispersions) defined by the vector $\mathbf{Y} = (y_1, y_2, ..., y_n)$, where n is the number of variables. Common examples of such variables include densities, pressures etc.

Dispersed in a continuous phase, the particles may undergo breakage, coalescence and aggregation/agglomeration/coagulation, all of which are considered birth and death processes. The latter three are not discussed here, but the interested reader is redirected to papers that treat these topics [29]. Breakage is a phenomenon where a mother particle breaks up (death process) into two or more fragments called daughter particles (birth process). The phenomenon preserves mass, so that the mass of the mother particle equates to the sum of masses for each daughter particle. Thus the mass of all mother particles breaking at time t equates to the mass of all daughter particles born at time t. Considering coalescence, two or more particles collide at some frequency, and at some probability they merge (death process) into one bigger particle (birth process). The same principle as for breakage applies here; the mass of all particles dying due to coalescence at time t equates to the mass of all particles born due to coalescence at time t. The mechanisms of breakage and coalescence are depicted in Figure 3.1a and Figure 3.1b, respectively. It is important to note that a birth process cannot take place without having a corresponding death process take place.


(a) The illustration shows the phenomenon called breakage, where a particle is torn apart by turbulent stresses caused by velocity fluctuations.

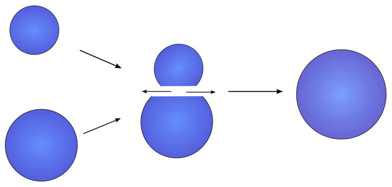

(b) The illustration shows the phenomenon called coalescence, in which two colliding particles create a thin film which drains over time and eventually ruptures.

Figure 3.1: The figure depicts the phenomenon of (a) breakage and (b) coalescence.

With these definitions, and with some simplified notation, Ramkrishna defined the population balance as follows

$$\begin{aligned}&\frac{\partial f_n(\mathbf{x},\mathbf{r},t)}{\partial t} + \nabla_{\mathbf{x}} \cdot [\mathbf{v}_{\mathbf{x}}(\mathbf{x},\mathbf{r},\mathbf{Y},t) f_n(\mathbf{x},\mathbf{r},t)] + \nabla_{\mathbf{r}} \cdot [\mathbf{v}_{\mathbf{r}}(\mathbf{x},\mathbf{r},\mathbf{Y},t) f_n(\mathbf{x},\mathbf{r},t)] \\ &= S(\mathbf{x},\mathbf{r},\mathbf{Y},t),\end{aligned} \tag{3.1}$$

where the dependent variables f_n, $\mathbf{v}_{\mathbf{x}}$, $\mathbf{v}_{\mathbf{r}}$ and S are the number density distribution function, the phase velocity vector, the space velocity vector and the source function, respectively. The physical interpretation of the second term from the left corresponds to growth due to motion in abstract property space, and the third term corresponds to growth due to motion in the physical space.

The choice of internal coordinate is important, and the options available are manifold. For instance, for a fermenting process, yeast cells may live in a solution containing a substrate that the yeast cells consume in order to produce a chemical compound. If the product inhibits the reaction, or for some other reason the yeast cells must be separated from this solution, a microorganism flotation technique [30] may be applied in order to recover the cells from the medium. This technique involves bubbling air into the reactor, and when the yeast collides with the air bubbles, they stick to the bubbles and rise to the surface. When they reach the surface, they form a froth, which is skimmed off, successfully recovering the microorganism. For this example the internal coordinate may be chosen as the number of yeast cells attached to an air bubble.

Usually, the choice of internal coordinate is less intricate than for the system above. The diameter, d, of the particles is often picked as internal coordinate, because it is easily measured, and it makes the PBE applicable to systems dependent on mass- or heat transfer, where surface area is crucial. It also gives the PBE a simple physical meaning. If the diameter, d, is chosen as internal coordinate, the PBE represents the number of particles of a particular size, d, at a particular position in space, $\mathbf{r}$, at a particular time, t. There are no bounds for how many internal coordinates that may be applied to the PBE, and systems with one or two internal coordinates are referred to as univariate (or monovariate) or bivariate systems, respectively. For more than two internal coordinates, the system is referred to as polyvariate.

Neglecting agglomeration/aggregation/coagulation, the source term of (3.1) consists of breakage and coalescence

$$S(\mathbf{x},\mathbf{r},\mathbf{Y},t) = B_B(\mathbf{x},\mathbf{r},\mathbf{Y},t) - D_B(\mathbf{x},\mathbf{r},\mathbf{Y},t) + B_C(\mathbf{x},\mathbf{r},\mathbf{Y},t) - D_C(\mathbf{x}.\mathbf{r},\mathbf{Y},t), \quad (3.2)$$

The subscript denotes which phenomenon the process is related to, and the base letter refers to birth or death. In other words, B_B and D_B are birth and death due to breakage, and B_C and D_C are birth and death due to coalescence, respectively.

3.1 Closure Equations

The population balance equation presented in (3.1) and (3.2) have some unclosed terms, namely B_B, D_B, B_C and D_C. For a given phenomenon, the related birth and death processes are modeled by the same constitutive equations, or *kernels*. However, how these are modeled is a field of research on its own, and many proposals have been made [31, 32, 33, 34].

Concerning breakage, the modeling approaches are trifold: those based on reaction-kinetic ideas [35], those based on the turbulent flow conditions [36], and those based on kinetic ideas only [37]. Martínez-Bazán et al. [37] argued that turbulent stresses, σ_t, caused by fluctuations in the velocity field will deform the particle considered. Whenever the turbulent stress becomes greater than the surface restorative forces (surface tension), σ_s, the particle breaks. This concept is based only on kinematics. Thus, the *breakage frequency* depends on the surface tension, the particle diameter, and the velocity fluctuations. Assuming that breakage occurs independently of other particles, the death term due to breakage is then defined as

$$D_B(\mathbf{x},\mathbf{r},\mathbf{Y},t) = g(\mathbf{x},\mathbf{r},\mathbf{Y},t)f_n(\mathbf{x},\mathbf{r},t), \quad (3.3)$$

where g is the breakage frequency. The birth term due to breakage also considers the probability of formation, $\beta(\mathbf{x},\mathbf{r};\mathbf{x}',\mathbf{r}',\mathbf{Y},t)$ of the particle considered due to breakage of other particles. This is also referred to as the daughter distribution function, or daughter redistribution function [38, 39], and the born fragments of state $(\mathbf{x},\mathbf{r})$ are given by

$$B_B(\mathbf{x},\mathbf{r},\mathbf{Y},t) = \\ \int_{V_{\mathbf{x},SV}} \int_{V_{\mathbf{r},SV}} \nu(\mathbf{x}',\mathbf{r}',\mathbf{Y},t) g(\mathbf{x}',\mathbf{r}',\mathbf{Y},t) \beta(\mathbf{x},\mathbf{r},\mathbf{x}',\mathbf{r}',\mathbf{Y},t) f_n(\mathbf{x}',\mathbf{r}',t) dV_{r'} dV_{x'}. \quad (3.4)$$

The average number of fragments born is denoted ν, and SV denotes the subvolume of interest. To simplify, the number of fragments born due to breakage may be assumed to be binary, i.e. $\nu = 2$. Keeping the general fragmentation function would require another semi-empirical closure, and it would require the fragmentation function to be found at every time step. The assumption is not unreasonable, however, it is a simplification. Given that breakage occurs, the probability over the entire subvolume has to equate to unity, i.e., if a particle breaks, new particles have to be formed. Mathematically this means

$$\int_{V_{\mathbf{x},SV}} \beta(\mathbf{x},\mathbf{r},\mathbf{x}',\mathbf{r}',\mathbf{Y},t) dV_{x'} = 1. \quad (3.5)$$

Imposing the assumptions of binary breakage, $\nu = 2$, and that the particles born from breakage are approximately located at the same place in physical space, (3.4) can be reduced to

$$B_B(\mathbf{x}, \mathbf{r}, \mathbf{Y}, t) = \int_{V_{\mathbf{x},SV}} 2g(\mathbf{x}', \mathbf{r}', \mathbf{Y}, t)\beta(\mathbf{x}, \mathbf{r}, \mathbf{x}', \mathbf{r}, \mathbf{Y}, t) f_n(\mathbf{x}', \mathbf{r}', t) dV_{x'}. \tag{3.6}$$

Note the change in β from (3.4) to (3.6).

The mechanism of coalescence was established in the previous section, and the coalescence terms are considered to depend on a coalescence frequency, a_C. This quantity is dependent on the particles coalescing. For the formation of particles of state $(\mathbf{x}, \mathbf{r})$, particles of state $(\mathbf{x}', \mathbf{r}')$ and particles of state $(\mathbf{x}'', \mathbf{r}'')$ have to coalesce. The general expression for this is denoted

$$\begin{aligned} &B_C(\mathbf{x}, \mathbf{r}, \mathbf{Y}, t) = \\ &\frac{1}{\delta} \int_{V_{\mathbf{r},SV}} \int_{V_{\mathbf{x},SV}} a_C(\mathbf{x}'', \mathbf{r}'', \mathbf{x}', \mathbf{r}', \mathbf{Y}, t) f_n^{(2)}(\mathbf{x}'', \mathbf{r}'', \mathbf{x}', \mathbf{r}', t) dV_{x''} dV_{r'}, \end{aligned} \tag{3.7}$$

where δ denotes the number of particles coalescing. The event only gives rise to one new particle, and the PBE is a number balance equation, so δ has to be included to avoid duplicate contributions. The number function $f_n^{(2)}$ is the average number density distribution of parent particles (coalescing particles) of states $(\mathbf{x}', \mathbf{r}')$ and $(\mathbf{x}'', \mathbf{r}'')$. To simplify, binary coalescence can be assumed, where only two particles are allowed to coalesce. Then $f_n^{(2)}(\mathbf{x}'', \mathbf{r}'', \mathbf{x}', \mathbf{r}', t) \approx f_n(\mathbf{x}'', \mathbf{r}'', t) f_n(\mathbf{x}', \mathbf{r}', t)$ and $\delta = 2$. Physical spatially averaging a_C simplifies the birth term further, and gives rise to the coalescence density, $\kappa_C(\mathbf{x}'', \mathbf{r}'', \mathbf{x}', \mathbf{r}', \mathbf{Y}, t)$. As seen, the integral spans the domain of the parent particle of state ($\mathbf{x}''$, $\mathbf{r}''$) and a Jacobian transformation may be used to express it in terms of the child particle of state ($\mathbf{x}$, $\mathbf{r}$). Imposing all assumptions just discussed yields

$$\begin{aligned} &B_C(\mathbf{x}, \mathbf{r}, \mathbf{Y}, t) = \\ &\frac{1}{2} \int_{V_{\mathbf{x},SV}} \kappa_C(\mathbf{x}'', \mathbf{r}'', \mathbf{x}', \mathbf{r}', \mathbf{Y}, t) f_n(\mathbf{x}'', \mathbf{r}'', t) f_n(\mathbf{x}', \mathbf{r}', t) \frac{\partial(\mathbf{x}'', \mathbf{r}'')}{\partial(\mathbf{x}, \mathbf{r})} dV_{x'}. \end{aligned} \tag{3.8}$$

The same assumptions could be employed for the death process of coalescence, resulting in

$$D_C(\mathbf{x}, \mathbf{r}, \mathbf{Y}, t) = f_n(\mathbf{x}, \mathbf{r}, t) \int_{V_{\mathbf{x},SV}} \kappa_C(\mathbf{x}', \mathbf{r}', \mathbf{x}, \mathbf{r}, \mathbf{Y}, t) f_n(\mathbf{x}', \mathbf{r}', t) dV_{x'}. \tag{3.9}$$

Even though equations for the birth and death processes have been established by (3.3), (3.6), (3.8) and (3.9), new functions (g, β and κ_C) were also introduced. These functions are what is collectively referred to as *kernels*. Coulaloglou and Tavlarides [31], Ross [35], Martínez-Bazán [37, 40], Chen et al. [41], Vankova et al. [32] and many others have done exceptional work on the breakage frequency, g, and the daughter distribution function, β. The coalescence density is usually split in two, a swept volume rate, ω, and a probability of coalescence, ψ_E, given a collision has occurred. Prince and Blanch [33] developed a swept volume rate model based on the kinetic gas theory which is frequently adopted. However, the assumption that the particles are perfectly elastic, and that their relative velocities are defined by $|\mathbf{v}_{rel,t,d,d'}| \approx (\bar{v}_{t,d}^2 + \bar{v}_{t,d'}^2)^{1/2}$ have been questioned by some [10]. The formula is based on the assumption that molecules are alike and can be considered hard spheres. Particles considered in the PBE may be deformable and agglomerate (stick to each other). The assumptions may therefore be violated. Chesters [34] and Kocamustafaogullari and Ishii [42] among others have worked to develop coalescence efficiency models. No analytically available kernels are yet available for both breakage and coalescence, and all of the four kernels discussed (g, β, ω and ψ_E) up until now are semi-empirical, that is, they are derived from mechanical or kinetic concepts, but include fitting parameters that need to be experimentally determined and validated. This is the greatest weakness of population balance modeling.

3.2 The Population Balance Equation for a Batch CSTR

The population balance described in the previous section is general, with very few assumptions. This section is dedicated to derive a specific form of the PBE for a batch CSTR for the experimental setup described in Chapter 1.2, Figure 1.3. The CSTR will consist of an oil dispersed in a continuous water phase, and the required physical data of the emulsion is given in Table 3.1. The temperature is well controlled, and the liquids are assumed incompressible. The two conditions just mentioned imply that the densities of both the oil and the water phase are constant. As a result, a mass conservation property translates to a volume conservation property. The experimental data obtained will measure the oil volume density distribution (VDD), $f_v(\mathbf{x}, \mathbf{r}, t)$, for given diameters, but the number density distribution can easily be converted into the VDD by multiplying the numbers by their respective volumes. To get denser grid points over the domain, the radius is chosen as internal coordinate as it easily relates to the diameter and volume.

Table 3.1: Fluid data for the oil (dispersed) and water (continuous) phases considered. The surface tension for water is not relevant here.

Quantity	Crude Oil B	Water
Surface tension	$22\,\mathrm{mN\,m^{-1}}$	-
Density	$837\,\mathrm{kg\,m^{-3}}$	$1000\,\mathrm{kg\,m^{-3}}$

The oil droplets are dispersed in a continuous water phase, and no mass transfer between the water and oil droplets are considered. It is assumed that there is no hydrostatic pressure gradients of considerable magnitude either, so the droplets do not change size due to pressure differences. Hence, no growth is apparent in the internal coordinate due to motion in abstract property space. The growth of droplets is assumed to be purely from breakage and coalescence phenomena, and as a consequence, the second term in (3.1) is neglected.

The CSTR considered is assumed well enough mixed for the physical space dependency to be negligible, and leaving the dependency in would only require more computational power and add complexity to the model. Adding this complexity also adds uncertainty due to the lack of specific information on the spatial variance of the turbulent energy dissipation rate, ε. The third term in (3.1), denoting the change in the number density due to motion in physical space is therefore neglected. Since volume is conserved, the PBE can act as a volume conservation equation. Before inserting the radius as internal coordinate, consider the number density distribution with volume as internal coordinate

$$\begin{aligned}\frac{\partial f_n(V(r),t)}{\partial t} = &\int_{V(r)}^{\infty} 2g(V(r'),\mathbf{Y},t)\beta(V(r'),V(r),\mathbf{Y},t)f_n(V(r'),t)dV' \\ &- g(V(r)),\mathbf{Y},t)f_n(V(r),t) \\ &+ \int_0^{V(r)/2} \kappa_C(V(r'),V(r''),\mathbf{Y},t)f_n(V(r'),t) \\ &\qquad f_n(V(r''),t)\frac{\partial(V(r''))}{\partial(V(r))}dV' \\ &- f_n(V(r),t)\int_0^{\infty} \kappa_C(V(r'),\mathbf{Y},t)f_n(V(r'),t)dV'.\end{aligned} \tag{3.10}$$

Droplets of volume $V(r)$ may be formed by breakage from all larger droplets ($V(r') > V(r)$) or coalescence from all smaller droplets ($V(r') < V(r)$). This is shown in the first and third term of the right hand side (RHS), respectively. At the same time, droplets of this volume, $V(r)$, may be lost due to breakage, or from coalescing with other droplets of any size. This is shown respectively in the second and fourth term of the RHS. The volume is proportional to the cube of the radius, and inserting for radius as internal coordinate yields

$$\begin{aligned}\frac{\partial f_n(r,t)}{\partial t} = &\int_r^{\infty} 2g(r',\mathbf{Y},t)\beta(r',r,\mathbf{Y},t)f_n(r',t)dr' \\ &- g(r,\mathbf{Y},t)f_n(r,t) \\ &+ \int_0^{r/\sqrt[3]{2}} \kappa_C(r',r'',\mathbf{Y},t)f_n(r',t)f_n(r'',t)\frac{\partial(r'')}{\partial(r)}dr' \\ &- f_n(r,t)\int_0^{\infty} \kappa_C(r',\mathbf{Y},t)f_n(r',t)dr'.\end{aligned} \tag{3.11}$$

The Jacobian transformation in the coalescence birth term is determined by utilizing the volume conservation $V(r) = V(r') + V(r'')$, meaning that $r^3 = r'^3 + r''^3$. Thus $\partial r''/\partial r = \partial(r^3 - r'^3)^{1/3}/\partial r = r^2/(r^3 - r'^3)^{2/3} = r^2/r''^2$. Introducing the VDD as $f_v(r,t) = V(r)f_n(r,t)$ and inserting into (3.11) transforms the PBE into

$$\begin{aligned}\frac{\partial}{\partial t}\left(\frac{f_v(r,t)}{V(r)}\right) = &\int_r^{\infty} 2g(r',\mathbf{Y},t)\beta(r',r,\mathbf{Y},t)\frac{f_v(r',t)}{V(r')}dr' \\ &- g(r,\mathbf{Y},t)\frac{f_v(r,t)}{V(r)} \\ &+ \int_0^{r/\sqrt[3]{2}} \kappa_C(r',r'',\mathbf{Y},t)\frac{f_v(r',t)}{V(r')}\frac{f_v(r'',t)}{V(r'')}\frac{r^2}{r''^2}dr' \\ &- \frac{f_v(r,t)}{V(r)}\int_0^{\infty} \kappa_C(r',\mathbf{Y},t)\frac{f_v(r',t)}{V(r')}dr'.\end{aligned} \tag{3.12}$$

The droplets are considered entirely spherical, and their volume is given by simple geometric considerations: $V(r) = \frac{4}{3}\pi r^3$. From the assumption of constant density, the volume of the drop does not have any temporal change. Hence, the first volume term can be placed outside the time derivative

$$\begin{aligned}\frac{1}{V(r)}\frac{\partial f_v(r,t)}{\partial t} &= \int_r^{\infty} 2g(r',\mathbf{Y},t)\beta(r',r,\mathbf{Y},t)\frac{f_v(r',t)}{V(r')}dr' \\ &- g(r,\mathbf{Y},t)\frac{f_v(r,t)}{V(r)} \\ &+ \int_0^{r/\sqrt[3]{2}} \kappa_C(r',r'',\mathbf{Y},t)\frac{f_v(r',t)}{V(r')}\frac{f_v(r'',t)}{V(r'')}\frac{r^2}{r''^2}dr' \\ &- \frac{f_v(r,t)}{V(r)}\int_0^{\infty}\kappa_C(r',\mathbf{Y},t)\frac{f_v(r',t)}{V(r')}dr'.\end{aligned} \tag{3.13}$$

The PBE as it stands in (3.13) can not be solved numerically because of its dependency of infinite integrals for breakage birth and coalescence death. However, the contributions from these terms will slowly decay as the radius increases, i.e. there exists a maximum radius called R_m that could mimic the behaviors of the infinite integral, making it finite. All contributions from radii above this value are assumed neglected. Introducing this *characteristic length*, presents a finite form of (3.13)

$$\begin{aligned}\frac{1}{V(r)}\frac{\partial f_v(r,t)}{\partial t} &= \int_r^{R_m} 2g(r',\mathbf{Y},t)\beta(r',r,\mathbf{Y},t)\frac{f_v(r',t)}{V(r')}dr' \\ &- g(r,\mathbf{Y},t)\frac{f_v(r,t)}{V(r)} \\ &+ \int_0^{r/\sqrt[3]{2}} \kappa_C(r',r'',\mathbf{Y},t)\frac{f_v(r',t)}{V(r')}\frac{f_v(r'',t)}{V(r'')}\frac{r^2}{r''^2}dr' \\ &- \frac{f_v(r,t)}{V(r)}\int_0^{R_m}\kappa_C(r',\mathbf{Y},t)\frac{f_v(r',t)}{V(r')}dr'.\end{aligned} \tag{3.14}$$

To lower the computational burden and stiffness of the rather stiff[1] PBE, (3.14) will be made dimensionless by introducing two new independent variables and one new dependent variable. The radius has dimensions m and can be made dimensionless by dividing by a characteristic length, namely R_m. The time can also be divided by some *characteristic time*. This characteristic time is denoted t_f, and is taken as the duration of the experiment, i.e. the time from the first measurement is made until the last measurement is made. Hence, the two independent variables introduced are $\xi = r/R_m$ and $\tau = t/t_f$. This also translates the volume to $V(r) = V(R_m\xi) = \frac{4}{3}\pi(R_m\xi)^3 = V_m\xi^3$. The dependent variable $f_v(r,t)$ has units m^{-1} and is made dimensionless by multiplying by the same characteristic length R_m, $\psi(\xi,\tau) = f_v(r,t)R_m$. The differentials in (3.14) will change according to the

[1]The term stiff is reagarded in this context as a rapid change in function value over small perturbations in independent variable (large gradient). Numerical solvers often struggle with integrating stiff equations because the step size taken must be reduced each iteration that is unsatisfactory. Step sizes reduced below tolerance raises errors.

definitions just made: $\partial f_v(r,t) = \partial\psi(\xi,\tau)/R_m$, $\partial t = t_f\partial\tau$ and $dr = R_m d\xi$. The PBE then reads

$$\begin{aligned}
\frac{1}{R_m t_f}\frac{\partial\psi(\xi,\tau)}{\partial\tau} =& V_m\xi^3\int_\xi^1 2g(\xi' R_m,\mathbf{Y},\tau t_f)\beta(\xi R_m,\xi' R_m,\mathbf{Y},\tau t_f)\\
&\times\frac{\psi(\xi',\tau)}{R_m}\frac{1}{V_m\xi'^3}R_m d\xi'\\
&-g(\xi R_m,\mathbf{Y},\tau t_f))\frac{\psi(\xi,\tau)}{R_m}\\
&+V_m\xi^3\int_0^{\frac{\xi}{\sqrt[3]{2}}}\kappa_C(\xi' R_m,\xi'' R_m,\mathbf{Y},\tau t_f)\\
&\times\frac{\psi(\xi',\tau)}{R_m V_m\xi'^3}\frac{\psi(\xi'',\tau)}{R_m V_m\xi''^3}\left(\frac{\xi R_m}{\xi'' R_m}\right)^2 R_m d\xi'\\
&-\frac{\psi(\xi,\tau)}{R_m}\int_0^1\kappa_C(\xi' R_m,\xi R_m,\mathbf{Y},\tau t_f)\frac{\psi(\xi',\tau)}{R_m V_m\xi'^3}R_m d\xi'.
\end{aligned} \tag{3.15}$$

The kernels, g, β and κ_C, now take r and t as their size and time arguments, however there has been a change of variables, so the kernels must be remapped over to other domains. The kernels are redefined accordingly

$$G(\xi,\mathbf{Y},\tau) = g(\xi R_m,\mathbf{Y},\tau t_f) \tag{3.16a}$$
$$B(\xi,\xi',\mathbf{Y},\tau) = \beta(\xi R_m,\xi' R_m,\mathbf{Y},\tau t_f) \tag{3.16b}$$
$$K(\xi',\xi,\mathbf{Y},\tau) = \kappa_C(\xi' R_m,\xi R_m,\mathbf{Y},\tau t_f). \tag{3.16c}$$

By simplifying and canceling terms in (3.15) as well as inserting these new kernels cleans up the PBE

$$\begin{aligned}
\frac{\partial\psi(\xi,\tau)}{\partial\tau} =& t_f R_m\xi^3\int_\xi^1 2G(\xi',\mathbf{Y},\tau)B(\xi,\xi',\mathbf{Y},\tau)\frac{\psi(\xi',\tau)}{\xi'^3}d\xi'\\
&-t_f G(\xi,\mathbf{Y},\tau)\psi(\xi,\tau)\\
&+\frac{t_f}{V_m}\xi^3\int_0^{\frac{\xi}{\sqrt[3]{2}}}K(\xi',\xi'',\mathbf{Y},\tau)\frac{\psi(\xi',\tau)}{\xi'^3}\frac{\psi(\xi'',\tau)}{\xi''^3}\left(\frac{\xi}{\xi''}\right)^2 d\xi'\\
&-\frac{t_f}{V_m}\psi(\xi,\tau)\int_0^1 K(\xi',\xi,\mathbf{Y},\tau)\frac{\psi(\xi')}{\xi'^3}d\xi'.
\end{aligned} \tag{3.17}$$

.

The PBE is now derived for the batch CSTR, however, (3.17) is not closed since kernels have not yet been chosen. Choosing kernels is vital for this modeling task, and it has been shown by Chen et al. [41] that different kernels produce different solutions. Therefore the kernels chosen should reflect the system at hand. For the breakage kernels, the breakage frequency, G, was chosen from Vankova et al. [32], because it was built based on a turbulent oil-in-water emulsion, just as the system in mention. The daughter distribution was chosen from Coulaloglou and Tavlarides [31], for the same reasons as the breakage frequency. Coulaloglou and Tavlarides based their work on a stirred, baffled tank, turbulently agitating the liquid-liquid dispersion they considered.

The coalescence density, K, was split into a swept volume rate and a probability of rupture, given a collision has occurred. When the droplets collide, they stick, and a thin film of continuous phase liquid forms between them. The time it takes for this to drain and eventually rupture, causing the droplets to coalesce, is called the drainage time, t_c. If the interaction time, t_i, exceeds the drainage time, then the droplets may coalesce. The coalescence efficiency is thus generally written

$$\psi_E(r, r') = \exp\left(-\frac{t_c}{t_i}\right). \tag{3.18}$$

Note that this efficiency, ψ_E, is different from the volume density distribution, ψ. The drainage time can be modeled as a differential equation of the height of the film. When the height drops to some critical height, the film ruptures.

The swept volume rate is usually split in three: turbulence-driven collisions, buoyancy-driven collisions, and laminar shear-driven collisions. Normally the total swept volume rate is the linear sum of them, and the rate used in this work is taken from Prince and Blanch [33]. The coalescence efficiency was taken from Vankova et al. [32]. The kernels for both breakage and coalescence is thus presented with radius as internal coordinate

$$g(r, \mathbf{Y}) = k_{b,1}\frac{\varepsilon^{1/3}}{2^{2/3}r^{2/3}}\sqrt{\frac{\rho_d}{\rho_c}}\exp\left[-k_{b,2}\frac{\sigma}{\rho_d 2^{5/3}\varepsilon^{2/3}r^{5/3}}\right] \tag{3.19a}$$

$$\beta(r, r') = \frac{2.4}{r'^3}\exp\left[-4.5\frac{(2r^3 - r'^3)^2}{r'^6}\right]3r^2 \tag{3.19b}$$

$$\omega(r', r'', \mathbf{Y}) = 4\sqrt[3]{2}k_{c,1}\varepsilon^{1/3}(r' + r'')^2(r'^{2/3} + r''^{2/3})^{1/2} \tag{3.19c}$$

$$\psi_E(r', r'', \mathbf{Y}) = \exp\left[-k_{c,2}\frac{\rho_c^{1/2}\varepsilon^{1/3}}{2^{1/6}\sigma^{1/2}}(r_{eq}(r', r''))^{5/6}\right] \tag{3.19d}$$

$$r_{eq}(r', r'') = \frac{1}{2}\left(\frac{1}{r'} + \frac{1}{r''}\right)^{-1}, \tag{3.19e}$$

where $k_{b,1}$, $k_{b,2}$, $k_{c,1}$ and $k_{c,2}$ are model fitted parameters, ε is the turbulent energy dissipation rate, σ is the surface tension of the dispersed droplets, ρ_d is the dispersed phase density, ρ_c is the continuous phase density and r_{eq} is the equidistant radius between the two droplets. By inspecting (3.19), the environment vector is given by $\mathbf{Y} = (\rho_d, \rho_c, \sigma, \varepsilon)$ From the chaotic nature of turbulence, the turbulent energy dissipation rate is stochastic is spatially varying. Since the PBE derived was considered spatially invariant, the turbulent energy dissipation rate is spatially averaged in order to produce a spatially invariant one [43]

$$\bar{\varepsilon} = \frac{P}{\rho_c V_l}. \tag{3.20}$$

P is the power supplied to the emulsion by the agitator, and V_l is the volume of the emulsion.

Since ε is spatially averaged and uncertainties are related to it, the uncertainties propagate into the kernels and the PBE. These uncertainties are captured by the parameters,

$k_{b,1}, k_{b,2}, k_{c,1}, k_{c,2}$. If ε was known, the parameters should close to unity as noted by Vankova et al., Chesters and Prince and Blanch [32, 34, 33]. This may not be the case.

In (3.19), the kernels were presented with dimensional radii as arguments. The PBE in (3.17) is non-dimensional, and inserting the relation from (3.16) into (3.19) yields

$$G(\xi) = k_{b,1}\frac{\varepsilon^{1/3}}{2^{2/3}R_m^{2/3}\xi^{2/3}}\sqrt{\frac{\rho_d}{\rho_c}}\exp\left[-k_{b,2}\frac{\sigma}{\rho_d 2^{5/3}\varepsilon^{2/3}R_m^{5/3}\xi^{5/3}}\right] \tag{3.21a}$$

$$B(\xi,\xi') = \frac{1}{R_m}\frac{2.4}{\xi'^3}\exp\left[-4.5\frac{(2\xi-\xi'^3)^2}{\xi'^6}\right]3\xi'^2 \tag{3.21b}$$

$$\Omega(\xi',\xi'') = R_m^{7/3}4\sqrt[3]{2}k_{c,1}\varepsilon^{1/3}(\xi'+\xi'')^2(\xi'^{2/3}+\xi''^{2/3})^{1/2} \tag{3.21c}$$

$$\Psi_E(\xi',\xi'') = \exp\left[-k_{c,2}R_m^{5/6}\frac{\rho_c^{1/2}\varepsilon^{1/3}}{2^{1/6}\sigma^{1/2}}(r_{eq}(\xi',\xi''))^{5/6}\right] \tag{3.21d}$$

$$r_{eq}(\xi',\xi'') = \frac{1}{2}\left(\frac{1}{\xi'}+\frac{1}{\xi''}\right)^{-1}. \tag{3.21e}$$

By inserting (3.21) into (3.17) and simplifying

$$\begin{aligned}\frac{\partial\psi(\xi,\tau)}{\partial\tau} &= k_1\xi^3\int_\xi^1 2\frac{1}{\xi'^{2/3}}\exp\left[-k_2\frac{1}{\xi'^{5/3}}\right]\frac{2.4}{\xi'^3}\exp\left[-4.5\frac{(2\xi^3-\xi'^3)^2}{\xi'^6}\right]3\xi^2\frac{\psi(\xi',\tau)}{\xi'^3}d\xi' \\ &- k_1\frac{1}{\xi^{2/3}} \\ &\times\exp\left[-k_2\frac{1}{\xi^{5/3}}\right]\psi(\xi,\tau) \\ &+ k_3\xi^3\int_0^{\frac{\xi}{\sqrt[3]{2}}}(\xi'+\xi'')^2(\xi'^{2/3}+\xi''^{2/3})^{1/2}\exp\left[-k_4\left(\frac{1}{\xi'}+\frac{1}{\xi''}\right)^{-5/6}\right] \\ &\times\frac{\psi(\xi',\tau)}{\xi'^3}\frac{\psi(\xi'',\tau)}{\xi''^3}\left(\frac{\xi}{\xi''}\right)^2 d\xi' \\ &- k_3\psi(\xi,\tau)\int_0^1(\xi'+\xi)^2(\xi'^{2/3}+\xi^{2/3})^{1/2} \\ &\times\exp\left[-k_4\left(\frac{1}{\xi'}+\frac{1}{\xi}\right)^{-5/6}\right]\frac{\psi(\xi',\tau)}{\xi'^3}d\xi',\end{aligned} \tag{3.22}$$

where the non-dimensional constants k_1, k_2, k_3 and k_4 are given by the model fitted parameters, the characteristic length and time, and the environment vector recently defined

$$k_1(R_m, t_f, \mathbf{Y}) = t_f k_{b,1} \frac{\varepsilon^{1/3}}{2^{2/3} R_m^{2/3}} \sqrt{\frac{\rho_d}{\rho_c}} \tag{3.23a}$$

$$k_2(R_m, t_f, \mathbf{Y}) = k_{b,2} \frac{\sigma}{\rho_d 2^{5/3} \varepsilon^{2/3} R_m^{5/3}} \tag{3.23b}$$

$$k_3(R_m, t_f, \mathbf{Y}) = \frac{t_f}{V_m} R_m^{7/3} 4\sqrt[3]{2} k_{c,1} \varepsilon^{1/3} \tag{3.23c}$$

$$k_4(R_m, t_f, \mathbf{Y}) = k_{c,2} R_m^{5/6} \frac{\rho_c^{1/2} \varepsilon^{1/3}}{2\sigma^{1/2}}. \tag{3.23d}$$

For the solution of the PBE, these constants do not change, as no environment variables change with time. They can therefore be computed before entering any solvers. To tidy up (3.22), the kernels may be extracted as

$$K_{BB}(\xi, \xi', \mathbf{Y}) = 2\frac{1}{\xi'^{2/3}} \exp\left[-k_2 \frac{1}{\xi'^{5/3}}\right] \frac{2.4}{\xi'^3} \exp\left[-4.5 \frac{(2\xi^3 - \xi'^3)^2}{\xi'^6}\right] 3\xi^2 \tag{3.24a}$$

$$K_{DB}(\xi, \mathbf{Y}) = \frac{1}{\xi^{2/3}} \exp\left[-k_2 \frac{1}{\xi^{5/3}}\right] \tag{3.24b}$$

$$K_{BC}(\xi', \xi'', \mathbf{Y}) = (\xi' + \xi'')^2 (\xi'^{2/3} + \xi''^{2/3})^{1/2} \exp\left[-k_4 \left(\frac{1}{\xi'} + \frac{1}{\xi''}\right)^{-5/6}\right] \tag{3.24c}$$

$$K_{DC}(\xi', \xi, \mathbf{Y}) = (\xi' + \xi)^2 (\xi'^{2/3} + \xi^{2/3})^{1/2} \exp\left[-k_4 \left(\frac{1}{\xi'} + \frac{1}{\xi}\right)^{-5/6}\right]. \tag{3.24d}$$

and substituted in (3.22)

$$\begin{aligned}\frac{\partial \psi(\xi, \tau)}{\partial \tau} &= k_1 \xi^3 \int_\xi^1 K_{BB}(\xi, \xi') \frac{\psi(\xi', \tau)}{\xi'^3} d\xi' \\ &- k_1 K_{DB}(\xi) \psi(\xi, \tau) \\ &+ k_3 \xi^3 \int_0^{\frac{\xi}{\sqrt[3]{2}}} K_{BC}(\xi', \xi'') \frac{\psi(\xi', \tau)}{\xi'^3} \frac{\psi(\xi'', \tau)}{\xi''^3} \left(\frac{\xi}{\xi''}\right)^2 d\xi' \\ &- k_3 \psi(\xi, \tau) \int_0^1 K_{DC}(\xi', \xi) \frac{\psi(\xi', \tau)}{\xi'^3} d\xi'.\end{aligned} \tag{3.25}$$

Equations (3.21), (3.23) and (3.25) close the set of equations, and the PBE can therefore be solved.

3.3 Numerical Procedure

The population balance equation is a partial integro-differential equation (PIDE), and it tends to be stiff. Therefore it is important to choose a numerical method of accuracy that evaluates the integrals as well as the differentials precisely. A thorough introduction to numerical mathematics was given in Chapter 2, and emphasis was put on their accuracy. Since the integrals require high precision, Riemann sums and trapezoids are not relevant. The integrals are therefore approximated by higher order polynomials, starting from Simpson's method. However, oscillations may occur at the boundaries for interpolating polynomials with an equidistant grid. This is called Runge's phenomenon, and as a consequence of this, a higher density of grid points must be placed at the boundaries. This motivates the use of Gauss, Gauss Lobatto or Gauss Radau quadratures. These numerical approximations have a high density of collocation points at the endpoints, and they are of high accuracy. The collocation points are roots of Jacobi polynomials and are therefore calculated on the domain where these polynomials are defined. This is not particularly useful unless the model is remapped over to this reference domain, Ω, or inversely the collocation points are remapped to the physical domain, where the model is defined. A remapping procedure for both the collocation points and Gaussian quadrature weights may be applied to remap from Ω to the closed interval $[\hat{x}_0, \hat{x}_N]$

$$[\hat{x}]_i = \frac{\hat{x}_N - \hat{x}_0}{x_{\Omega,N} - x_{\Omega,0}}([x_\Omega]_i - x_{\Omega,0}) + \hat{x}_0 \tag{3.26a}$$

$$[\hat{w}]_i = \frac{\hat{x}_N - \hat{x}_0}{x_{\Omega,N} - x_{\Omega,0}}[w_\Omega]_i, \tag{3.26b}$$

where i denotes that it is the i-th collocation point in the vector of collocation points. To illustrate, consider $\Omega = [-1, 1]$ and $\hat{x} \in [1, 10]$. Then the remapping procedure will preserve the spacing of the collocation points and weights. This is presented in Figure 3.2 and Figure 3.3, respectively. Note that the amplitude of the weight has changed in the latter of them.

The Jacobi polynomials chosen are the ultraspherical Legendre polynomials on the reference domain, $\Omega = (-1, 1)$, with weight function $W(x) = 1$. They are well known for being robust in terms of the condition number of the coefficient matrix, as well as having low errors as function of polynomial order [44]. The location of the roots depend on the order of the polynomial. A higher order polynomial will result in more collocation points, and according to the notation in (3.26), $x_{\Omega,0}$ and $x_{\Omega,N}$ will move towards -1 and 1, respectively. The endpoints -1, 1 are enforced with the Gauss Lobatto procedure discussed in Chapter 2.

Since all the integrals in (3.25) are on different physical domains, a change of variables may be applied. In order to avoid defining one set of collocation points and integral weights for all integrals, it is convenient that they span the same domain. Therefore they are all remapped to the domain where the VDD is defined, namely $[\hat{x}_0, \hat{x}_N] = [0, 1]$. The coalescence death term is left as it is, but the birth terms are, however, rescaled by introducing two new dimensionless quantities

$$\gamma = \frac{\xi' - \xi}{1 - \xi} \tag{3.27a}$$

$$\alpha = \frac{\xi'}{\frac{\xi}{\sqrt[3]{2}}}. \tag{3.27b}$$

The introduced quantities also have Jacobian transformations, and those are reflected by

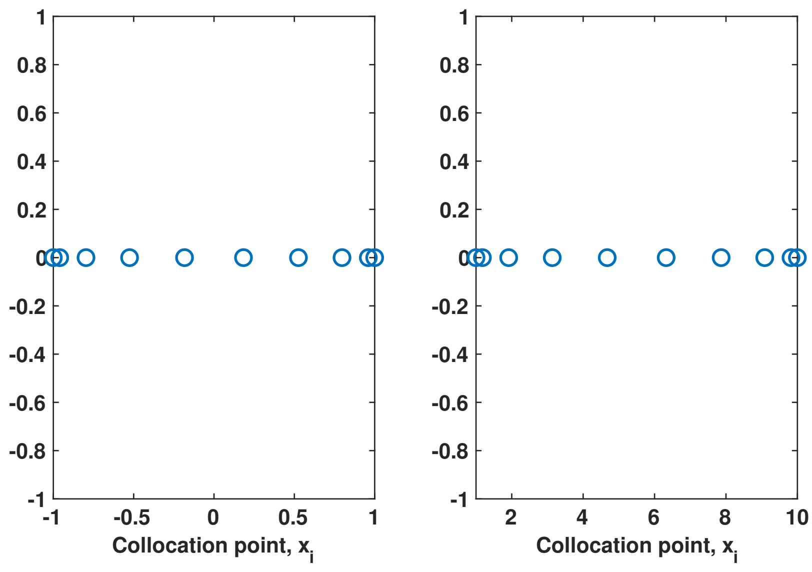


Figure 3.2: The collocation points were remapped from the reference domain [-1, 1] to another arbitrary domain [1, 10] for demonstration purposes.

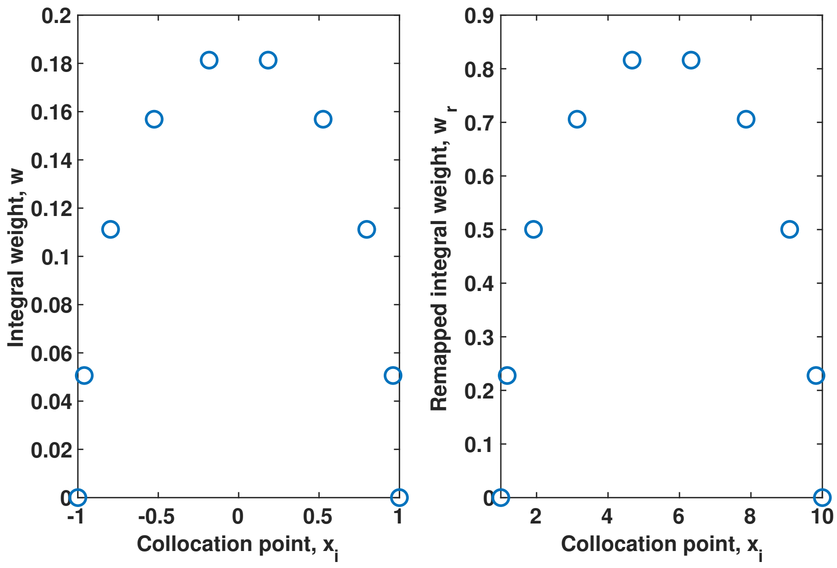


Figure 3.3: The integral weights were remapped from the reference domain [-1, 1] to an arbitrary domain [1, 10] for demonstration purposes.

the change in differentials in the integrals. The arising PBE that is ready for implementation reads

$$\begin{aligned}\frac{\partial\psi(\xi,\tau)}{\partial\tau} &= k_1\xi^3\int_0^1 K_{BB}(\xi,\xi')\frac{\psi([1-\xi]\gamma+\xi,\tau)}{\xi'^3}(1-\xi)d\gamma \\ &- k_1K_{DB}(\xi)\psi(\xi,\tau) \\ &+ k_3\xi^3\int_0^1 K_{BC}(\xi',\xi'')\frac{\psi(\frac{\xi\alpha}{\sqrt[3]{2}},\tau)}{\xi'^3}\frac{\psi(\xi[1-\frac{\alpha^3}{2}]^{1/3},\tau)}{\xi''^3}\left(\frac{\xi}{\xi''}\right)^2\frac{\xi}{\sqrt[3]{2}}d\alpha \\ &- k_3\psi(\xi,\tau)\int_0^1 K_{DC}(\xi',\xi)\frac{\psi(\xi',\tau)}{\xi'^3}d\xi',\end{aligned} \tag{3.28}$$

or alternatively on fully discretized form

$$\begin{aligned}\frac{\partial\psi(\xi,\tau)}{\partial\tau} &= k_1\xi^3\sum_{i=1}^N w_iK_{BB}(\xi,[1-\xi]\gamma_i+\xi)\frac{\psi([1-\xi]\gamma_i+\xi,\tau)}{([1-\xi]\gamma_i+\xi)^3}(1-\xi) \\ &- k_1K_{DB}(\xi)\psi(\xi,\tau) \\ &+ k_3\xi^3\sum_{i=1}^N w_iK_{BC}\left(\frac{\xi\alpha_i}{\sqrt[3]{2}},\xi[1-\frac{\alpha_i^3}{2}]^{1/3}\right) \\ &\times\frac{\psi(\frac{\xi\alpha_i}{\sqrt[3]{2}},\tau)}{\left(\frac{\xi\alpha_i}{\sqrt[3]{2}}\right)^3}\frac{\psi(\xi[1-\frac{\alpha_i^3}{2}]^{1/3},\tau)}{(\xi[1-\frac{\alpha_i^3}{2}]^{1/3})^3}\left(\frac{\xi}{\xi[1-\frac{\alpha_i^3}{2}]^{1/3}}\right)^2\frac{\xi}{\sqrt[3]{2}} \\ &- k_3\psi(\xi,\tau)\sum_{i=1}^N w_iK_{DC}(\xi_i,\xi)\frac{\psi(\xi_i,\tau)}{\xi_i^3},\end{aligned} \tag{3.29}$$

As seen in the transition from (3.28) to (3.29), the equation is transformed to a system of differential equations. The variables ψ, ξ and τ are therefore vectors. In each equation, the VDD is interpolated from ξ' over to γ and α for breakage and coalescence birth, respectively. The interpolations are performed using Steffen's method, a method of piecewise cubic splines that preserves monotonicity (non-oscillatory behavior) in the polynomial interpolated. The kernels are time independent, and to save computing power, they are therefore evaluated before entering the solver. The weights are known and computed from (2.43), (2.44) and (2.53).

3.4 Experimental Details and Measurements

In Chapter 1.2, the experimental setup was shown. The setup consists of a CSTR with the oil-in-water emulsion, and to measure the VDD, a MasterSizer 3000 is used. This is a light-scattering device that measures the diameter of a certain droplet. After measuring all droplets, it returns the percentage of the total volume that is occupied by each size class measured. The particular data set which will be regressed is shown in Figure 3.4. The experimental radii of interest range from $1\,\mu\text{m}$ to $200\,\mu\text{m}$. Also, each distribution has been rescaled to match the phase fraction at which the experiment was conducted, $\varphi = {}^{V_{oil}}/_{V_{emulsion}} = 0.7 \times 10^{-2}$ (0.7%)

$$f_v^{exp}(r,t_j) = \frac{\varphi}{\int_{r=r_0}^{r=r_N} f_{v,0}^{exp}(r,t_j)dr} f_{v,0}^{exp}(r,t_j), \tag{3.30}$$

where $r_0 = 1\,\mu\text{m}$ and $r_N = 200\,\mu\text{m}$. If the solution of the PBE should ever violate the mass balance, the integral at any given time should be unequal to the phase fraction. Thus, the validity of a simulation can easily be checked by checking the integral at each time step.

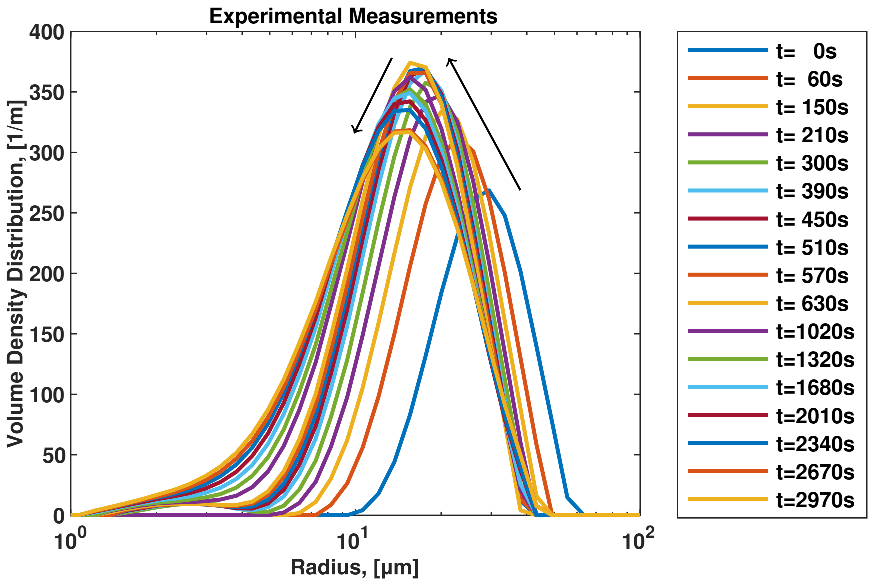


Figure 3.4: The experimental data obtained from Moon [45]. Each distribution shown is a measurement in time. The arrows show which direction the volume density distribution moves with respect to time.

By inspecting Figure 3.4, it is observed that the distribution tends towards smaller droplets as time passes on. This implies strong breakage, which is further confirmed by the long leading edge. It should also be noted that the trailing edge is held back and has a steep gradient. This may imply that coalescence produces droplets at this size, so that breakage death and coalescence birth have reached an equilibrium at this size. Also it should be pointed out how the distribution rises in the beginning. As the volume is constant, the area under the distribution must also be constant, and therefore, if the distribution rises, it has to narrow. At the end it settles down again, causing the distribution to widen. This is due to the production of smaller droplets.

Table 3.2: The table shows the physical data and some parameter values.

Quantity	Value
φ	0.7×10^{-2}
P	$0.366\,\mathrm{W}$
V_l	$725 \times 10^{-6}\,\mathrm{m}^3$
R_m	$500 \times 10^{-6}\,\mathrm{m}$
V_m	$5.24 \times 10^{-10}\,\mathrm{m}^3$
t_f	$2970\,\mathrm{s}$
ρ_c	$1000\,\mathrm{kg\,m^{-3}}$
ρ_d	$837\,\mathrm{kg\,m^{-3}}$
σ	$22 \times 10^{-3}\,\mathrm{N\,m^{-1}}$

3.5 Regression Approaches

An equation can obviously be solved numerically, but it is not of relevance if it is not experimentally validated and verified. Solving the PBE is no different; the PBE contains four parameters in this case, and those are to be fitted to experimental data for the model to be useful. The model was in the previous sections deduced for a batch CSTR, just as the lab setup for the experimental data provided by Moon [45].

The main goal of this work is to validate the model derived, and the approaches on this are trifold. All approaches are non-linear regression techniques, however the cost function to optimize for varies. The first approach is to consider residuals between the modeled and experimental data for all size classes measured *and* all measurements in time, i.e.

$$\begin{aligned} \min_{\beta} \quad & J = \sum_{j=1}^{N_t} \sum_{i=1}^{N_p} (f_v^{exp}(r_i, t_j) - f_v^{num}(r_i, t_j; \beta))^2 \\ \text{s.t.} \quad & \beta_i \geq 0, \forall \beta_i \in \beta, \end{aligned} \tag{3.31}$$

where β is the fitting vector, composed of $k_{b,1}$, $k_{b,2}$, $k_{c,1}$ and $k_{c,2}$, N_t is the number of measurements in time, and N_p is the number of size classes measured. This formulation is referred to as the sum of squared errors on the distributions (dSSE). As (3.31) suggests, the PBE presented in (3.25) must be dimensionalized and interpolated back onto the domain where the size classes are measured before the comparisons can be made, that is, the residuals must be defined for the same radius and time, r_i and t_j. The minimization statement requires the model to be solved iteratively, constantly trying new parameter values to chart the descent direction. This can be time consuming, and emphasis should be put on runtimes for the model evaluations.

The second technique of regression is very similar to the first, however a weighting function will be imposed to inform the optimizer of which measurements should be emphasized. Since the model is spatially invariant, the high shear near the impeller will not be reflected in the model. Hence, smaller droplets produced in experimental data may not be accounted for. The weighting function may be chosen as a shifted Sigmoid function, or an error function, so that the weighting on these small droplets approaches zero. With this method, the objective is to be able to recreate the main trend without emphasizing droplets that are unaccounted for. The weighting function is as follows[2]

$$w(r) = \frac{1}{1 + \exp \frac{m-r}{s}}, \tag{3.32}$$

where $m = 6\,\mu\text{m}$ is the r-value of the midpoint of the s-shaped function and $s = 1\,\mu\text{m}$ is the slack of the function. Decreasing the slack will result in a steeper slope. The parameters m and s was chosen after inspecting the experimental measurements in detail. The resulting optimization problem is written

$$\begin{aligned} \min_{\beta} \quad & J = \sum_{j=1}^{N_t} \sum_{i=1}^{N_p} w(r_i)(f_v^{exp}(r_i, t_j) - f_v^{num}(r_i, t_j; \beta))^2 \\ \text{s.t.} \quad & \beta_i \geq 0, \forall \beta_i \in \beta. \end{aligned} \tag{3.33}$$

This formulation is referred to as the weighted sum of squared errors on the distributions (wdSSE).

The third technique employed will be slightly different, however the idea is the same. The focus will be to locate the center of volume, referred to as the *mean*, μ. It is the first statistical moment, and it is calculated from (2.58) with $j = 1$. The cost function will not be a function of radii directly, since the mean is the average over the radial domain. The problem reads

$$\begin{aligned} \min_{\beta} \quad & J = \sum_{i=1}^{N_t} (\mu^{exp}(t_i) - \mu^{num}(t_i; \beta))^2 \\ \text{s.t.} \quad & \beta_i \geq 0, \forall \beta_i \in \beta. \end{aligned} \tag{3.34}$$

This formulation is referred to as the sum of squared errors of the mean (mSSE). Fitting the mean of the distribution may put less weight on fitting all measurements exactly, but instead catch the main trend of where the distribution is shifting. Even though the objective with this method is to fit the mean, the parameter combination to be found will produce some modeled distribution which will be compared to experimental distributions. This is to verify that the parameter combination indeed is a good one, not falling under the false impression that a reasonably good fit for the mean is sufficient.

[2]Note that the weighting function, $w(r)$, is not to be confused with the integral weights, w_i. The former is used in the minimization objective, and the latter is used to discretize integrals.

Chapter 4

Results and Discussion

The population balance equation presented in (3.29) was solved using orthogonal collocation as presented in Chapter 2, and implemented in MATLAB and C++. The computational burden of performing parameter estimation on the PBE was found to be too expensive for MATLAB and its built-in function `lsqcurvefit`, and so the program was rewritten to object-oriented code in C++. The rewritten program made use of external numerical libraries, specifically SUNDIALS [46] for the solution of ordinary differential equations (ODEs), and GNU Scientific Library (GSL) [47] for linear algebra, interpolation and parameter estimation. Dakota [48] was also unsuccessfully attempted for parameter estimation.

As noted earlier, there are a few limitations to the model that was developed in Chapter 3. The model was developed for a zero-dimensional CSTR, that is, it has been assumed spatially invariant. Consequently, the turbulent energy dissipation rate, $\bar{\varepsilon}$, has been spatially averaged as well. As a result, a high shear rate near the impeller of the tank is unaccounted for. By inspecting Figure 4.1, there is a strong coalescence birth contribution at the right tail. There is also strong breakage over the whole radial span. However, for the breakage to be that strong for both large droplets *and* small droplets, and for coalescence to be that strong for large droplets and weak for small droplets is contradictory, unless the shear rate varies over the tank. Breakage is naturally dominant for large droplets, however, when the production of small droplets becomes large, the collision frequency in (3.21d) grows large, consuming the smaller droplets by coalescence. By this reasoning, the smaller droplets could be produced due to either a) spatially varying ε, b) several fragments born due to breakage, or c) unmodeled contribution from the pump in Figure 1.3. Either way, the model fails to account for the small droplets, and emphasis will not be put on these, but rather on the main trend of the evolution and the radii at which the distributions peak (mode). The long tail pushes down the experimental distribution, and therefore, the value of the peak is also anticipated to be off for the final fit.

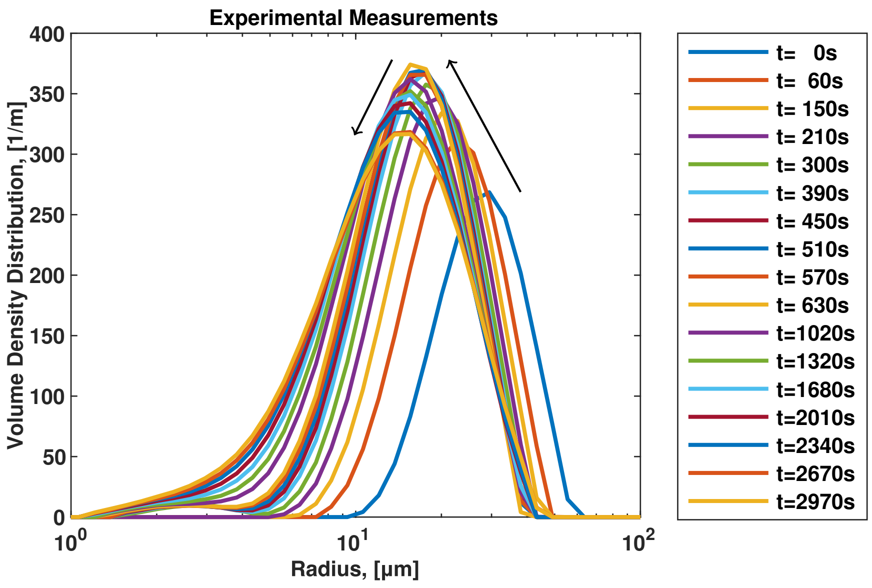


Figure 4.1: The experimental data obtained from Moon [45]. Each distribution shown is a measurement in time. The arrows show which direction the volume density distribution moves with respect to time. The figure is reprinted from Chapter 3.4 for accessibility in the current chapter.

4.1 Charting the Parameter Space

Parameter estimation on non-linear functions results in non-linear regression. One of the main issues regarding non-linear optimization is the non-convexities of the objective function formulations. Convex problems ensure that any local minimum is also the global minimum [49]. For non-convex problems, this is not the case, which means algorithms specializing in these minimization objectives may or may not find the global minimum. Possibly it may get stuck in local minima, or encounter non-descending search directions for bad initial guesses. Therefore, the parameter estimation of this non-convex problem requires a good initial guess. Finding this guess requires knowledge of the behavior of the model, and it requires the user to be familiar with how the errors will be a function of the different parameter combinations.

By inspecting equations (3.23) and (3.24), it is seen that $k_{b,2}$ and $k_{c,2}$ are in the exponential terms with negative signs. For an arbitrary negative exponent, the function looks like the plot presented in Figure 4.2. When $x \to 0$, the function takes the value of unity, and when $x \to \infty$, the function takes the value of zero. In between, the function is sensitive to change. However, the values outside the interval $[1 \times 10^{-3}, 1 \times 10^{1}]$ approximate to 1 and 0 respectively, and hence, the function can be considered insensitive to change there. That means, there exist values for $k_{b,2}$ and $k_{c,2}$ for which the kernel is insensitive to change. Keeping $k_{b,1}$ and $k_{c,1}$ constant may result in insensitivities in the objective function with respect to the exponential parameters $k_{b,2}$ and $k_{c,2}$. This is explored further later.

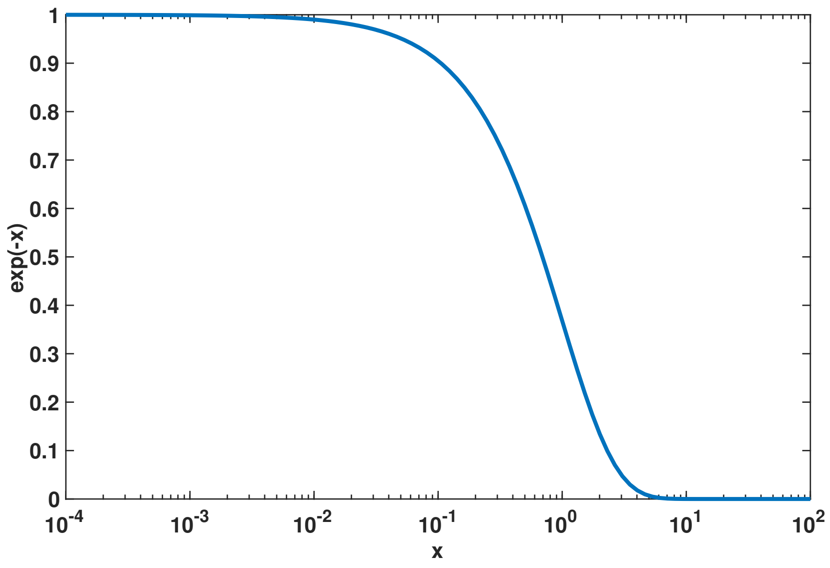


Figure 4.2: For a negative exponential function, the function takes the value of 1 when $x \to -\infty$, and 0 when $x \to \infty$. In between, the function is sensitive to change.

The author of this work previously familiarized himself with the model and the validity range of the parameters to fit. He discovered that $k_{b,1}$ and $k_{c,1}$ mainly govern the dynamical behavior of the VDD, and that $k_{b,2}$ and $k_{c,2}$ mainly govern where the steady state VDD settles. Increasing $k_{b,2}$ will decrease the influence of breakage on smaller droplets, and increasing $k_{c,2}$ will decrease the influence of coalescence on larger droplets. For the duration of this experiment, t_f, dynamical parameters ($k_{b,1}$ and $k_{c,1}$) above unity is unreasonable as it makes the VDD reach steady state too fast. As a result, an upper bound could be placed on these parameters.

Another approach was used to find bounds for the other two parameters. First, coalescence was disregarded (setting its dynamical parameter, $k_{c,1}$, to zero), and then the log-normal distribution depicted in Figure 4.3a was used as initial condition in the PBE. Then, $k_{b,2}$ was varied until droplets below some critical size was unaffected by breakage, that is, breakage did not produce droplets below this size. This critical radius was determined from the experimental data in Figure 4.1 to be approximately $5\,\mu\text{m}$, and the modeled dynamical evolution is shown in Figure 4.4a. With $k_{b,1} = 3 \times 10^{-6}$, the parameter found was $k_{b,2} = 2 \times 10^{-4}$. Any smaller values would produce smaller droplets.

The same procedure was applied for coalescence with a different log-normal distribution. The objective here was to limit the contribution of coalescence below some critical radius. This radius was found from the experimental data to be approximately $40\,\mu\text{m}$. Thus, breakage was disregarded (setting $k_{b,1}$ to zero), and $k_{c,2}$ was varied until coalescence stopped producing droplets above $40\,\mu\text{m}$. With a dynamical parameter of $k_{c,1} = 1 \times 10^{-4}$, this value was found to be $k_{c,2} = 3 \times 10^{2}$. Any smaller value would produce larger droplets. The log-normal distribution used as initial condition is seen in Figure 4.3b, and the evolution is seen in Figure 4.4b.

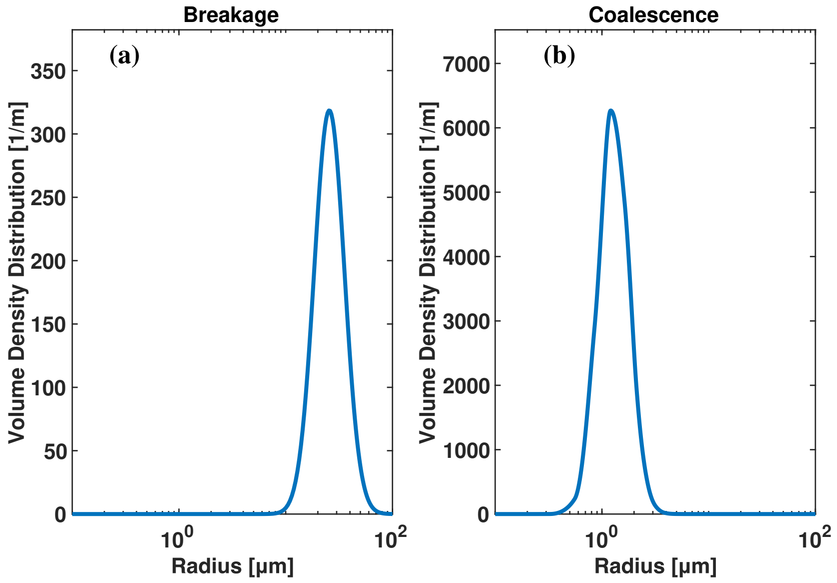


Figure 4.3: Initial conditions used to find bounds for (a) $k_{b,2}$ (breakage only), and (b) $k_{c,2}$ (coalescence only).

By doing this experiment for both breakage and coalescence it was also found that the parameters are, to some extent, coupled. This means, when switching off either phenomenon, the parameters for the other phenomenon would not affect the dynamical behavior of the VDD independently. Even though $k_{b,1}$ and $k_{c,1}$ mainly affect how fast the VDD reaches steady state, they also to some extent determine where it settles. The magnitude of contribution of coalescence can for instance be amplified by increasing $k_{c,1}$, even for values of $k_{c,2}$ that would normally be limiting.

The goal of this work is to find the optimal parameter combination that drives the residuals to their minimum. Suspecting coupled parameters and non-descending search directions (flat gradients) motivated charting the search space for the objective functions formulated in Chapter 3.5. The parameters found previously for breakage were held constant, and the coalescence parameters were varied over an appropriate parameter space. This experiment will from now on be referred to as the coalescence experiment. The cost function in (3.31) – which is the sum of squared errors for the VDD (dSSE) – is presented as a function of $k_{c,1}$ and $k_{c,2}$ in Figure 4.5. As observed, there seems to be some values of $k_{c,2}$ that flatten out the objective function as foreseen. However by traversing the space of $k_{c,1}$, the descent direction is found. The plot shows an objective function that somewhat resemble the non-linear Rosenbrock function. It has a clear minimum at the bottom of the valley, but this minimum is for a fixed set of breakage parameters, so whether it is the global optimum or not is yet to be answered. The error rises quickly with regard to $k_{c,2}$,

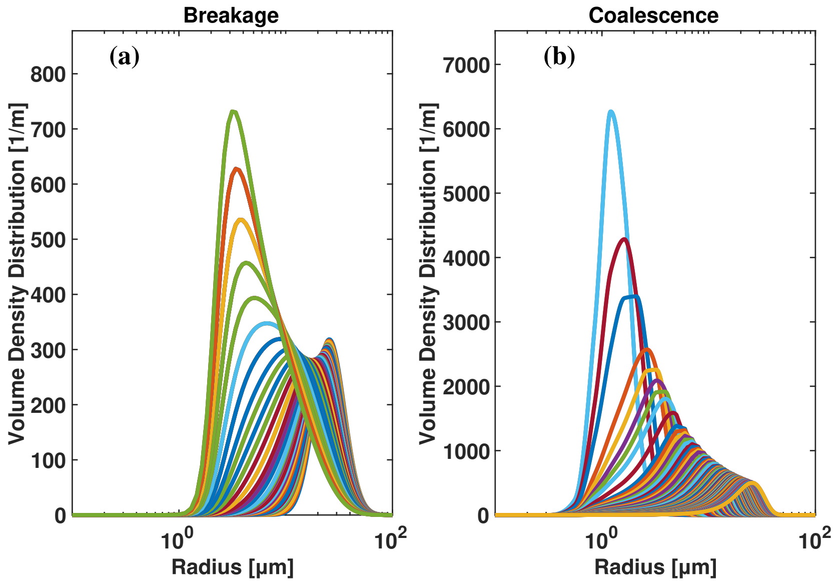


Figure 4.4: The log-normal experiment produced the temporal evolution of the volume density distribution depicted. The cases were done for (a) breakage only, and (b) coalescence only.

and the steep gradient indicates that for some perturbations of the initial guess of $k_{c,2}$, an optimizer is likely to fail. Traversing larger values than $k_{c,2} = 4 \times 10^3$ gave a flat gradient and is not presented here. The optimal parameter combination is given along with those for the other regression approaches in Table 4.1.

The cost function in (3.33) – the weighted sum of squared errors for the VDD – is presented as a function of the coalescence parameters, $k_{c,1}$ and $k_{c,2}$, in Figure 4.6. It should be pointed out that since the weight function ranges from zero to unity, the wdSSE can never take values greater than the original dSSE presented in Figure 4.5. It is observed that the wdSSE has the same trends as the dSSE, which is to be expected. However, it has narrowed and become shallower. This makes it harder to identify and provide an initial guess inside the valley. Initializing a gradient-based optimizer at the flat areas will probably not lead anywhere. Increasing the areas of flatness is therefore not desirable. The optimal parameter combination is shown in Table 4.1, and surprisingly, it was very similar to that of the dSSE. The findings were odd and required further investigation.

Finally, the coalescence experiment was completed by charting the sensitivities of the mSSE as a function of $k_{c,1}$ and $k_{c,2}$. This is the third cost function formulated in Chapter 3.5, and it is seen in (3.34). The result is shown in Figure 4.7. Also here, the valley twists along larger $k_{c,1}$ and $k_{c,2}$. The valley may be approached from the rightmost part of the figure with a gradient-based optimizer. From that angle the cost function descends gradually towards the minimum, making it possible to traverse the objective function down to the optimal coalescence parameters. However, coming from small values of $k_{c,2}$ may be troubling. There is a large bump in the upper left corner of the figure, and this bump tells

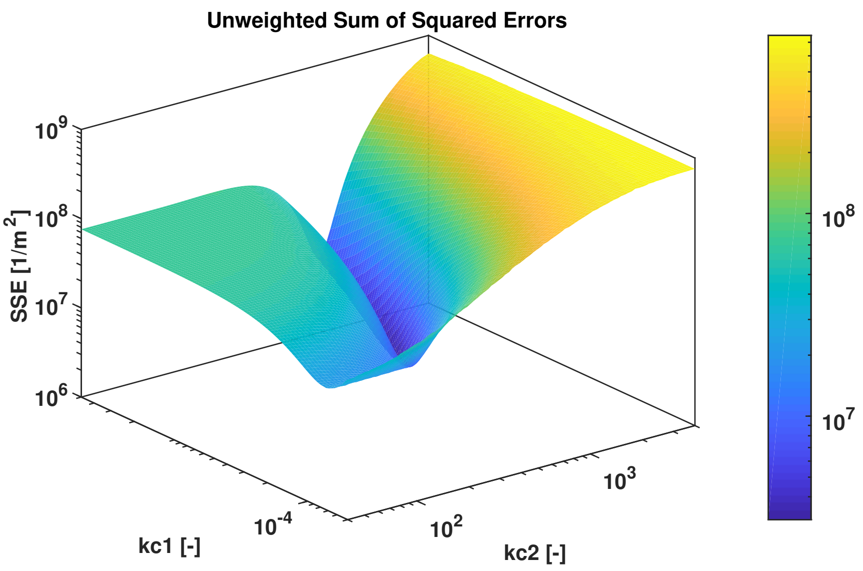


Figure 4.5: The cost function in (3.31) is depicted as a function of the coalescence parameters, $k_{c,1}$ and $k_{c,2}$. The breakage parameters, $k_{b,1}$ and $k_{b,2}$, were held constant in the charting procedure. The figure shows a clear minimum at the bottom of the valley.

the optimizer to move in the other direction. The bump may result in a returned "optimal" value at approximately $k_{c,1} = 1 \times 10^{-3}$ and $k_{c,2} = 50$. Obviously this is not optimal, and the cost function takes values two orders of magnitude greater than the "true" optimum. It is therefore true that for these breakage parameters, the mSSE has several local minima.

Table 4.1: By keeping the breakage parameters constant, the optimal coalescence parameters were determined by charting an appropriate parameter space. The optima are listed for the different optimization formulations. The objective function value is abbreviated OFV.

Method	OFV	$k_{b,1}$	$k_{b,2}$	$k_{c,1}$	$k_{c,2}$
dSSE	3.08×10^6	3×10^{-6}	2×10^{-4}	8.54×10^{-4}	5.67×10^2
wdSSE	2.94×10^6	3×10^{-6}	2×10^{-4}	6.46×10^{-4}	5.17×10^2
mSSE	6.20×10^2	3×10^{-6}	2×10^{-4}	3.77×10^{-3}	8.89×10^2

For the parameter combinations found in Table 4.1, the dynamic simulations were carried out to compare the actual fit to the experimental data. The simulations show both the temporal evolution, the steady state, and the fit of the mean, all of which are presented in Appendix A.1. It is, for all cases, seen that breakage is not strong enough for any of the optima. None of the optima seem to get the mean low enough (approximately $20\,\mu\text{m}$), and none of them get the steady state solution correct. The variations between the normal dSSE and the wdSSE are also not emphasized in either of the figures. The discoveries are unsatisfactory, and they suggest that other breakage parameters should be attempted.

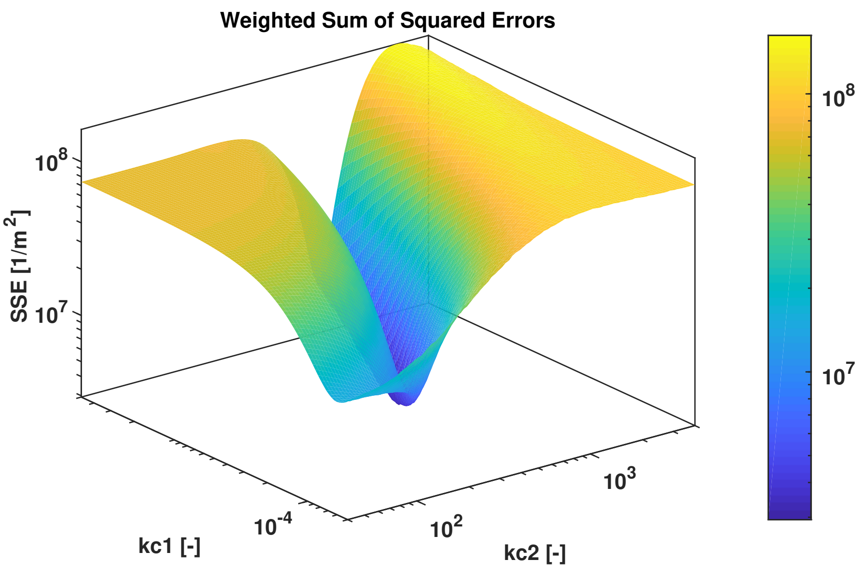


Figure 4.6: The cost function of (3.33) is depicted as a function of the coalescence parameters, $k_{c,1}$ and $k_{c,2}$. The breakage parameters, $k_{b,1}$ and $k_{b,2}$, were held constant in the charting procedure. The figure shows a clear minimum at the bottom of the valley, however, the valley is shallower, and it is narrower than for the dSSE.

By the coalescence experiment, both of the suspected problems were confirmed; the dSSE and the wdSSE contained flat gradients, and the mSSE contained multiple minima, both of which complicates the optimization process due to the difficult gradient navigation. However, both the dSSE and the wdSSE had clear minima at the bottom of a valley, and there should be no difficulties of navigating the valley, given that a sufficiently good initial guess is provided. The minimum may or may not be found for bad initial guesses, based on some factors: a) the Jacobian step size taken, b) function tolerance and c) the choice of algorithm. For greater Jacobian step sizes, the optimizer will take longer steps in the defined search direction, and hopefully it will arrive at a point of descension. From there it is likely to find its way through the valley. For smaller function tolerances, the optimizer will keep navigating the flat areas until the relative change in the objective function is less than the threshold. If this threshold is small enough, it may cross the flat area and arrive at a point of larger gradients, successfully finding its way to the minimum. There exist many different algorithms, and each algorithm specializes in different topics. Employing an algorithm meant for navigating areas of flat gradients could also be attempted.

The three points just mentioned are for bad initial guesses. Hopefully this can be avoided. The three "solutions" do not offer any guarantee of solving the problem, and at best it *may* find the optimum. This is not good enough, and other methods were employed to guarantee an initial guess at a point of descension. The methods include drawing the cost function as a function of the two-dimensional parameter space, creating similar surface plots as for Figure 4.5, Figure 4.6 and Figure 4.7, but for varying breakage parameters as

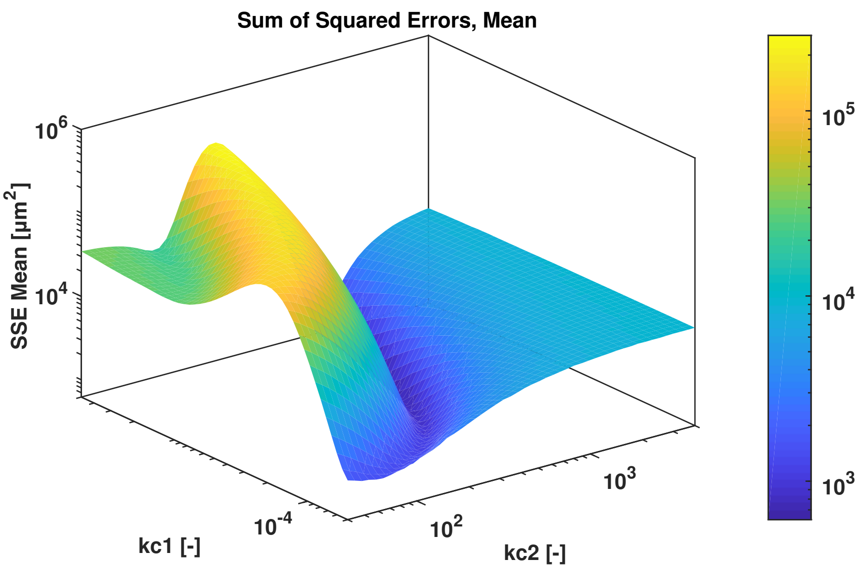


Figure 4.7: The cost function of (3.34) is depicted as a function of the coalescence parameters, $k_{c,1}$ and $k_{c,2}$. The breakage parameters, $k_{b,1}$ and $k_{b,2}$, were held constant in the charting procedure. The valley is pronounced right after a bump in the cost function.

well. That is, the coalescence experiment is repeated for different breakage parameters, successfully charting the entire parameter space. The squared residual surfaces will be shown from different viewpoints to highlight difficulties in the navigation problem.

4.2 Augmenting the Search Space

As expected, the breakage parameter combination used for the coalescence experiment was not optimal. However, the findings of the experiment are still important and highly relevant. The experiment has revealed the nature of the three optimization formulations, at least in the coalescence parameter space. The discoveries show that a random initial guess will be a bad idea, and that the parameter estimation task is non-trivial. Given the fit in Appendix A.1, an augmented parameter search could possibly provide stronger breakage, while still keeping the pronounced coalescence edge at large droplets. The augmentation spanned four orders of magnitude in $k_{b,1}$ and $k_{c,1}$, and five orders of magnitude for $k_{b,2}$ and $k_{c,2}$. The parameter space is shown in Table 4.2.

The PBE is a stiff system, and some parameter combinations stiffens the system further. Sometimes this resulted in long runtimes, and for a four-dimensional search space, the computational burden grew significantly. To get sufficiently high accuracy per simulation, 200 collocation points were used in the radial domain. By having a search space of the dimensions specified in Table 4.2, even the coarse grid of 10 points in each direction resulted in 10000 simulations with 200 collocation points in each simulation. This coarse

Table 4.2: The table presents the coarse parameter space of the augmented parameter search. The dSSE, wdSSE and the mSSE were charted in this parameter space.

Bound	$k_{b,1}$	$k_{b,2}$	$k_{c,1}$	$k_{c,2}$
Lower	1×10^{-7}	1×10^{-5}	1×10^{-7}	1×10^{-2}
Upper	1×10^{-3}	1	1×10^{-3}	4×10^{3}

search did not offer very high precision, but it displayed the nature and trend of the objective functions, and how they responded to changes in the different parameters, without rendering the system too large. The coarse charting is shown in Appendix A.2 and the optimal parameter combination is threefold; there is one optimal parameter combination for each optimization formulation. The three optimal combinations are listed in Table 4.3[1]. For visualization purposes, the surface plots are shown for varying parameters x and y at the optimal parameter combination z and w. That is, the optimal parameter combination was found, and the two-dimensional parameter variations are at the fixed, optimal combination of the other two parameters. As seen, there are vast areas of insensitivity for two fixed, optimal parameters, some of them even coming from unsolvable PBEs arising from stiff parameter combinations. Those combinations are the yellow areas. These insensitive areas are not particularly interesting, and the sensitive areas were not accurate enough to draw any conclusions.

Table 4.3: The optimal parameter combinations for the three different optimization formulations. The combinations are crude, and they are for the coarse grid. The objective function value is abbreviated OFV.

Method	OFV	$k_{b,1}$	$k_{b,2}$	$k_{c,1}$	$k_{c,2}$
dSSE	2.25×10^{6}	4.64×10^{-5}	4.64×10^{-4}	1.00×10^{-3}	2.28×10^{2}
wdSSE	1.89×10^{6}	4.64×10^{-5}	4.64×10^{-4}	1.00×10^{-3}	2.28×10^{2}
mSSE	6.53	3.59×10^{-4}	5.99×10^{-3}	1.29×10^{-4}	2.28×10^{2}

It is interesting to note that the $\frac{4!}{2!(4-2)!} = 6$ different variations in Appendix A.2 were very different in nature. Some of them, for instance variations in $k_{b,1}$ and $k_{c,1}$, produced smooth, well-defined surfaces, whereas others, for instance variations in $k_{b,1}$ and $k_{c,2}$, produced jagged surfaces, which can be difficult to extract any information from. For fixed $k_{b,2}$ and $k_{c,1}$, the surfaces in Figure A.10, Figure A.11 and Figure A.12 will most likely be hard to navigate, especially if the step size taken is relatively large. Those for varying $k_{b,2}$ and $k_{c,2}$ are also jagged and can be hard to traverse. The other surfaces look reasonable. All parameter spaces are relevant in the search of the minimum, and at some point $k_{b,1}$ and $k_{c,2}$ has to be varied, possibly simultaneously. In that case, the optimizer enters the jagged surfaces.

To find out whether the jaggedness was a matter of resolution or an inherent structure

[1]Note that the grids are very sensitive to change in their parameter values. The parameter search was computationally demanding, and the grids therefore had to be coarse. As a result, the optimalities are also coarse, and possibly perturbed from the global optima.

Table 4.4: The table presents the refined parameter space of the augmented parameter search. The dSSE, wdSSE and the mSSE were charted in this parameter space.

Bound	$k_{b,1}$	$k_{b,2}$	$k_{c,1}$	$k_{c,2}$
Lower	2.15×10^{-6}	1.29×10^{-4}	1.67×10^{-5}	1.30×10^{1}
Upper	1.0×10^{-3}	2.15×10^{-2}	1×10^{-3}	4×10^{3}

of the optimization formulations, the areas where the surfaces were sensitive to change were subjected to a refined parameter search. The grid resolution was enhanced to 15 grid points in each direction ($15^4 = 50625$ simulations in total), and the search space was contracted, removing areas of flat gradients. The refined search is viewed from all six combinations of parameters in Appendix A.3, and the search space was condensed from Table 4.2 to Table 4.4. By inspecting Figure A.28, Figure A.29 and Figure A.30, it was discovered that the jaggedness in Figure A.10, Figure A.11 and Figure A.12 was a matter of resolution, and that the true nature of the objective functions was s-shaped in the two-dimensional parameter space. Some of the simulations still failed to solve, due to small values of $k_{c,2}$ or large values of $k_{b,1}$. Decreasing $k_{c,2}$ produces increasingly larger droplets, and below a certain value, the distribution is pushed out of the radial domain ($r > R_m$). High values of $k_{b,1}$ increase the dynamics of breakage on magnitudes where the PBE was unable to be integrated in time. The refined optimal parameter table is given in Table 4.5.

Table 4.5: The optimal parameter combinations for the three different optimization formulations. The combinations are still crude, even though they were refined from Table 4.3. The objective function value is abbreviated OFV.

Method	OFV	$k_{b,1}$	$k_{b,2}$	$k_{c,1}$	$k_{c,2}$
dSSE	1.46×10^{6}	2.99×10^{-5}	5.56×10^{-4}	5.56×10^{-4}	2.28×10^{2}
wdSSE	1.06×10^{6}	7.19×10^{-5}	8.02×10^{-4}	7.47×10^{-4}	1.51×10^{2}
mSSE	7.09	2.68×10^{-4}	4.98×10^{-3}	7.20×10^{-5}	2.94×10^{1}

By inspecting Table 4.3 and Table 4.5, it is found that the refined search actually gave a worse objective function value (OFV) for the mSSE than the coarse grid, which may seem peculiar. The parameter values are most notably changed in the coalescence parameters, however the breakage parameters were also affected. The coalescence perspective is displayed in Figure 4.8. It is seen that the coalescence parameters that were previously optimal are now significantly suboptimal. This must be a consequence of the change in the breakage parameters from $k_{b,1} = 3.59 \times 10^{-4} \rightarrow k_{b,1} = 2.68 \times 10^{-4}$ and $k_{b,2} = 5.99 \times 10^{-3} \rightarrow k_{b,2} = 4.98 \times 10^{-3}$. The perturbation may seem small, but it gave a significant change in the OFV. From a different perspective, the breakage parameters were varied over the same coalescence parameters as for the coarse grid, see Figure 4.9. It is seen that the resolution is still insufficient, and the grid is still very much coarse. Since the exact same parameter combination is not available for the refined surface plots, the exact same OFV was not produced.

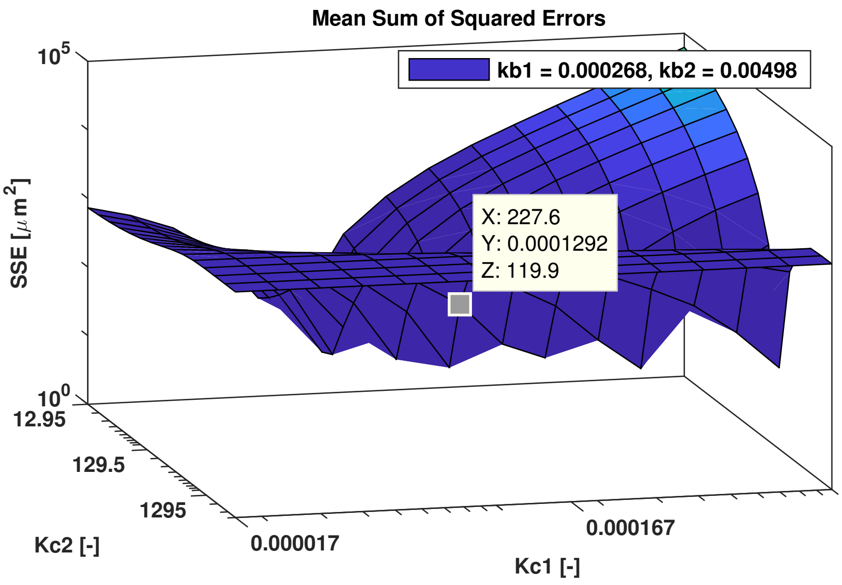


Figure 4.8: The figure is a surface plot of varying $k_{c,1}$ and $k_{c,2}$ with perturbed breakage parameters from the coarse grid. The objective function value has increased significantly.

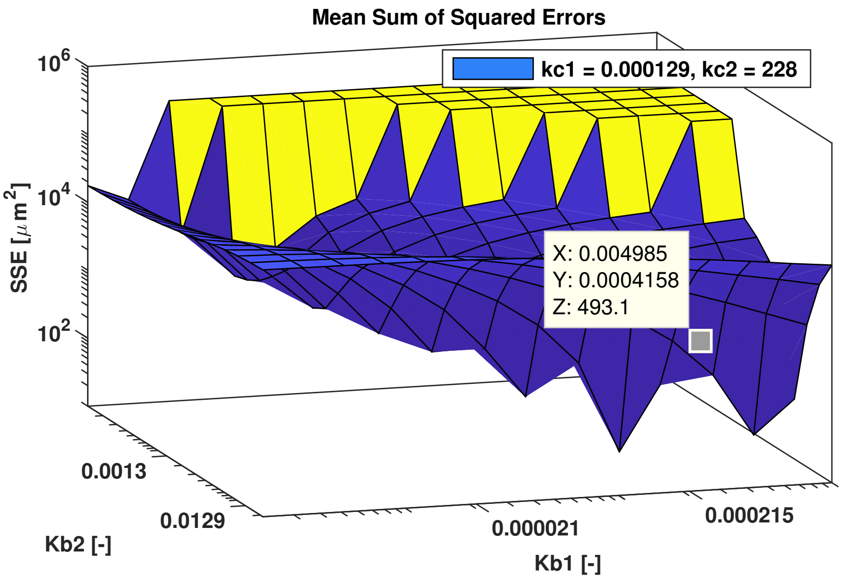


Figure 4.9: The figure is a surface plot of varying $k_{b,1}$ and $k_{b,2}$ with the same coalescence parameters as from the coarse grid. The objective function value has increased significantly.

4.3 Parameter Estimation

The optima from Table 4.3 and Table 4.5 were used as initial guesses in a trust-region optimization method, specifically Levenberg-Marquardt. The optimization was performed through the multifit non-linear driver of GSL with the methods `parameterEstimationSSE` and `parameterEstimationMean` in the user-defined `PBModel` class. The objective of the optimization was to identify the real global minimum of the optimization formulations, and more importantly, how many experimental distributions were necessary to stably converge to the same set of parameters. The number of distributions will pre-determine the minimum required number of measurement devices in a process design optimization, and is an important aspect of the current work.

The optimization routine employed used subroutines for scaling the parameters when they differed in their order of magnitude, such as for the system considered. The Moré damping strategy supposedly took care of rescaling the parameters so that the relative step size was approximately the same. However, employing the optimization routine without further considerations took the initial guess nowhere, and the same point was returned. The parameters were then manually rescaled to around unity, and the Jacobian step size was increased accordingly. The parameter search was then completed successfully and the parameters returned with their 95 % confidence intervals are presented in Table 4.6 and Table 4.7 for initial guesses from Table 4.3 and Table 4.5, respectively. The OFVs are presented in Table 4.8.

Table 4.6: The table displays the optimal parameters resulting from the parameter estimation. The initial guess provided was taken from Table 4.3.

Method	$k_{b,1}$	$k_{b,2}$	$k_{c,1}$	$k_{c,2}$
dSSE	$(3.90 \pm 0.03) \times 10^{-5}$	$(5.12 \pm 0.03) \times 10^{-4}$	$(9.89 \pm 0.03) \times 10^{-4}$	$(2.86 \pm 0.02) \times 10^{2}$
wdSSE	$(3.92 \pm 0.01) \times 10^{-5}$	$(5.80 \pm 0.02) \times 10^{-4}$	$(9.06 \pm 0.03) \times 10^{-4}$	$(2.76 \pm 0.01) \times 10^{2}$
mSSE	$(3.56 \pm 0.14) \times 10^{-4}$	$(6.13 \pm 0.14) \times 10^{-3}$	$(1.18 \pm 0.07) \times 10^{-4}$	$(2.29 \pm 0.10) \times 10^{2}$

Table 4.7: The table displays the optimal parameters resulting from the parameter estimation. The initial guess provided was taken from Table 4.5.

Method	$k_{b,1}$	$k_{b,2}$	$k_{c,1}$	$k_{c,2}$
dSSE	$(2.90 \pm 0.02) \times 10^{-5}$	$(5.62 \pm 0.02) \times 10^{-4}$	$(5.81 \pm 0.01) \times 10^{-4}$	$(2.50 \pm 0.01) \times 10^{2}$
wdSSE	$(5.30 \pm 0.04) \times 10^{-5}$	$(7.20 \pm 0.04) \times 10^{-4}$	$(7.93 \pm 0.05) \times 10^{-4}$	$(2.09 \pm 0.02) \times 10^{2}$
mSSE	$(2.48 \pm 0.09) \times 10^{-4}$	$(5.04 \pm 0.12) \times 10^{-3}$	$(6.28 \pm 0.53) \times 10^{-5}$	$(3.34 \pm 0.11) \times 10^{1}$

It is interesting to note that the OFV for the dSSE and wdSSE became better by refining the coarse grid, however, the opposite is true for the mSSE, see Table 4.3 and Table 4.5. By providing an initial guess with a lower OFV, the parameter estimation results became worse than if the initial guess with a higher OFV were to be provided, see Table 4.8. This is both true for the dSSE, the wdSSE and the mSSE. The initial guess from the coarse grid provided better results for both the dSSE and the wdSSE, and the initial guess from the refined grid provided better results for the mSSE, even though these initial guesses were arguably worse than their counterparts. The findings may be a result of the Jacobian step

Table 4.8: Objective function values for the parameter estimation with coarse and refined initial guesses.

Method	Coarse	Refined
dSSE	1.20×10^6	1.30×10^6
wdSSE	6.91×10^5	7.27×10^5
mSSE	5.17	4.28

size increase that was discussed previously. If the step size has been increased too much, the initial step might be too large, so that starting closer to the optimum might actually make the optimizer move past the optimum. It is also to be pointed out that breakage and coalescence are competing phenomena, and their highly non-linear behavior can be hard to estimate correctly, especially when their dynamical parameters grow large, which stiffens the PBE further. Small changes in these parameters can lead to significantly different behavior, as seen previously in Figure 4.9.

To check how sensitive the parameter estimation was to the initial guess, the optimal initial parameter combination for the dSSE in Table 4.3 was perturbed by a factor of 2 in each of the parameters. There are 14 different parameter combinations that were subject to perturbation, and they are all listed in Table 4.9. It is seen that the initial guess matters, and that the small perturbation of a factor 2 seems to move the optimal solution for all cases. None of the perturbed cases have converged to the non-perturbed case. Some of the perturbations resulted in solutions that were no longer even close to the nominal optimum. This is true for perturbing a) $k_{b,1}$ and $k_{c,2}$, and b) $k_{b,2}$ and $k_{c,1}$. The latter even gives non-sensical parameter values (negative). Perturbing $k_{b,2}$ alone even produced smaller values than the non-perturbed case, which is strange. It means that the value of $k_{b,2}$ has moved from a great value, past the non-perturbed optimal value, and to a smaller value. As seen before, the error surfaces twist in at least two dimensions, so it might mean that the other parameters were not at the point where the non-perturbed $k_{b,2}$ was optimal. In this case, the optimizer would move straight past the non-perturbed optimal $k_{b,2}$.

The results from the perturbation experiment may also indicate that the Jacobian step size taken was too large. There are many tunable internal parameters in the Levenberg-Marquardt optimization routine employed, and the Jacobian step size is only one of many. To confirm that the different optima found in Table 4.9 was not a matter of internal parameters, the parameter estimations should at least be attempted at lower Jacobian step sizes and function tolerances. This would ensure that the perturbations have indeed put the initial guess at a non-descending point. As previously seen from the surface plots, the objective function values vary several orders of magnitude over the parameter space. If the initial guesses were at non-descending points, they would most likely not converge to the same order of magnitude as the non-perturbed case. Since all optima found (with a few exceptions) are on the same order of magnitude, this may imply that they are in fact in the valley. The optimization routine is subject to internal optimization if this work is to be continued. This means that the internal parameters should be adjusted in order to ensure desirable convergence properties, and to ensure that the global minimum is found given a point of descension is provided. The experiment was neither repeated for the wdSSE, nor the mSSE.

Table 4.9: The table displays the effect of perturbing the coarse initial guess on the parameter estimation, and the sensitivity of the optimizer with respect to the initial guess. The values are raised to the power in the header of each column for readability.

Perturbation	OFV $[10^6]$	$k_{b,1}$ $[10^5]$	$k_{b,2}$ $[10^4]$	$k_{c,1}$ $[10^3]$	$k_{c,2}$ $[10^{-2}]$
$k_{b,1}$	1.22	4.25 ± 0.03	5.26 ± 0.03	1.09 ± 0.00	2.88 ± 0.02
$k_{b,1}$, $k_{b,2}$	1.28	5.12 ± 0.02	5.59 ± 0.03	1.14 ± 0.01	2.64 ± 0.01
$k_{b,1}$, $k_{b,2}$, $k_{c,1}$	1.20	4.23 ± 0.03	4.70 ± 0.02	1.30 ± 0.01	3.17 ± 0.01
$k_{b,1}$, $k_{b,2}$, $k_{c,2}$	1.21	3.89 ± 0.02	3.53 ± 0.02	1.76 ± 0.01	3.78 ± 0.01
$k_{b,1}$, $k_{c,1}$	1.22	4.49 ± 0.03	4.29 ± 0.01	1.60 ± 0.01	3.40 ± 0.01
$k_{b,1}$, $k_{c,1}$, $k_{c,2}$	1.30	5.03 ± 0.03	3.65 ± 0.02	2.26 ± 0.02	3.75 ± 0.02
$k_{b,1}$, $k_{c,2}$	56.3	12.4 ± 0.0	3.10 ± 0.00	1.33 ± 0.00	0.58 ± 0.00
$k_{b,2}$	1.18	3.44 ± 0.02	4.61 ± 0.02	1.07 ± 0.01	1.20 ± 0.01
$k_{b,2}$, $k_{c,1}$	37.0	18.2 ± 0.0	-0.65 ± 0.00	0.93 ± 0.00	-1.11 ± 0.00
$k_{b,2}$, $k_{c,1}$, $k_{c,2}$	1.23	4.46 ± 0.02	4.06 ± 0.02	1.71 ± 0.01	3.51 ± 0.01
$k_{b,2}$, $k_{c,2}$	1.21	4.34 ± 0.02	4.33 ± 0.02	1.53 ± 0.01	3.39 ± 0.01
$k_{c,1}$	1.26	5.07 ± 0.02	4.89 ± 0.02	1.37 ± 0.00	2.92 ± 0.01
$k_{c,1}$, $k_{c,2}$	1.28	5.31 ± 0.03	4.76 ± 0.02	1.63 ± 0.01	3.19 ± 0.02
$k_{c,2}$	1.19	3.50 ± 0.02	3.74 ± 0.02	1.52 ± 0.01	3.74 ± 0.01
No perturb	1.20	3.90 ± 0.03	5.12 ± 0.03	0.99 ± 0.00	2.86 ± 0.02

Up until now, all distributions have been used in all objective function formulations, i.e. the residuals of 89 distributions with 80 measurements in each have been used. By using only some of these 89 distributions, the parameters could hopefully stabilize for some minimum number of distributions. The costly measurement devices could then be reduced. The last distribution should always be used because this is the distribution that would enter downstream units. Therefore, this distribution can not be omitted from the objective function formulation. For N distributions, the $N-1$ first distributions would be used to get the dynamics right and the N-th distribution would be included last. The resulting parameter combinations and their OFVs and OFVs per degree of freedom are presented as a function of N distributions chosen in Figure 4.10, Figure 4.11, Figure 4.12 and Figure 4.13 for the coarse and refined initial guesses for dSSE and wdSSE, respectively. In the figures, $|J|$ denotes the square root of the OFV, p is the number of parameters, and $n-p$ denotes the degrees of freedom. For the dSSE, the values of $k_{b,1}$, $k_{c,1}$, $k_{c,2}$ and the OFV per degree of freedom seems to stabilize more or less at approximately 30 distributions for both the coarse and refined initial guess. The values of $k_{b,2}$ does not seem to stabilize, and it can be explained by the fact that including more distributions puts more weight on the last distributions. From the experimental data, it is clear that the distributions shift towards smaller droplets, and from the work prior to the coalescence experiment described earlier, it was discovered that $k_{b,2}$ controls which droplets are allowed to break. For smaller values of $k_{b,2}$, the critical radius – which is the radius where breakage starts to fade out – is shifted towards smaller radii. This means that when more weight is put on distributions at smaller radii, the values of $k_{b,2}$ also decrease. Therefore the value does not seem to stabilize. The number of distributions that seem to stabilize for the rest of the parameters are hence 30 distributions.

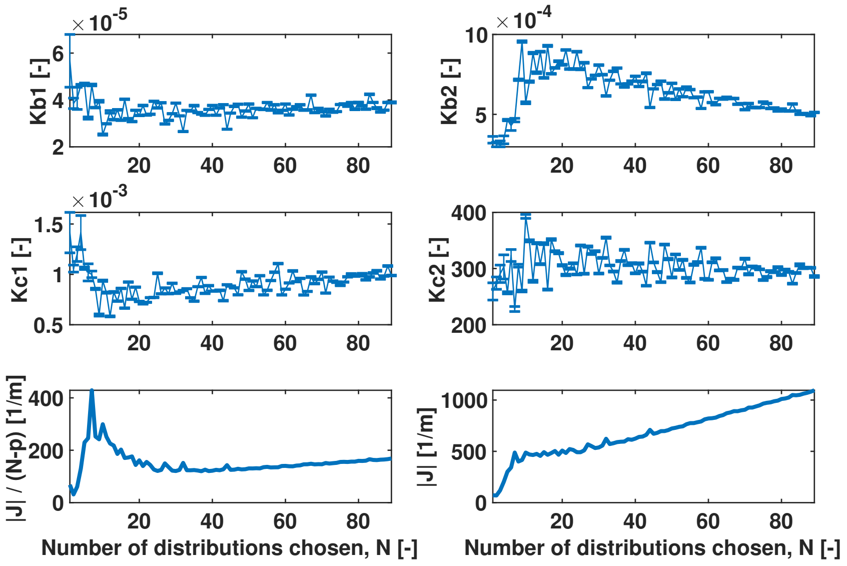


Figure 4.10: Parameters, square root of the dSSE OFV per degree of freedom and square root of the dSSE OFV as function of distributions chosen. The experiment is for initial guesses from the coarse grid.

For the weighted case, the OFV per degree of freedom stabilizes earlier, even as early as 20 distributions for the refined case. However all parameters produce oscillatory behaviors for the coarse initial guess, and $k_{b,1}$, $k_{b,2}$ and $k_{c,2}$ produce oscillatory behaviors for the refined initial guess. From the perturbation experiment, it was learned that perturbations as small as a factor of 2 might have a significant impact on the optimal solution. From the smallest parameter value produced to the largest parameter value produced, there seems to be an offset by a factor of 2 for most parameters. The oscillatory behavior produced from the wdSSE is therefore unsatisfactory, as it brings uncertainty along to the data fitting.

It was earlier mentioned that the perturbation experiment is ambiguous. This ambiguity questions the validity of the optimum from the parameter estimations, and whether it is the global optimum or not. Consequentially, the minimization of the number of measurement devices is also questioned by this uncertainty. If the internal parameters are to be optimized, the perturbation experiment should be repeated in order to ensure that the convergence is consistent given the initial guess is still at a point of descension. For internal parameters ensuring consistent convergence, the minimization of the number of measurement devices should then be repeated to verify the results.

The best fit, i.e. the lowest OFV, that the Levenberg-Marquardt algorithm was able to provide is shown for the dynamical behavior and the steady state for all objective function formulations in Figure 4.14 and Figure 4.15, respectively. It is clear that the optimal parameter combination retrieved from the dSSE and the wdSSE are very similar. The dynamical behavior of the two regression approaches are nearly identical, and the same goes for the steady state location. The benefits from doing a weighted SSE are therefore

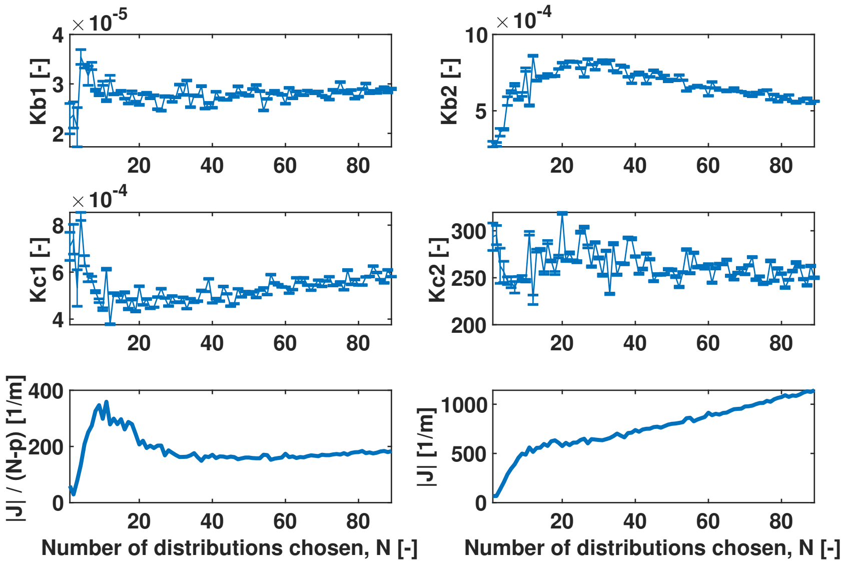


Figure 4.11: Parameters, square root of the dSSE OFV per degree of freedom and square root of the dSSE OFV as function of distributions chosen. The experiment is for initial guesses from the refined grid.

marginal, and from the previous findings, it has, on the contrary, only had negative effects on the parameter estimation: the valleys from the surfaces became tighter and shallower, and the fluctuations in the parameters are larger. Using the weighted SSE for parameter estimation on the PBE in future work is therefore discouraged.

It is noteworthy that all optimal parameter combinations retrieved has reached a steady state. The dSSE and wdSSE widen initially, pushing the peak down, before they tighten and their peaks are pushed to their steady state location. The findings contradict the experimental data. The mSSE, on the other hand, tightens up and pushes the peak upwards, but it fails to come down again, for reasons discussed earlier: the small droplets are unaccounted for. However, from Figure 4.15 it is seen that it is not nearly wide enough to fit the experimental steady state. All regression approaches seem to keep the pronounced edge at the right tail coming from coalescence birth, which is desirable.

It should also be emphasized that the dynamics are very quick for all formulations, causing them all to reach steady state quickly. Most of the experimental measurements were approximately at steady state, meaning that the sum of squared errors formulations (all of them) weight this steady state behavior heavily. The optimizer therefore made parameter combinations that would get to this state as fast as possible, disregarding the dynamical evolution. The steady state solution, however, is fairly good. As noted before, the smaller droplets could not be accounted for, and as a result, the value of the peak was also unaccounted for. The mode seems approximately correct, which is what was aimed for.

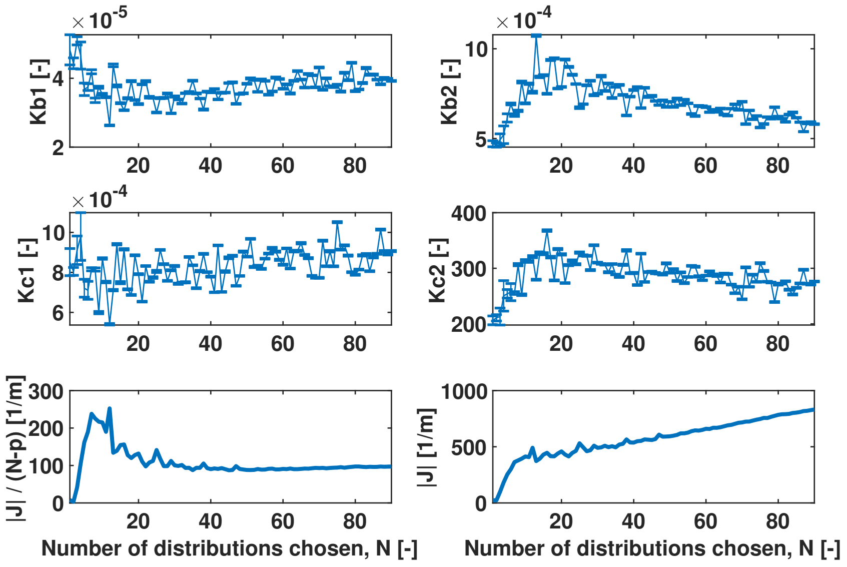


Figure 4.12: Parameters, square root of the wdSSE OFV per degree of freedom and square root of the wdSSE OFV as function of distributions chosen. The experiment is for initial guesses from the coarse grid.

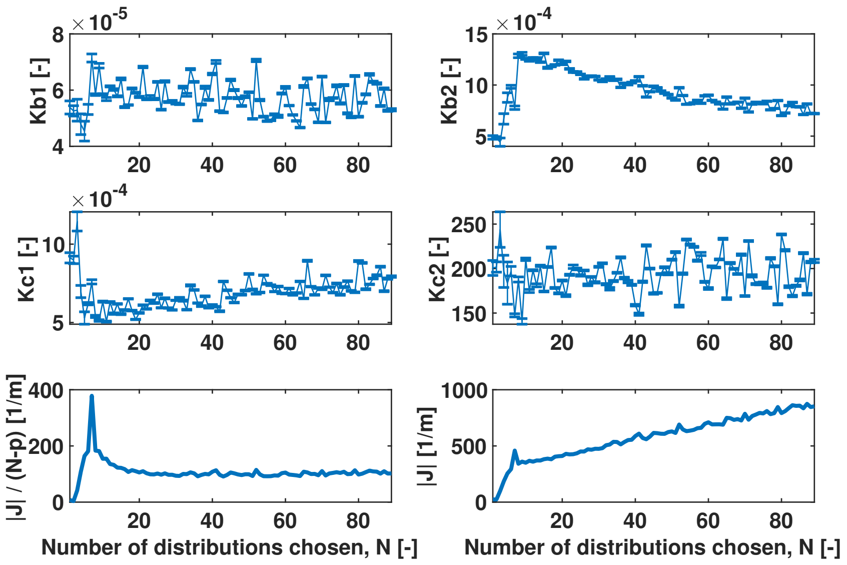


Figure 4.13: Parameters, square root of the wdSSE OFV per degree of freedom and square root of the wdSSE OFV as function of distributions chosen. The experiment is for initial guesses from the refined grid.

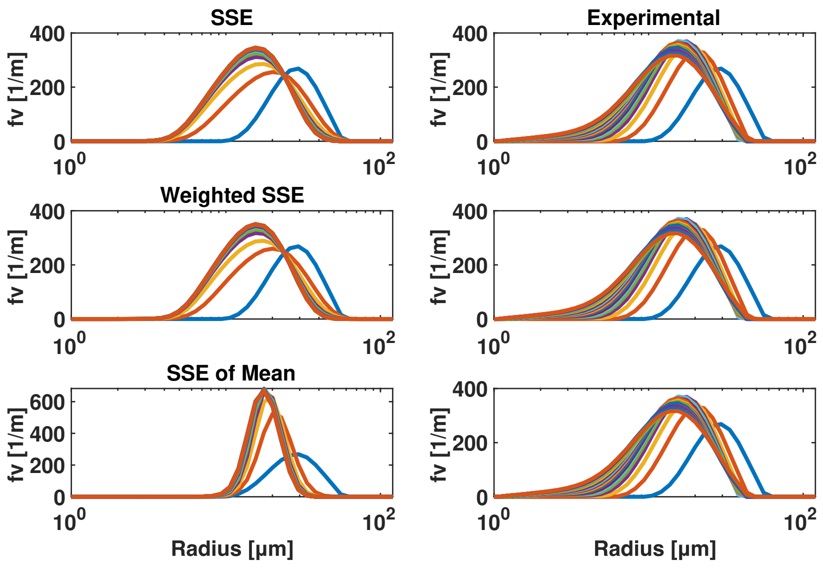


Figure 4.14: The figure depicts the produced dynamical behavior of the different optimal parameter combinations.

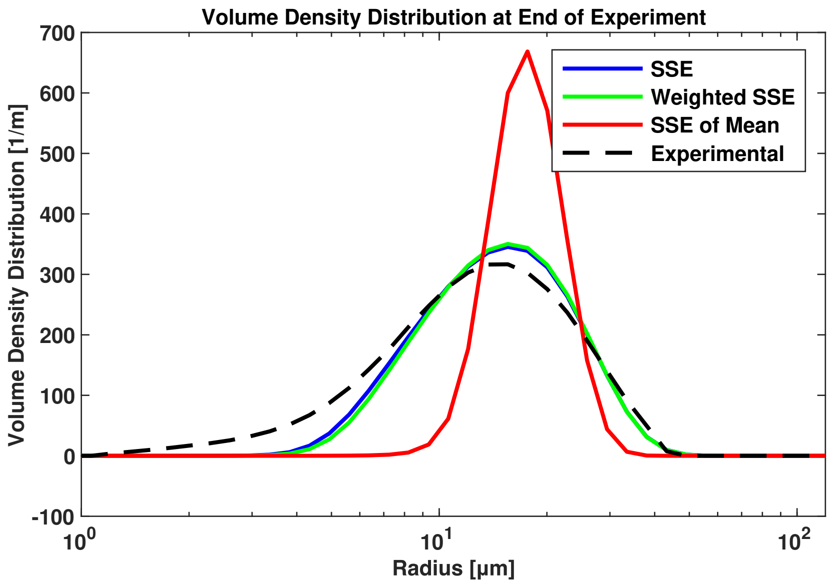


Figure 4.15: The figure depicts the produced steady state location of the different optimal parameter combinations.

The means of the distributions are shown in Figure 4.16. It is there seen that as a consequence of getting a good fit on the dynamical and steady state behavior, the dSSE and wdSSE also got a good fit on the temporal evolution of the mean. The objective function formulation that aimed at fitting this value alone, the mSSE, has of course reached a better fit. This fit is as close to the experimental data as can be, lying almost exactly on top of the measurements. The mSSE produced a satisfactory fit for what was aimed for, however the secondary output (the SSE of the distribution) was unsatisfactory. Since fitting the regular dSSE automatically made the mean approximately fit, this regression approach fits both objective functions automatically. Fitting the dSSE is therefore encouraged, and the weighted SSE (wdSSE) along with the SSE of the mean (mSSE) objective function formulations are discouraged.

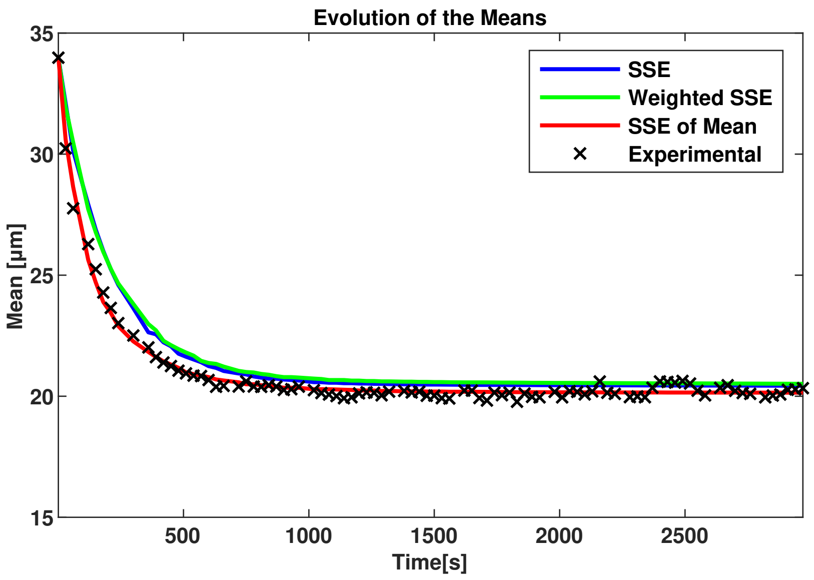


Figure 4.16: The figure depicts the dynamical behavior of the mean for the different optimal parameter combinations.

Finally, it will be mentioned that the parameter values are, at most, five orders of magnitude off unity, which is approximately the value used in the literature for the kernels used [31, 32, 33, 34]. The parameters are correction factors for uncertainties related to the a) modeling and b) turbulent energy dissipation rate, ε. The values being so far from unity may imply there has been an overestimation of ε. The true value of ε may therefore be smaller than the one calculated by (3.20). To get a better estimate of this quantity, the zero-dimensional (physical space) model could be revised to a three-dimensional model, and computational fluid dynamics could be employed to get a better understanding of its spatial variations. It may also be interesting to try and account for multiple droplets produced from breakage events, i.e. $\nu > 2$, or/and multiple droplets consumed from coalescence events, i.e. $\delta > 2$. Third and last, the effect of the centrifugal pump used in the experimental setup on ε is unknown and could be explored further. For instance, the turbine used to stir the emulsion could be switched off and the Reynolds number could be measured with and without the turbine. This would give an indication on how the pump affects the turbulence in the system, and also if there is any breakage occurring outside the tank, which of course is unaccounted for by the model.

Chapter 5

Conclusion

A zero-dimensional volumetric population balance equation was employed to calibrate four parameters to experimentally measured and rescaled volume density distributions. Three regression approaches were attempted: sum of squared errors for the distribution (dSSE), weighted sum of squared errors for the distribution (wdSSE) and sum of squared errors of the mean (mSSE), all of which resulted in non-linear regression. The non-convexity inherently present in a non-linear regression formulation proved difficult as far as initial guesses go, and an appropriate parameter space was charted, attempting to provide a sufficiently good initial guess. Due to high computational burden, the charting was done with a low resolution, making the surfaces look jagged. The surfaces were refined and the resolution was improved. Suspicions about flat gradients and several local optima were confirmed for both the coarse and the refined grid. The wdSSE looked similar to the normal dSSE with the exception of being narrower and shallower. The best values received from the charting for both the coarse and the refined grid were provided as initial guesses to a Levenberg-Marquardt non-linear regression routine implemented in the GNU Scientific Library, and optimal parameter combinations were found for all regression strategies.

To further check how sensitive the parameter estimation was with respect to the initial guess, the initial guess was perturbed by a factor of 2. That is, all possible combinations of parameters were multiplied by 2, one at a time, resulting in 14 different perturbations. Some of the perturbations gave a significant change (an order of magnitude) in the objective function value. The other cases resulted in optima with objective function values of the same order of magnitude. Since the surface plots vary several orders of magnitude over the parameter space, this may imply that most of the perturbed initial guesses were still at points of descension. The optimization routine employed is gradient-based, and most of the perturbations should therefore have resulted in the same global optimum. This was not the case, and the results are inconclusive.

Since experimental measurement devices are expensive, it was attempted to find the minimum number of measurements necessary for the parameter combination to remain the same, even for increasing number of measurements. Using the $N-1$ first measurements and the final N-th measurement, parameter estimations were performed on a corresponding objective function. The dSSE and the wdSSE displayed different behaviors. The dSSE stabilized at 30 distributions with the exception of the parameter $k_{b,2}$, which decreased for more distributions. The objective function value per degree of freedom for the wdSSE stabilized at 20 distributions, however, $k_{b,1}$, $k_{b,2}$ and $k_{c,2}$ were oscillating. For this reason it is hard to draw a conclusion for the wdSSE. It is still uncertain whether the oscillations are connected to the internal parameters of the optimization routine employed or not. The fact that $k_{b,2}$ decreased with increasing N is due to the increase in the number of steady state distributions chosen. As a result of the higher emphasis on the steady state, more breakage, and hence lower $k_{b,2}$, is favored.

Finally, the lowest objective function value produced and its corresponding parameter combination for all regression approaches were used to confirm that the fit was truly a good fit. The dSSE and the wdSSE had very similar behavior, both with respect to dynamics, steady state location and the temporal evolution of the mean. The dynamics were too quick, however, they did settle approximately at the state of the final experimental distribution. The small droplets were unaccounted for due to a) spatially varying turbulent energy dissipation rate, b) several fragments born due to breakage, c) contribution from the centrifugal pump that is unaccounted for in the model formulation, or d) a combination of the factors just mentioned. As a result, the value of the peak is also off, however, the mode of the distribution and the pronounced right edge from coalescence fit the experimental one. They also had a satisfactory fit on the temporal evolution of the mean. Fitting the mSSE had similar dynamical behavior to the experimental distribution, however, this regression approach also had quicker dynamics than what was experimentally observed. The steady state location is also off, and it is too narrow. Being the secondary output, the dynamics and steady state was not expected to fit. The primary output, being the mean, had an exceptional fit, lying on top of the measurements. However, since fitting the dSSE had better dynamics and steady state location, while also fitting the secondary output (the mean) fairly well, this is the preferred regression approach. The wdSSE was inferior to the dSSE in every way: the residual surfaces were narrower, shallower and harder to navigate, the parameter combinations oscillate, and the actual fit was slightly worse. With all these discoveries, the preferred regression strategy is the dSSE, as it fits the distributions, and as a consequence it also fits the mean. Fitting the weighted SSE (wdSSE) or the SSE of the mean (mSSE) is discouraged.

5.1 Further Work

For future work on this topic, the author would recommend to a) do more rigorous calculations on the turbulent energy dissipation rate, b) do experiments without the agitator to see how the centrifugal pump affects the results, and possibly c) extend the model to account for multiple droplets produced from breakage events, i.e. $\nu > 2$. a) may include extending the model to three dimensions in physical space and to do computational fluid dynamics to extract information on the the turbulent energy dissipation rate. If the model still is to maintain its simplicity, this variance can then be spatially averaged. b) will isolate if there are any breakage events occurring inside the pump, or if the turbulent energy dissipation rate is disturbed by the pump. c) will allow the model to produce many small droplets for each breakage event, potentially recreating the observations.

The perturbation experiment was ambiguous and inconclusive. Therefore the author would recommend to internally optimize the tunable parameters of the Levenberg-Marquardt routine from the GNU Scientific Library. This includes adjusting the Jacobian step size and the function tolerance in order to ensure desirable convergence properties, and to ensure that the global minimum is found consistently given that a point of descension is provided. The results from the perturbation experiment and the minimization of the number of measurement devices would consequentially have to be revised by repeating the experiments.

Finally, a multi-objective function may be formulated to include the sum of squared errors from the distribution, its mode and maybe also several of the statistical moments. It may also be an option to formulate a multi-objective function that fits the sum of squared errors from the volumetric *and* number density distribution. Since the number density distribution would emphasize the smaller droplets even more, this objective function formulation would require more information about the behaviors of the tail for small droplets, whether it can be recreated theoretically or if it is the pump breaking the droplets.

Bibliography

[1] D. Ramkrishna, *Population Balances: Theory and Applications to Particulate Systems in Engineering*. Elsevier Science, 2000.

[2] C. J. Backi and S. Skogestad, "A simple dynamic gravity separator model for separation efficiency evaluation incorporating level and pressure control," *2017 Proceedings of the American Control Conference (ACC)*, pp. 2823–2828, 2017.

[3] M. Renardy, *Numerical Methods*. Society for Industrial and Applied Mathematics, 2000, no. 6.

[4] C. Hirsch, *Iterative Methods for the Resolution of Algebraic Systems-Chapter 10*. Elsevier Ltd, 2007.

[5] J. E. Dennis, "Numerical methods for unconstrained optimization and nonlinear equations," Philadelphia, 1996.

[6] J. H. Ferziger, *Computational methods for fluid dynamics*, 3rd ed., M. Perić, Ed. Berlin: Springer, 2002.

[7] A. H. D. Cheng and D. T. Cheng, "Heritage and early history of the boundary element method," *Engineering Analysis with Boundary Elements*, vol. 29, no. 3, pp. 268–302, 2005.

[8] D. Gibb, "A Course in Interpolation and Numerical Integration for the Mathematical Laboratory (Edinburgh Mathematical Tracts, No. 2) (Book Review)," pp. 67–68, 1916.

[9] S. R. Otto, "An Introduction to Programming and Numerical Methods in MATLAB," 2005.

[10] H. A. Jakobsen, *Chemical Reactor Modeling: Multiphase Reactive Flows*. Cham: Springer International Publishing: Cham, 2014.

[11] E. Kreyszig, *Advanced engineering mathematics*, 9th ed. Hoboken, N.J: Wiley, 2006.

[12] G. Szegö, *Orthogonal polynomials*, ser. Colloquium publications. New York: American Mathematical Society, 1939, vol. 23.

[13] J. Solsvik, S. Tangen, and H. A. Jakobsen, "Evaluation of weighted residual methods for the solution of the pellet equations: The orthogonal collocation, Galerkin, tau and least-squares methods," *Computers and Chemical Engineering*, vol. 58, p. 223, 2013.

[14] J. Villadsen, *Solution of differential equation models by polynomial approximation*, ser. Prentice-Hall international series in the physical and chemical engineering sciences, M. L. Michelsen, Ed. Englewood Cliffs, N.J: Prentice-Hall, 1978.

[15] A. D. Randolph, *Theory of particulate processes : analysis and techniques of continuous crystallization*, 2nd ed., M. A. Larson, Ed. San Diego: Academic Press, 1988.

[16] M. Kostoglou and A. J. Karabelas, "On sectional techniques for the solution of the breakage equation," *Computers and Chemical Engineering*, vol. 33, no. 1, pp. 112–121, 2009.

[17] M. Vanni, "Approximate Population Balance Equations for Aggregation-Breakage Processes," *Journal of colloid and interface science*, vol. 221, no. 2, p. 143, 2000.

[18] S. Kumar and D. Ramkrishna, "On the solution of population balance equations by discretization—II. A moving pivot technique," *Chemical Engineering Science*, vol. 51, no. 8, pp. 1333–1342, 4 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/000925099500355X

[19] ——, "On the solution of population balance equations by discretization—I. A fixed pivot technique," *Chemical Engineering Science*, vol. 51, no. 8, pp. 1311–1332, 4 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0009250996884892

[20] J. Kumar and G. Warnecke, "Convergence analysis of sectional methods for solving breakage population balance equations-I: the fixed pivot technique," *Numerische Mathematik*, vol. 111, no. 1, pp. 81–108, 2008.

[21] S. V. Patankar, *Numerical heat transfer and fluid flow*, ser. Series in computational methods in mechanics and thermal sciences. Washington: Hemisphere Publ., 1980.

[22] R. W. Johnson, *Handbook of Fluid Dynamics*, 2nd ed. Taylor & Francis Group, CRC Press, 2016.

[23] T. M. Cover and J. A. Thomas, *Maximum Entropy*. Hoboken, NJ, USA: Hoboken, NJ, USA: John Wiley & Sons, Inc., 2005.

[24] A. D. Randolph, *Theory of particulate processes : analysis and techniques of continuous crystallization*, M. A. Larson, Ed. New York: Academic Press, 1971.

[25] H. Hulburt and S. Katz, “Some problems in particle technology: A statistical mechanical formulation,” *Chemical Engineering Science*, vol. 19, no. 8, pp. 555–574, 8 1964. [Online]. Available: https://www.sciencedirect.com/science/article/pii/0009250964850478

[26] R. Mcgraw, “Description of aerosol dynamics by the quadrature method of moments,” *Aerosol Sci. Technol.*, vol. 27, no. 2, pp. 255–265, 1997.

[27] D. L. Marchisio and R. O. Fox, “Solution of population balance equations using the direct quadrature method of moments,” *Journal of Aerosol Science*, vol. 36, no. 1, pp. 43–73, 1 2005. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0021850204003052

[28] R. Fan, D. L. Marchisio, and R. O. Fox, “Application of the direct quadrature method of moments to polydisperse gas–solid fluidized beds,” *Powder Technology*, vol. 139, no. 1, pp. 7–20, 1 2004. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0032591003002997

[29] S. Rigopoulos, “Population balance modelling of polydispersed particles in reactive flows,” *Progress in Energy and Combustion Science*, vol. 36, no. 4, pp. 412–443, 8 2010. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0360128509000690

[30] S. Schmideder, C. Kirse, J. Hofinger, S. Rollié, and H. Briesen, “Separation of microorganisms in bioprocesses by flotation: effect of dispersities,” in *Population Balance Modeling Conference*, Ghent, Belgium, 2018.

[31] C. A. Coulaloglou and L. L. Tavlarides, “Description of interaction processes in agitated liquid-liquid dispersions,” *Chemical Engineering Science*, vol. 32, no. 11, pp. 1289–1297, 1977.

[32] N. Vankova, S. Tcholakova, N. D. Denkov, V. D. Vulchev, and T. Danner, “Emulsification in turbulent flow: 2. Breakage rate constants,” *Journal of Colloid And Interface Science*, vol. 313, no. 2, pp. 612–629, 2007.

[33] M. J. Prince and H. W. Blanch, “Bubble coalescence and break-up in air-sparged bubble columns,” *AIChE Journal*, vol. 36, no. 10, pp. 1485–1499, 1990.

[34] A. Chesters, “THE MODELING OF COALESCENCE PROCESSES IN FLUID LIQUID DISPERSIONS - A REVIEW OF CURRENT UNDERSTANDING,” *Chem. Eng. Res. Des.*, vol. 69, no. 4, pp. 259–270, 1991.

[35] S. L. Ross and R. L. Curl, “Measurement and models of the dispersed phase mixing process,” in *Joint Chem Eng Conf, Paper 29b, Symposium Series 139*, Vancouver, Canada, 1973.

[36] C. Tsouris and L. L. Tavlarides, “Breakage and coalescence models for drops in turbulent dispersions,” *AIChE Journal*, vol. 40, no. 3, pp. 395–406, 1994.

[37] C. Martínez-Bazán, J. L. Montañés, and J. C. Lasheras, "On the breakup of an air bubble injected into a fully developed turbulent flow. Part 1. Breakup frequency," *Journal of Fluid Mechanics*, vol. 401, pp. 157–182, 1999.

[38] V. Alopaeus, M. Laakkonen, and J. Aittamaa, "Numerical solution of moment-transformed population balance equation with fixed quadrature points," *Chemical Engineering Science*, vol. 61, no. 15, pp. 4919–4929, 2006.

[39] M. Petitti, M. Vanni, D. L. Marchisio, A. Buffo, and F. Podenzani, "Simulation of coalescence, break-up and mass transfer in a gas–liquid stirred tank with CQMOM," *Chemical Engineering Journal*, vol. 228, pp. 1182–1194, 2013.

[40] C. Martinez-Bazan, J. Montanes, and J. Lasheras, "On the breakup of an air bubble injected into a fully developed turbulent flow. Part 2. Size PDF of the resulting daughter bubbles," *J. Fluid Mech.*, vol. 401, pp. 183–207, 1999.

[41] P. Chen, J. Sanyal, and M. P. Duduković, "Numerical simulation of bubble columns flows: effect of different breakup and coalescence closures," *Chemical Engineering Science*, vol. 60, no. 4, pp. 1085–1101, 2005.

[42] G. Kocamustafaogullari and M. Ishii, "Foundation of the interfacial area transport equation and its closure relations," *International Journal of Heat and Mass Transfer*, vol. 38, no. 3, pp. 481–493, 1995.

[43] J. Solsvik and H. A. Jakobsen, "Single drop breakup experiments in stirred liquid–liquid tank," *Chemical Engineering Science*, vol. 131, pp. 219–234, 2015.

[44] J. Solsvik and H. Jakobsen, "Solution of the Pellet Equation by use of the Orthogonal Collocation and Least Squares Methods: Effects of Different Orthogonal Jacobi Polynomials," *International Journal of Chemical Reactor Engineering*, vol. 10, no. 1, 2012.

[45] S. K. Moon, "Experimental investigation of droplet breakage in the oil-in-water emulsion in a stirred tank," Trondheim, 2018.

[46] A. C. Hindmarsh, P. N. Brown, K. E. Grant, S. L. Lee, R. Serban, D. E. Shumaker, and C. S. Woodward, "{SUNDIALS}: Suite of nonlinear and differential/algebraic equation solvers," *ACM Transactions on Mathematical Software (TOMS)*, vol. 31, no. 3, pp. 363–396, 2005.

[47] M. Galassi, *GNU Scientific Library Reference Manual*, 3rd ed. [Online]. Available: http://www.gnu.org/software/gsl/

[48] B. Adams, L. Bauman, W. Bohnhoff, K. Dalbey, M. Ebeida, J. Eddy, M. Eldred, P. Hough, K. Hu, J. Jakeman, J. Stephens, L. Swiler, D. Vigil, and T. Wildey, *Dakota, A Multilevel Parallel Object-Oriented Framework for Design Optimization, Parameter Estimation, Uncertainty Quantification, and Sensitivity Analysis: Version 6.0 User's Manual*, 6th ed. Sandia Technical Report SAND2014-4633, 2017.

[49] J. Nocedal, "Numerical Optimization," 2006.

Appendix A

Auxiliary Figures

A.1 Charting Coalescence Experiment

The optimal parameter combinations found for the coalescence experiment produced the dynamical behaviors, steady state locations and temporal evolutions of the mean seen in Figure 4.14, Figure A.2 and Figure A.3, respectively.

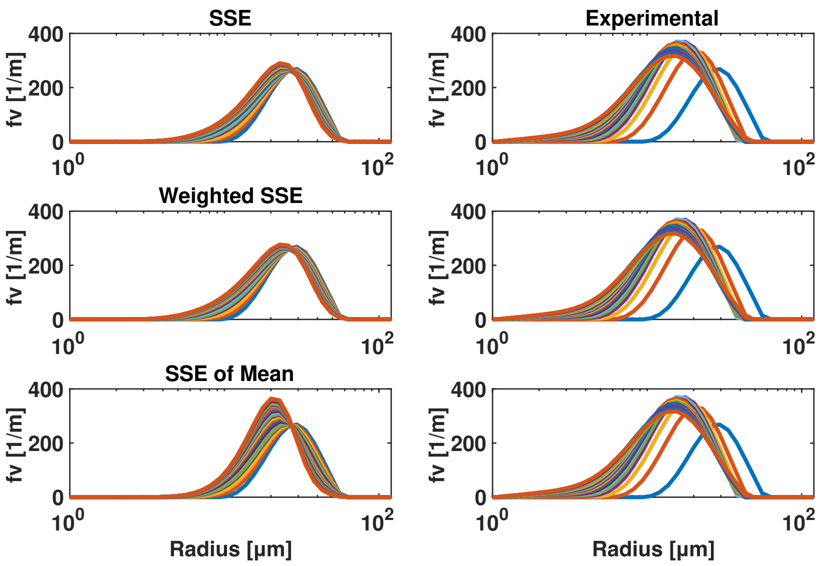


Figure A.1: The figure depicts the produced dynamical behaviors for the different optimal parameter combinations received from the coalescence experiment.

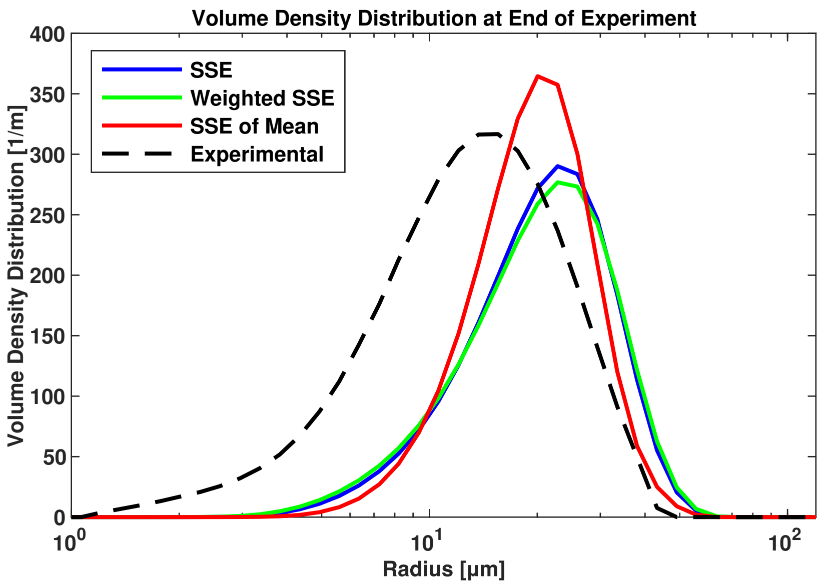


Figure A.2: The figure depicts the produced steady state locations for the different optimal parameter combinations received from the coalescence experiment.

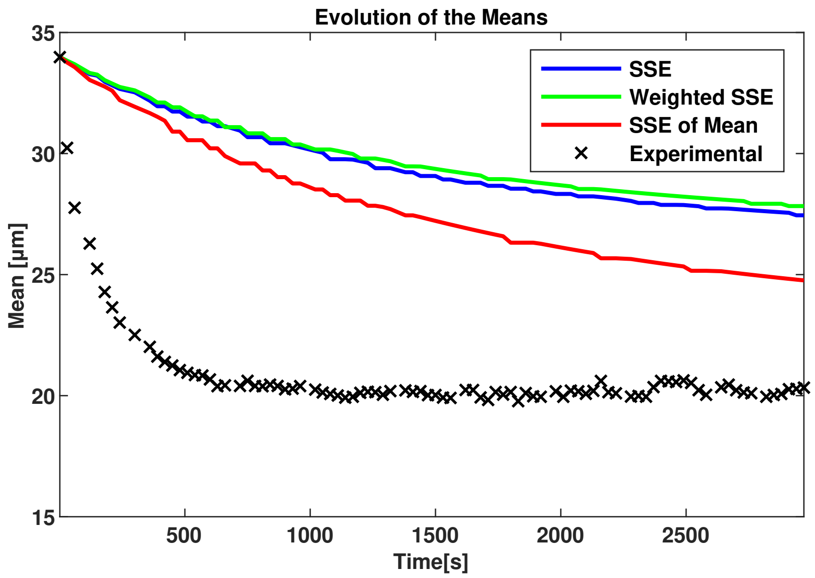


Figure A.3: The figure depicts the produced temporal evolutions of the mean for the different optimal parameter combinations received from the coalescence experiment.

A.2 Augmented Parameter Search, Coarse

The augmented parameter search can be viewed in $\frac{4!}{2!(4-2)!} = 6$ ways, since the order of changing the parameters does not matter. The different combinations are shown below in their own subsection

A.2.1 Varying $k_{b,1}$ and $k_{b,2}$

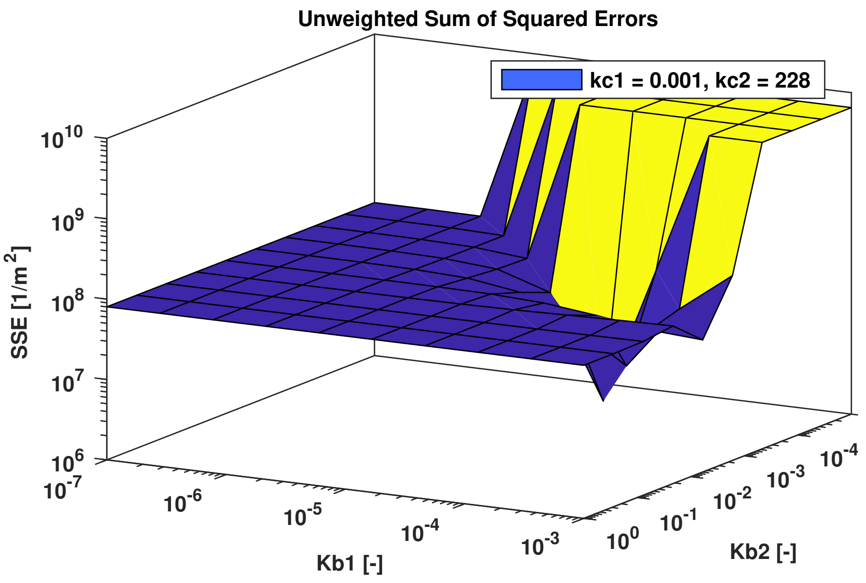


Figure A.4: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE.

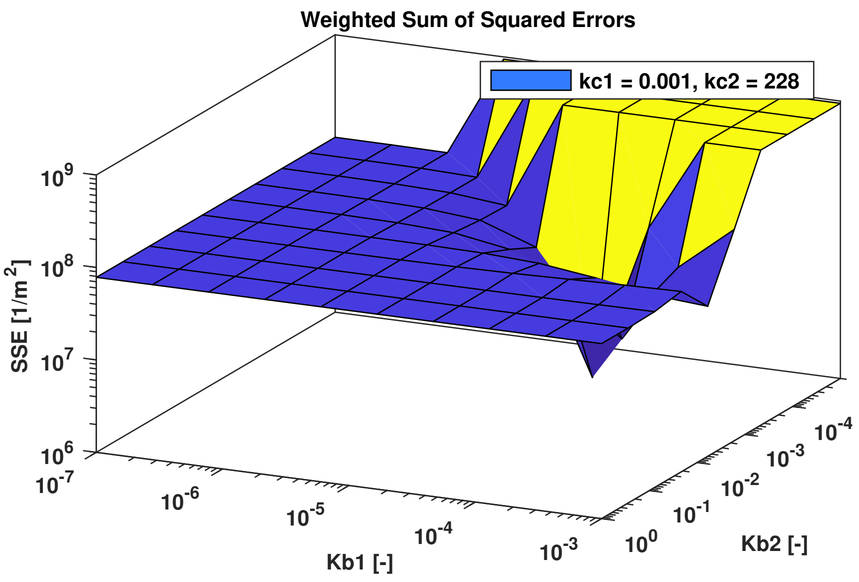


Figure A.5: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the weighted SSE.

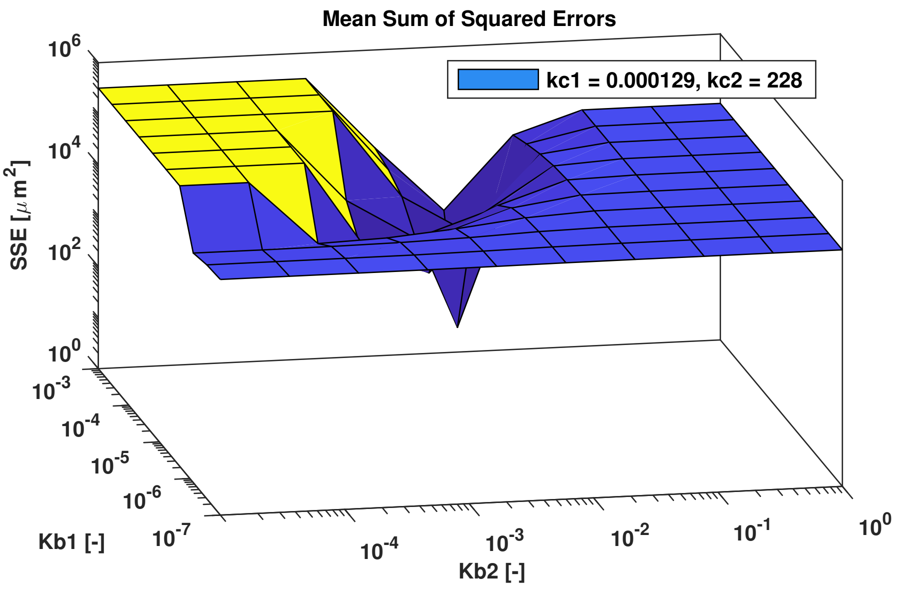


Figure A.6: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE of the mean.

A.2.2 Varying $k_{b,1}$ and $k_{c,1}$

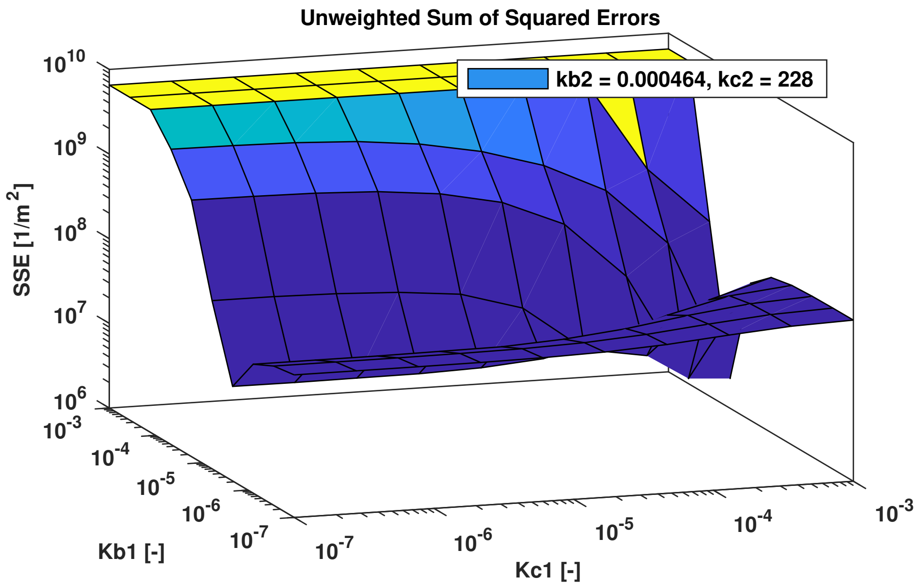


Figure A.7: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE.

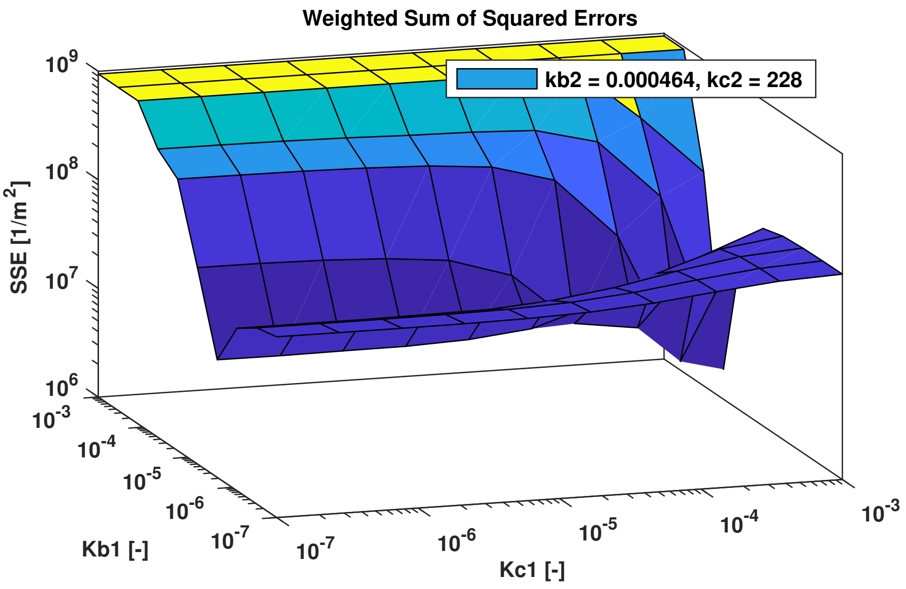


Figure A.8: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the weighted SSE.

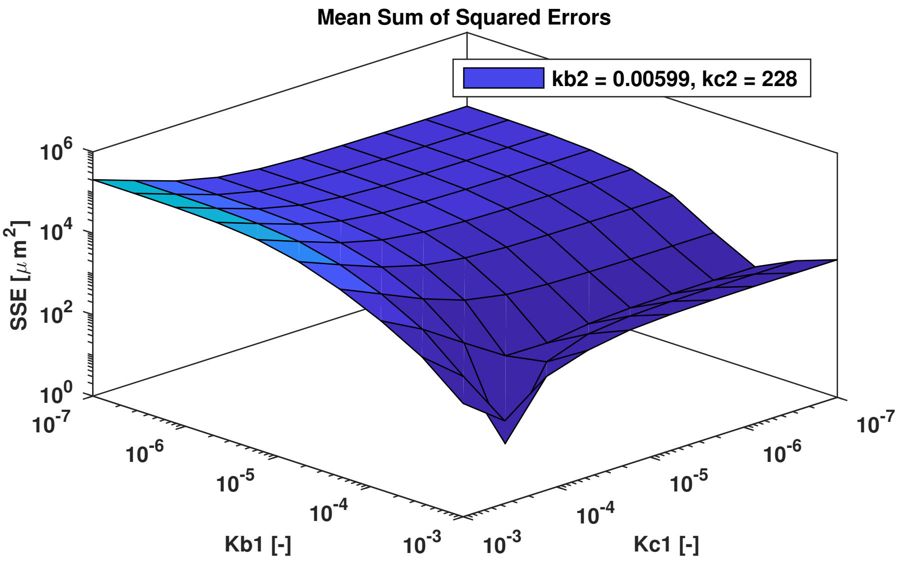


Figure A.9: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE of the mean.

A.2.3 Varying $k_{b,1}$ and $k_{c,2}$

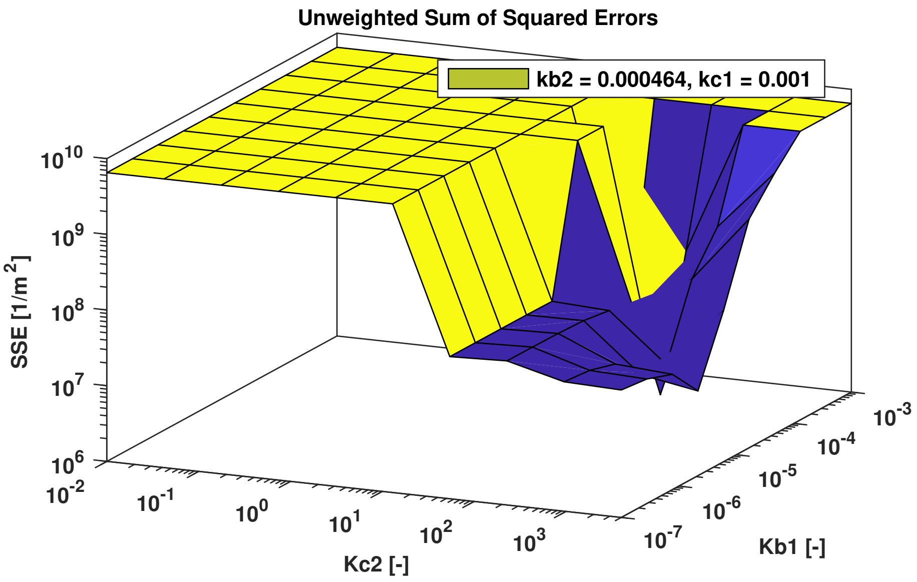


Figure A.10: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE.

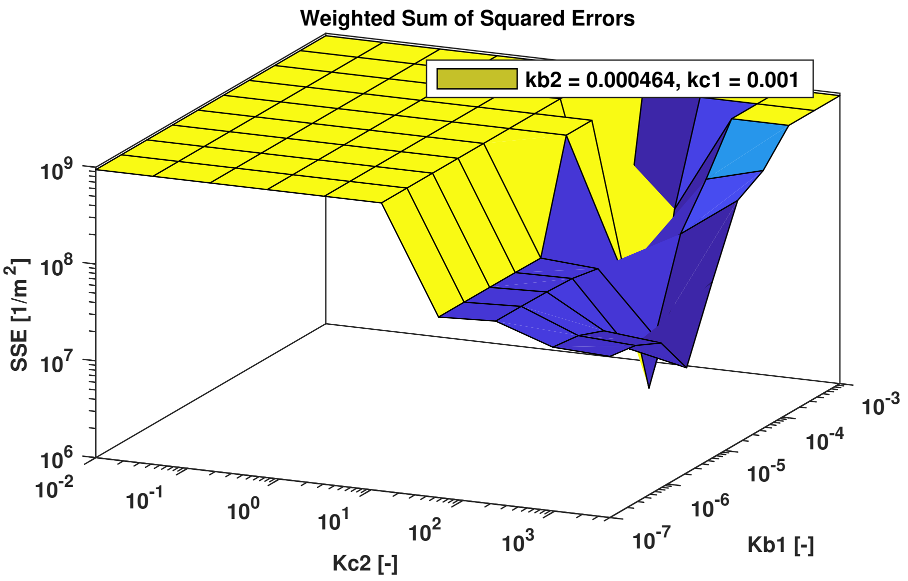


Figure A.11: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the weighted SSE.

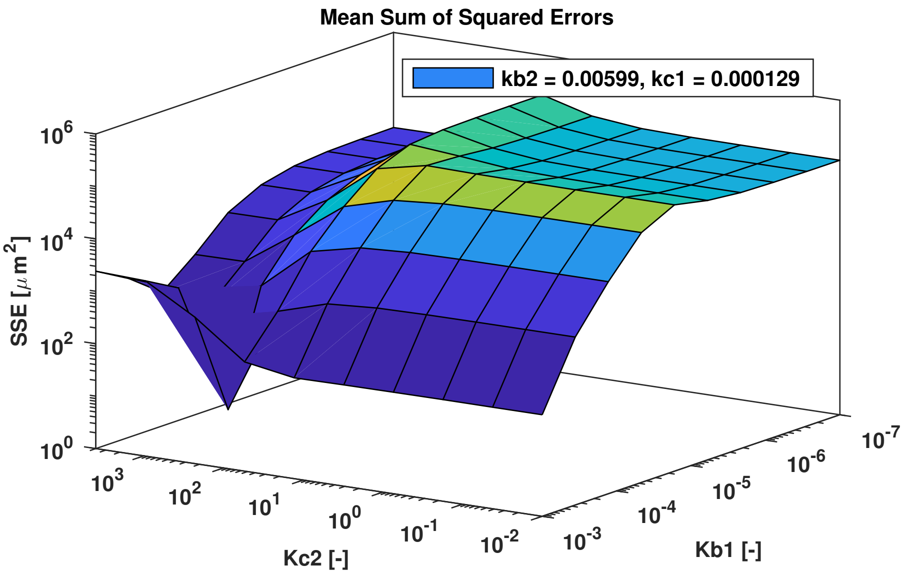


Figure A.12: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE of the mean.

A.2.4 Varying $k_{b,2}$ and $k_{c,1}$

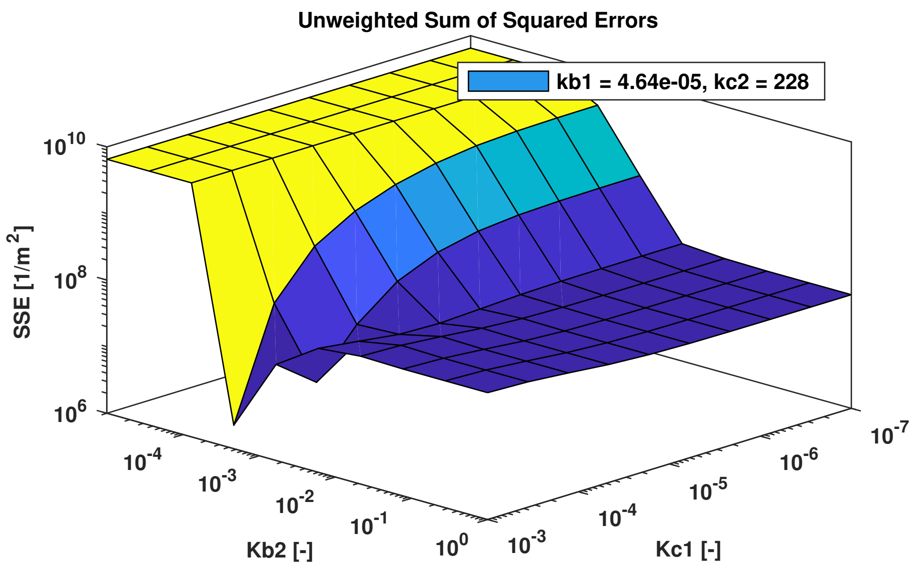


Figure A.13: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE.

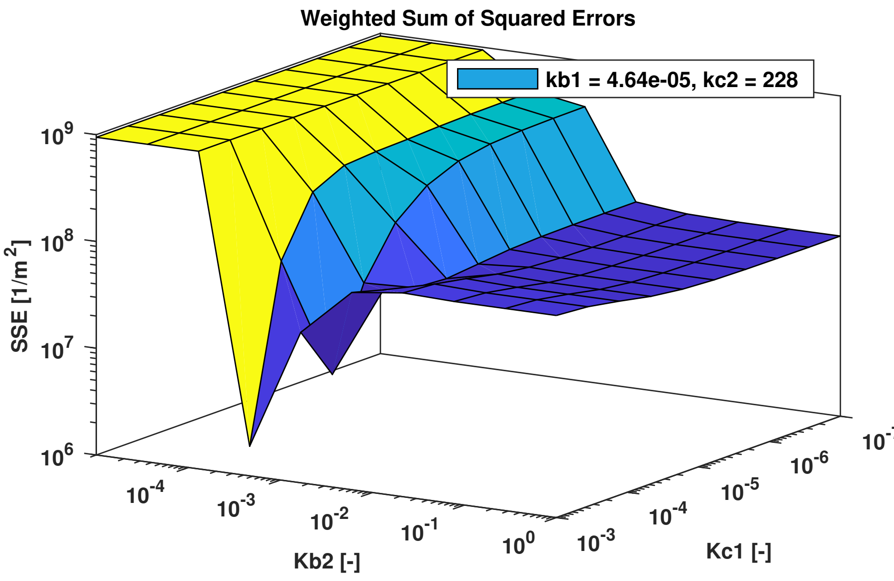


Figure A.14: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the weighted SSE.

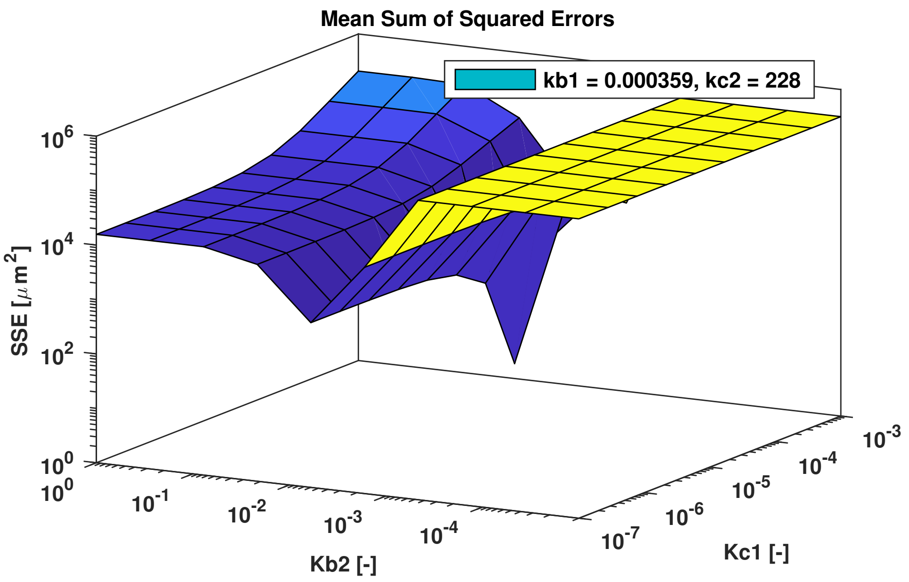


Figure A.15: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE of the mean.

A.2.5 Varying $k_{b,2}$ and $k_{c,2}$

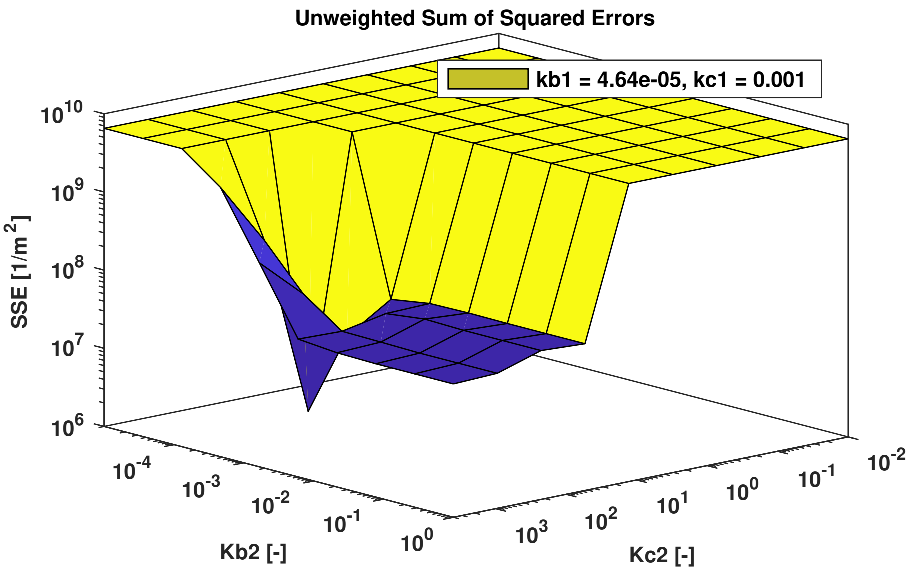


Figure A.16: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE.

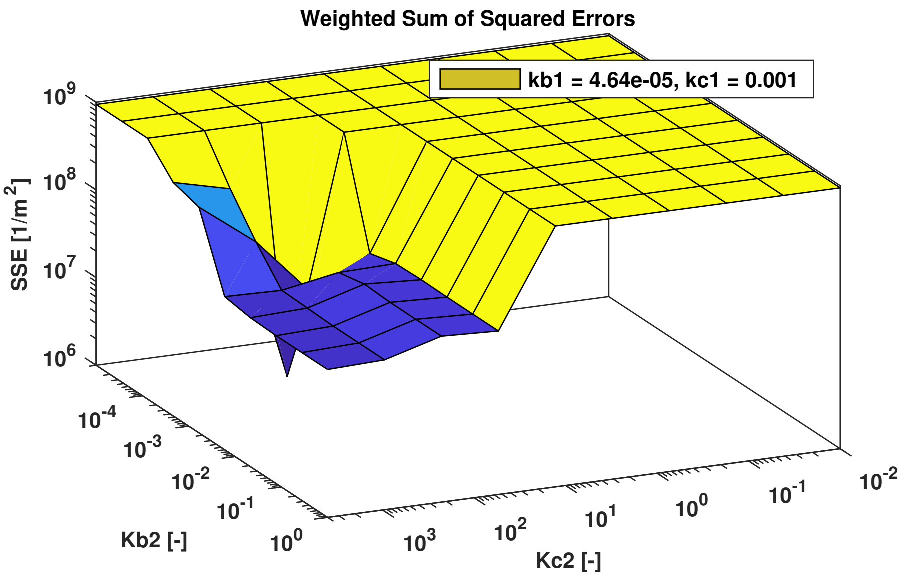


Figure A.17: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the weighted SSE.

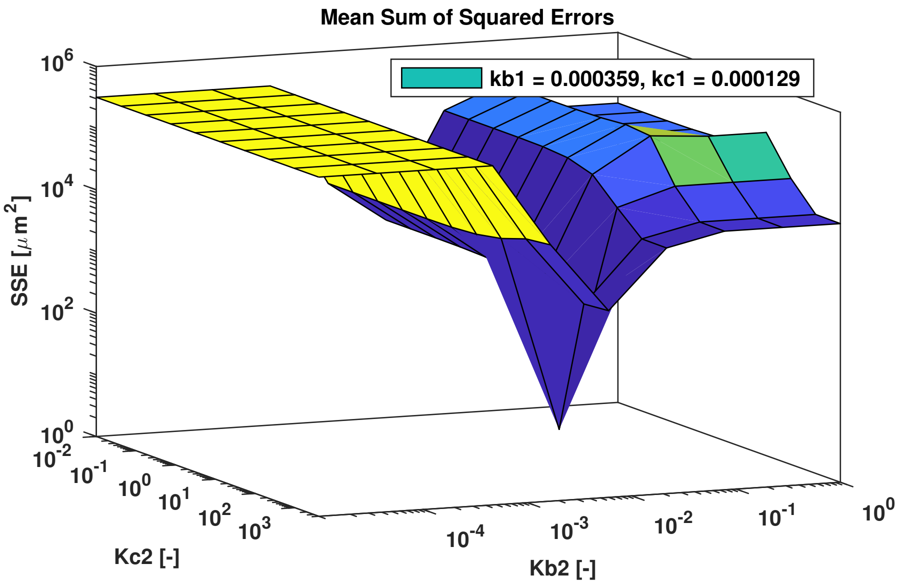


Figure A.18: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE of the mean.

A.2.6 Varying $k_{c,1}$ and $k_{c,2}$

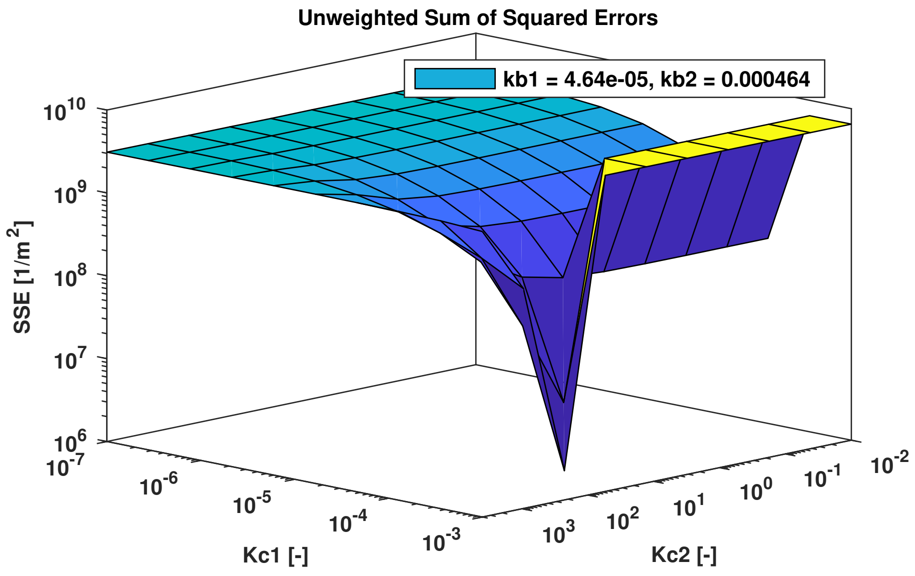


Figure A.19: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE.

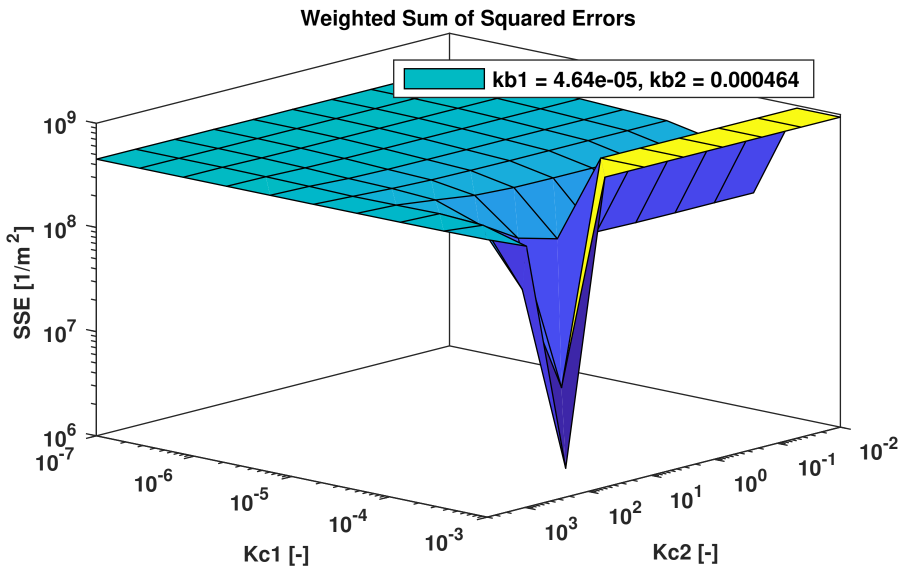


Figure A.20: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the weighted SSE.

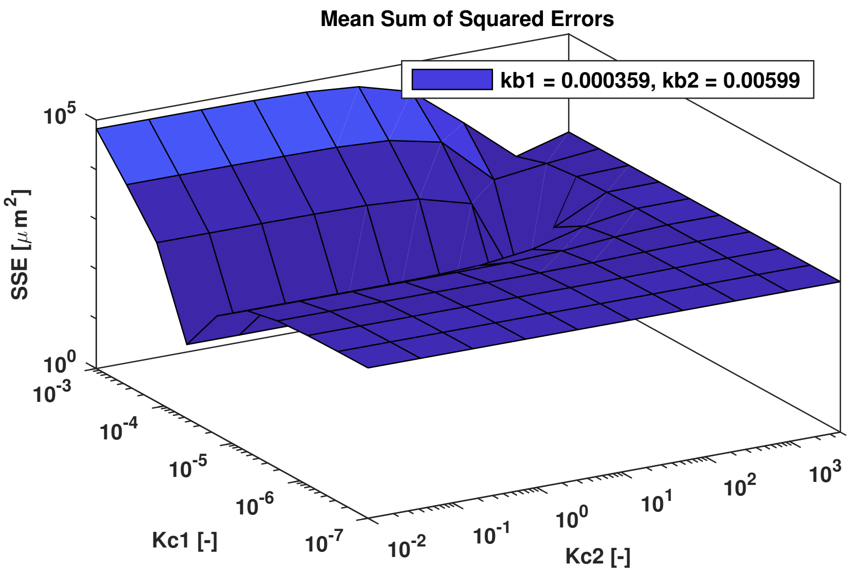


Figure A.21: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE of the mean.

A.3 Augmented Parameter Search, Refined

The coarse parameter search was refined, and closer attention was put on areas of interest, increasing the resolution to 15 grid points in each direction. The different viewpoints are presented in the subsections below.

A.3.1 Varying $k_{b,1}$ and $k_{b,2}$

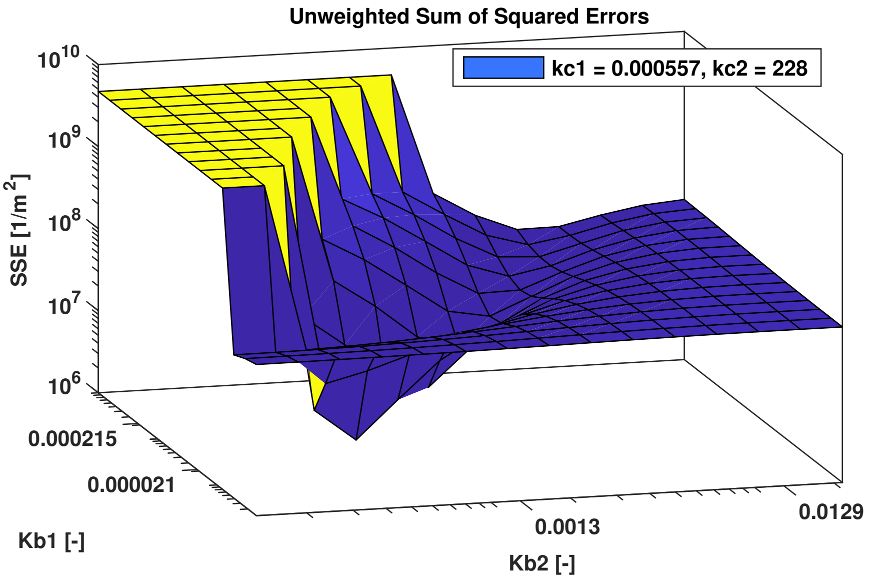


Figure A.22: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE.

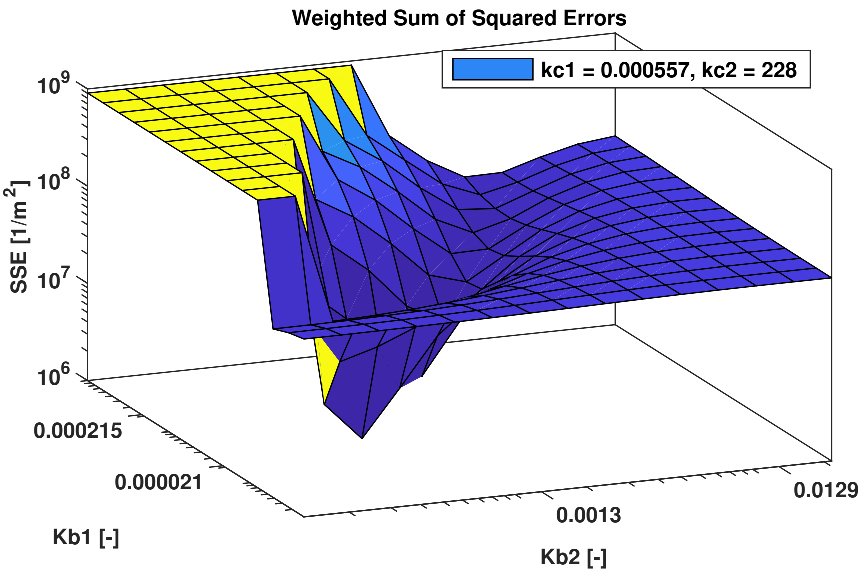


Figure A.23: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the weighted SSE.

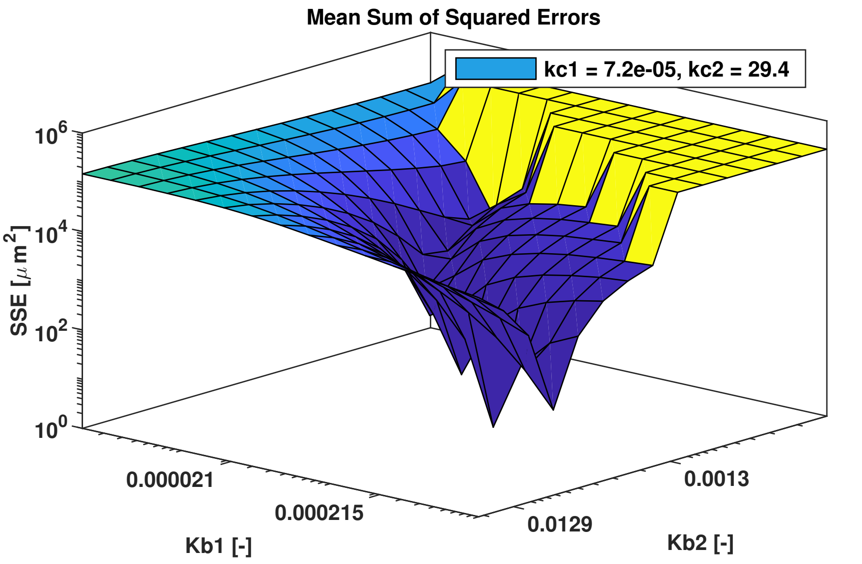


Figure A.24: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE of the mean.

A.3.2 Varying $k_{b,1}$ and $k_{c,1}$

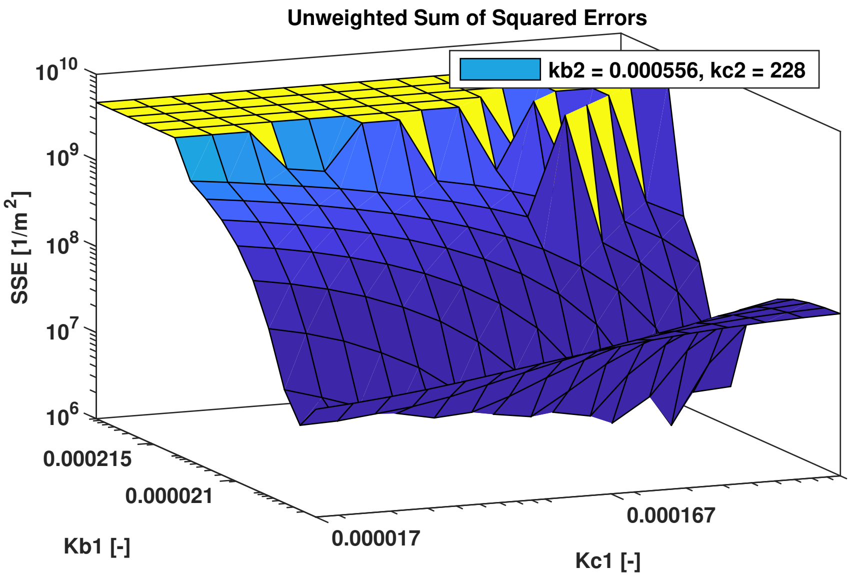


Figure A.25: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE.

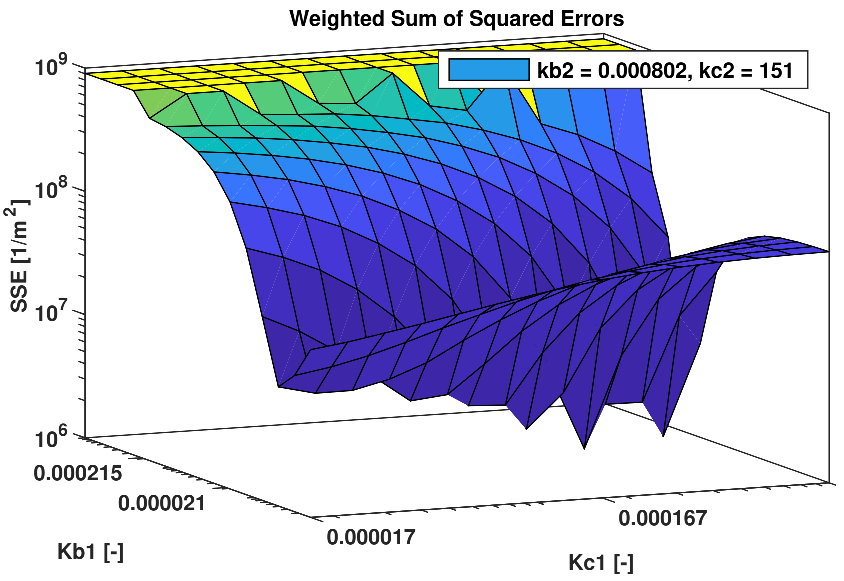


Figure A.26: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the weighted SSE.

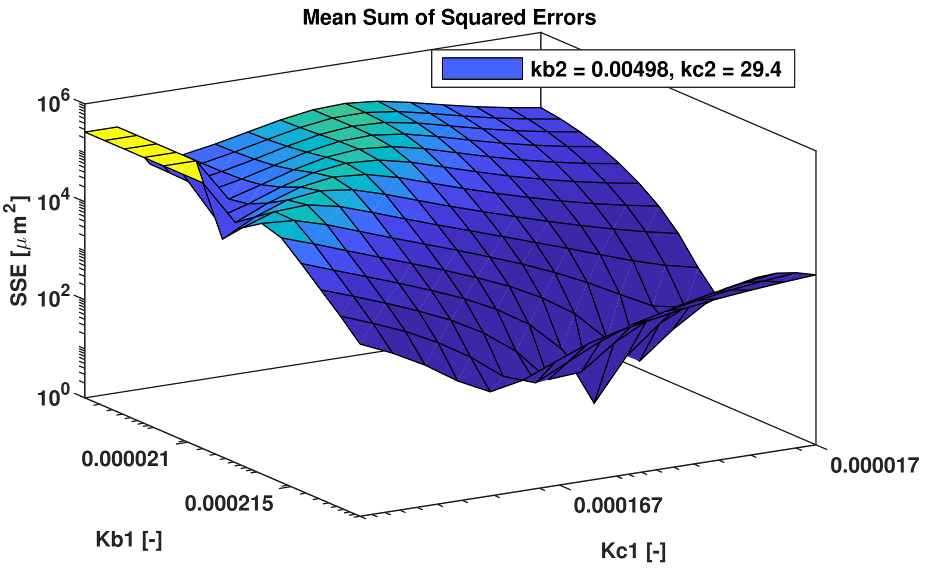


Figure A.27: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE of the mean.

A.3.3 Varying $k_{b,1}$ and $k_{c,2}$

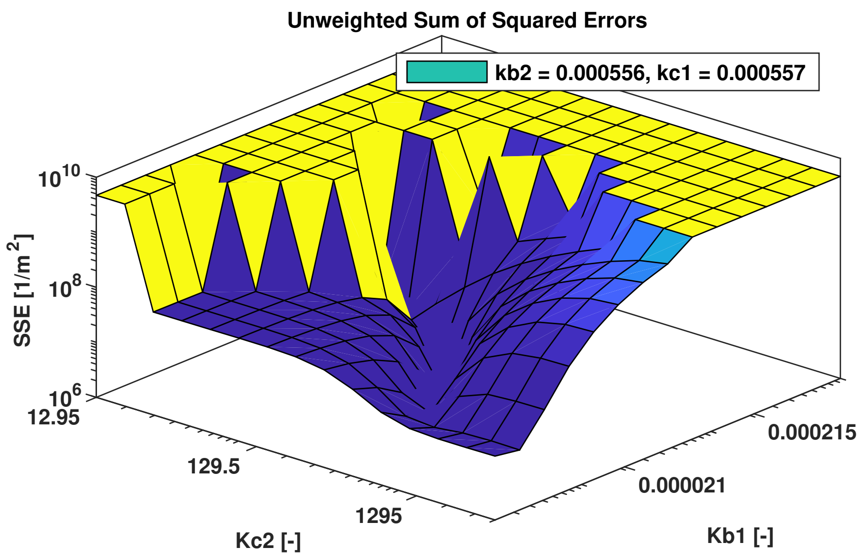


Figure A.28: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE.

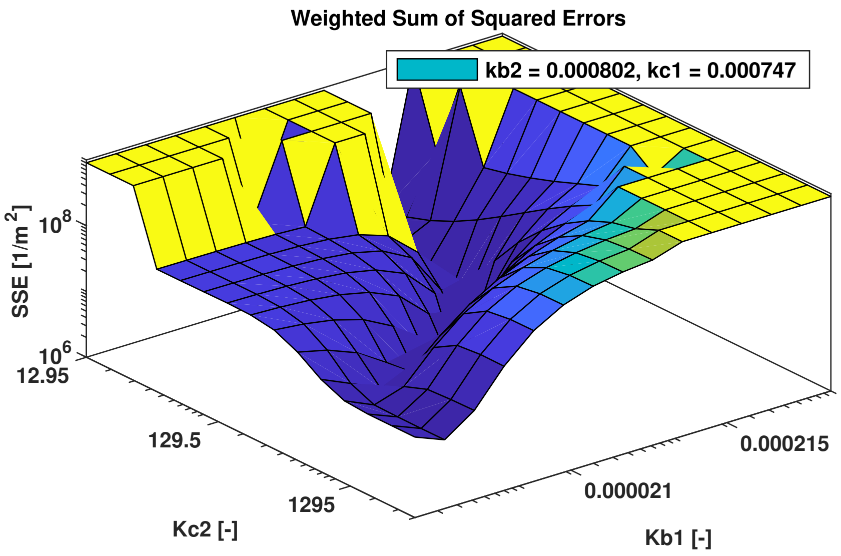


Figure A.29: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the weighted SSE.

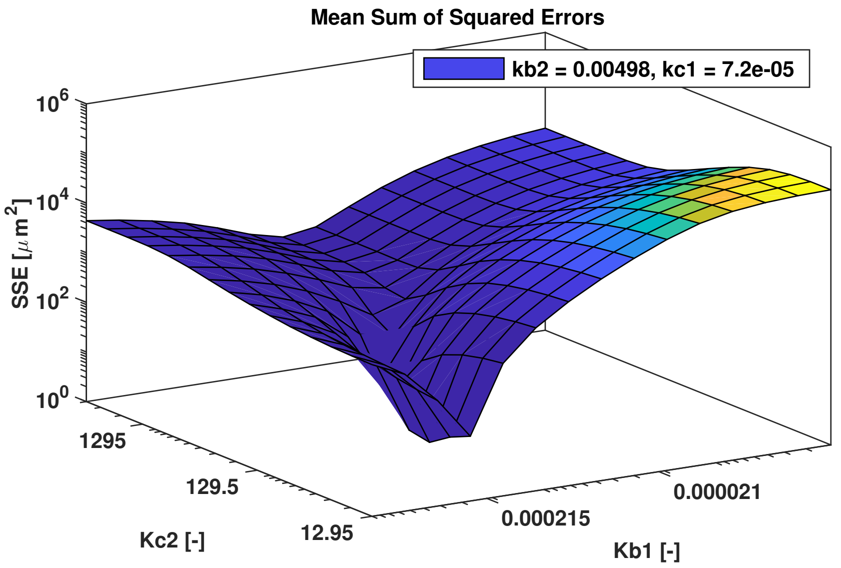


Figure A.30: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE of the mean.

A.3.4 Varying $k_{b,2}$ and $k_{c,1}$

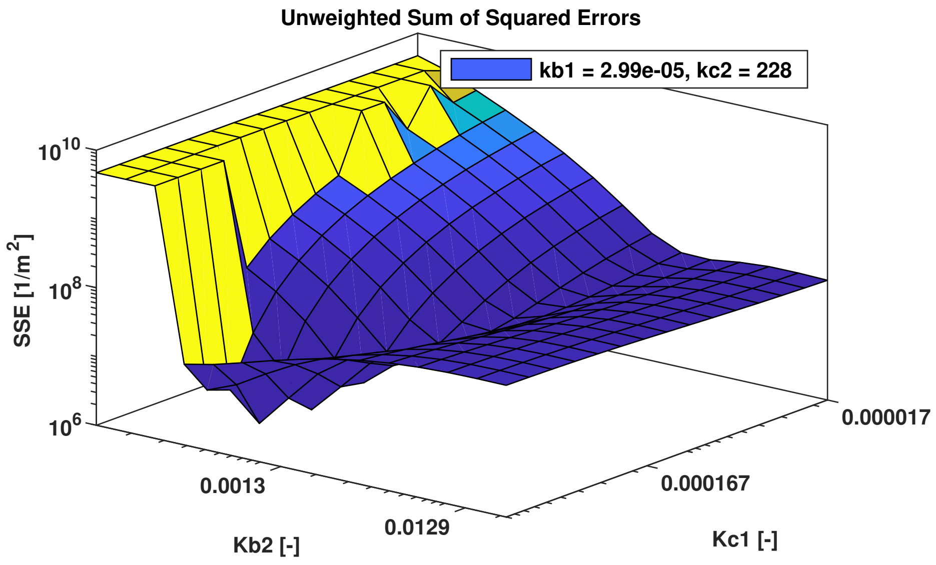


Figure A.31: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE.

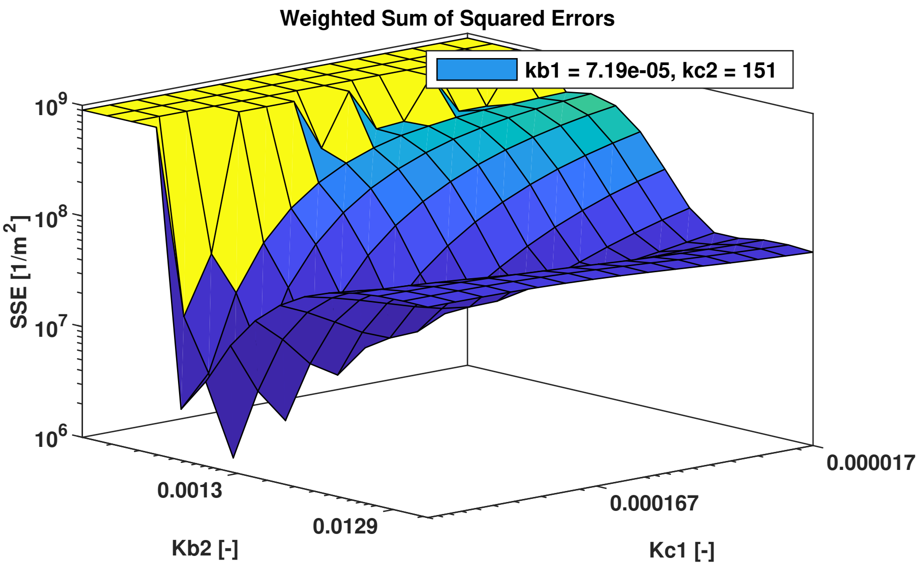


Figure A.32: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the weighted SSE.

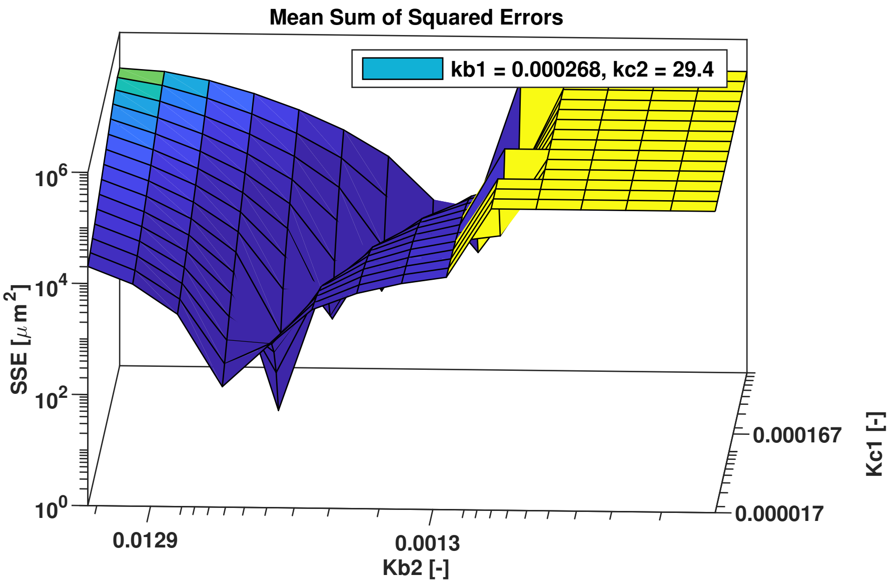


Figure A.33: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE of the mean.

A.3.5 Varying $k_{b,2}$ and $k_{c,2}$

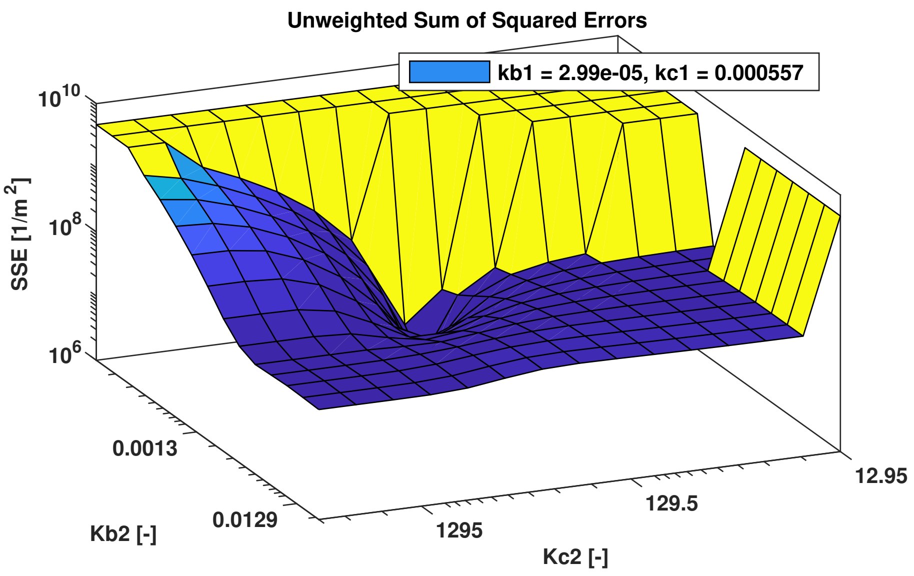


Figure A.34: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE.

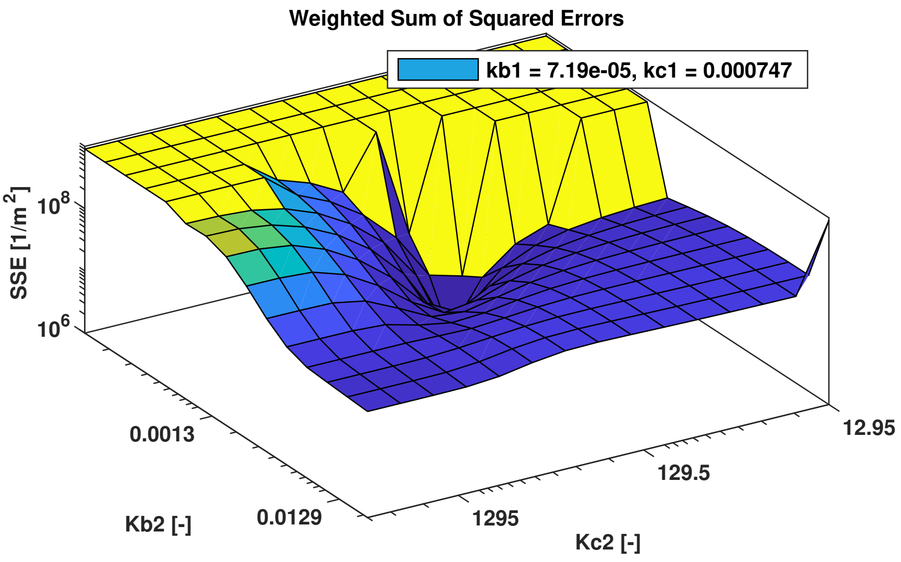


Figure A.35: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the weighted SSE.

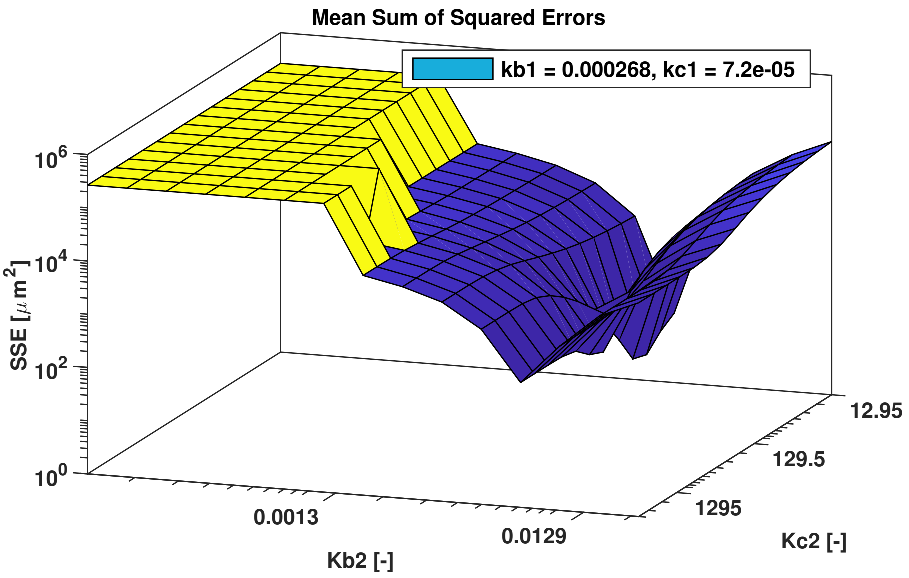


Figure A.36: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE of the mean.

A.3.6 Varying $k_{c,1}$ and $k_{c,2}$

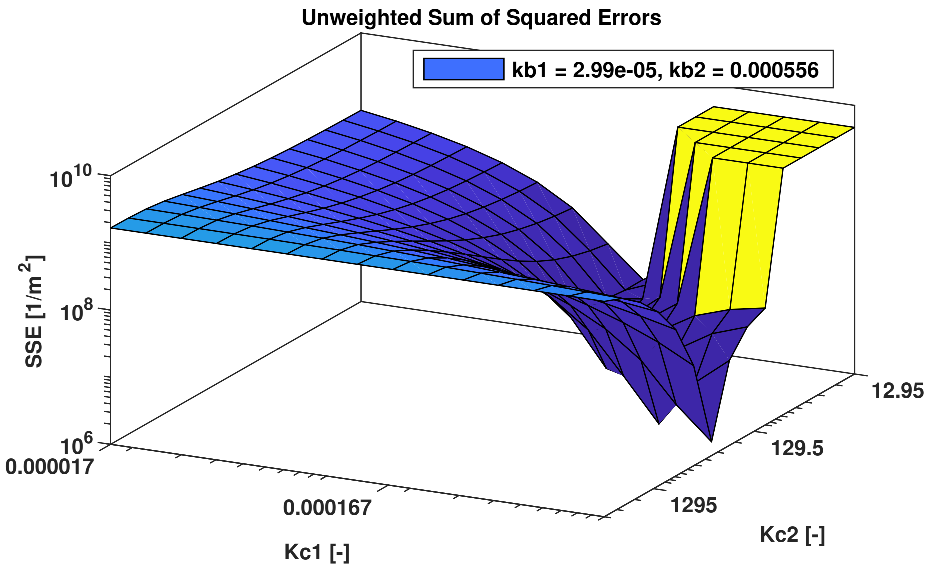


Figure A.37: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE.

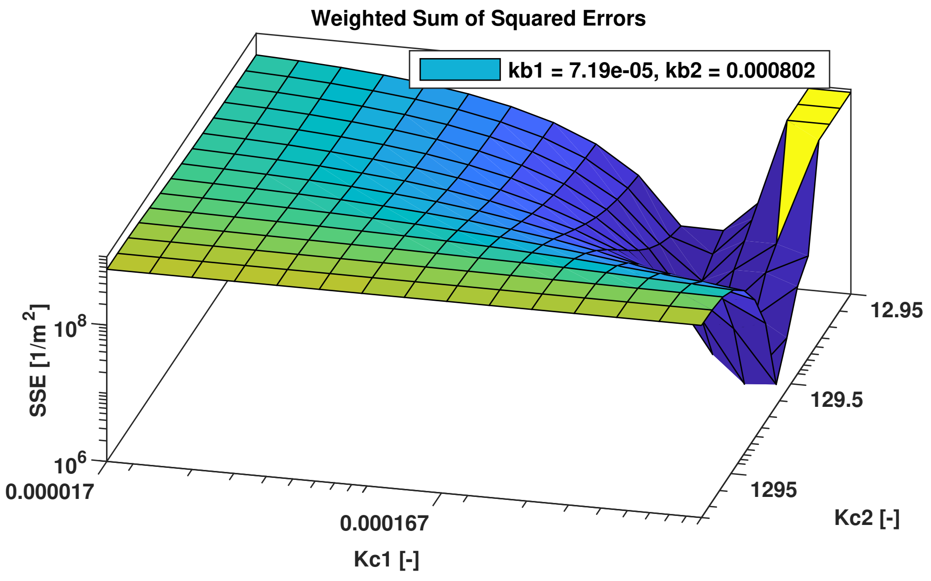


Figure A.38: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the weighted SSE.

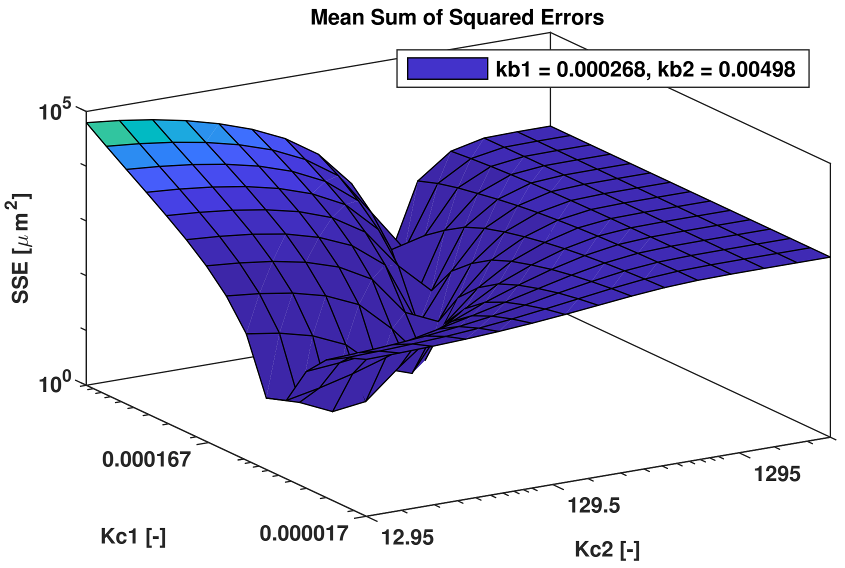


Figure A.39: The parameter space is produced by holding the other two parameters constant at their optimum. The figure is for the SSE of the mean.

Appendix B

C++ Program

The C++ program had two dependencies: GSL and SUNDIALS. To make the program run, the compiler was informed of the dependencies by use of a CMake program that is printed below.

```
cmake_minimum_required(VERSION 3.9)
project(MasterProjectCpp)

set(CMAKE_CXX_STANDARD 11)

set(programs main testSolution testLogNormalInitialCondition checkSensitivity chartMeanSSE
        bruteForceParamEstimation)
set(programs_dependencies
        cmake-build-debug/crudeB.csv
        results/raw_logNormal3.txt
        Fluid.cpp Fluid.h
        SystemProperties.cpp SystemProperties.h
        Grid.cpp Grid.h
        Kernels.cpp Kernels.h
        PBModel.cpp PBModel.h
        )

############################ SUNDIALS ############################
# Specify path to SUNDIALS header files
SET(SUNDIALS_INC_DIR
        /Users/Sindre/Sundials/instdir/include
        CACHE STRING
        "Location of SUNDIALS header files")

# Add path to SUNDIALS header files
```

```
INCLUDE_DIRECTORIES(${SUNDIALS_INC_DIR})

# Set search path for SUNDIALS libraries
SET(SUNDIALS_LIB_DIR /Users/Sindre/Sundials/instdir/lib)

# Find the SUNDIALS solver's library
FIND_LIBRARY(SUNDIALS_SOLVER_LIB
        sundials_cvode ${SUNDIALS_LIB_DIR}
        DOC "CVODE library")

# Find the NVECTOR library
FIND_LIBRARY(SUNDIALS_NVEC_LIB
        sundials_nvecserial ${SUNDIALS_LIB_DIR}
        DOC "NVECTOR library")

# Set an extra link directory if necessary (false if empty)
IF(EXISTS )
    LINK_DIRECTORIES()
ENDIF()

# Set additional libraries
SET(SUNDIALS_EXTRA_LIB  -lm CACHE STRING "Additional libraries")

# List of Sundials libraries shared across all examples
SET(SUNDIALS_LIBS ${SUNDIALS_SOLVER_LIB} ${SUNDIALS_NVEC_LIB}
↪   ${SUNDIALS_EXTRA_LIB})
########################################################################⌋
↪   ############


############################ GNU Scientific Library
↪   ############################
# Specify path to GSL header files
SET(GSL_INC_DIR
        /Users/sindre/GSL/instdir/include
        CACHE STRING
        "Location of GSL header files")
# Add path to GSL header files
INCLUDE_DIRECTORIES(${GSL_INC_DIR})
# Set search path for GSL libraries
SET(GSL_LIB_DIR /Users/sindre/GSL/instdir/lib)
# Find library
FIND_LIBRARY(GSL_LIBRARY
        gsl ${GSL_LIB_DIR}
        DOC "GSL library")
FIND_LIBRARY(GSL_CBLAS
        gslcblas ${GSL_LIB_DIR}
        DOC "GSL BLAS Library")
```

```
# List of GSL libraries
SET(GSL_LIBS ${GSL_LIBRARY} ${GSL_CBLAS})
##########################################################################
↪  #########
FOREACH(program ${programs})

        # example source files
        ADD_EXECUTABLE(${program} ${program}.cpp ${programs_dependencies})

        # libraries to link against
        TARGET_LINK_LIBRARIES(${program} ${SUNDIALS_LIBS} ${GSL_LIBS})

ENDFOREACH(program ${programs})
```

Listing B.1: CMakeLists.txt: Instructions for compiler

B.1 Source Files

```
/* Built-in header files */
#include <iostream>
#include <vector>

/* User-defined header files */
#include "PBModel.h"

int main(int argc, char **argv) {
    std::ifstream fin(argv[1]);
    if (!fin) {
        std::cerr << "\nError: failure opening " << argv[1] << std::endl;
        exit(-1);
    }
    /*********************************************************************
    ↪  *********************/
    /* Description of program
    ↪                         */
    /*********************************************************************
    ↪  *********************/
    /* This is the main program that solves the Population Balance
    ↪  Equation (PBE)
     * We have some classes to help us solve the model:
     *  - Grid              :: Contains all variables needed for Gaussian
↪   quadrature rule
     *  - Fluid             :: Contains density, surface tension and
↪   viscosity for a fluid
```

```
     *  - SystemProperties  :: Contains variables such as maximum radius,
↪   volume of tanks etc
     *  - Constants         :: Contains parameters such as
↪   k1,k2,k3,k4,kb1,kb2,kc1,kc2
     *  - Kernels           :: Contains kernels for breakage (KBB,kDB)
↪   and coalescence (KBC,KDC)
     *  - PBModel           :: Solves the entire model by the use of an
↪   ODE solver:
     *                         Utilizes all above classes
     */

    /********************************************************************⌋
    ↪   ********************/
    /* Input handling from Dakota
    ↪                     */
    /********************************************************************⌋
    ↪   ********************/
    /* We are processing a input file of format
     *                                     4 variables
     *                  0.000000000000000e+00 kb1
     *                  0.000000000000000e+00 kb2
     *                  0.000000000000000e+00 kc1
     *                  0.000000000000000e+00 kc2
     *                                      80 functions
     *                                      1 ASV_1:least_sq_term_1
     *                                      1 ASV_2:least_sq_term_2
     *                                      1 ASV_3:least_sq_term_3
     *                                      ...
     *                                      1 ASV_80:least_sq_term_80
     *                                      4 derivative_variables
     *                                      1 DVV_1:kb1
     *                                      2 DVV_2:kb2
     *                                      3 DVV_3:kc1
     *                                      4 DVV_4:kc2
     *                                      0 analysis_components
     *                                      1 eval_id
     */
    size_t i, j, k, num_vars, num_fns, num_deriv_vars;  /* num means
    ↪   number of                  */
    std::string vars_text, fns_text, dvv_text;          /* Description
    ↪   text (2nd column above)   */

    // Get the parameter std::vector and ignore the labels
    fin >> num_vars >> vars_text;
    std::vector<double> x(num_vars);
    for (i=0; i<num_vars; i++) {
        fin >> x[i];
        fin.ignore(256, '\n');
    }

    // Get the ASV std::vector and ignore the labels
    /* Possible ASV values:
     * 1 (function value)
     * 2 (gradient)
     * 3 (function value and gradient)
     * 4 (hessian)
     * 5 (function value and hessian)
     * 6 (gradient and hessian)
     * 7 (function value, gradient and hessian)
     */
    fin >> num_fns >> fns_text;
    std::vector<int> ASV(num_fns);
    for (i=0; i<num_fns; i++) {
        fin >> ASV[i];
        fin.ignore(256, '\n');
    }

    // Get the DVV std::vector and ignore the labels
    fin >> num_deriv_vars >> dvv_text;
    std::vector<int> DVV(num_deriv_vars);
    for (i=0; i<num_deriv_vars; i++) {
        fin >> DVV[i];
        fin.ignore(256, '\n');
    }

    /*********************************************************************↲
     ↪   ********************/
    /* Declaration of variables
     ↪                      */
    /*********************************************************************↲
     ↪   ********************/
    const size_t Np = 200;                      /* Number of grid points
     ↪     */
    char const *fileName      = "crudeB.csv";   /* Experimental data
     ↪     */
    const std::string outName = "output1.dat";  /* Output filename
     ↪     */
    realtype kb1 = x[0],                   /* Model fitted parameters  */
             kb2 = x[1],
             kc1 = x[2],
             kc2 = x[3];
    /* x[2] is actually the ratio kc1/kb1. kc1 = kb1*x[2] */

    /*********************************************************************↲
     ↪   ********************/
```

```cpp
    /* Instantiation and solution                              */
    /*****************************************************************************************/
    /* Helper classes */
    Grid g = Grid(Np, 0, 1, 0, 0, 2);
    Fluid disp = Fluid(0.837e3, 22.0e-3, 16.88e-3); /* Oil       */
    Fluid cont = Fluid(1.0e3, 1, 1);                /* Water     */
    SystemProperties s = SystemProperties(500.0e-6, 725.0e-6, 0.366, disp);
    /* Solution class */
    PBModel m = PBModel(fileName, kb1, kb2, kc1, kc2, g, s, cont, disp, 0);
    m.solvePBE();

    /*****************************************************************************************/
    /* Output handling to Dakota                               */
    /*****************************************************************************************/
    std::ofstream fout(argv[2]);
    if (!fout) {
        std::cerr << "\nError: failure creating " << argv[2] << std::endl;
        exit(-1);
    }
    fout.precision(15); // 16 total digits
    fout.setf(std::ios::scientific);
    fout.setf(std::ios::right);

    /** Evaluate residuals for t = tf (end of time horizon) **/
    size_t M = m.getM(), N = m.getN();
    double currentRes = 0;
    if (!m.checkMassBalance()){
        for (i = 0; i < M; i++) {
            for (j = 0; j < N; j++) {
                if (ASV[i] & 1) {
                    fout << "                     " << 1.e30 << " f" << i * N + j + 1 << std::endl;
                }
            }
        }
    } else {
        for (i = 0; i < M; i++) {
            for (j = 0; j < N; j++) {
                if (ASV[i] & 1) {
                    currentRes = m.getResidualij((size_t) round(i * M / M), j);
                    fout << "                      " << currentRes << " f"
                     ↪  << i * N + j + 1 << std::endl;
                }
            }
        }
    }
    fout.flush();
    fout.close();
    return 0;
}
```

Listing B.2: main.cpp: Program that was used to interface with Dakota

```
//
// Created by Sindre Bakke Øyen on 10.05.2018.
//

#include <iostream>
#include <chrono>
#include <ctime>
#include <vector>

/* User-defined header files */
#include "PBModel.h"

int main() {
    /********************************************************************
     ↪   ********************/
    /* Description of program
     ↪                    */
    /********************************************************************
     ↪   ********************/
    /* This is the main program that solves the Population Balance
     ↪   Equation (PBE)
     * We have some classes to help us solve the model:
     *  - Grid              :: Contains all variables needed for Gaussian
↪   quadrature rule
     *  - Fluid             :: Contains density, surface tension and
↪   viscosity for a fluid
     *  - SystemProperties  :: Contains variables such as maximum radius,
↪   volume of tanks etc
     *  - Constants         :: Contains parameters such as
↪   k1,k2,k3,k4,kb1,kb2,kc1,kc2
     *  - Kernels           :: Contains kernels for breakage (KBB,kDB)
↪   and coalescence (KBC,KDC)
     *  - PBModel           :: Solves the entire model by the use of an
↪   ODE solver:
     *                         Utilizes all above classes
     */

    /*********************************************************************
    ↪   ********************/
    /* Declaration of variables
    ↪                         */
    /*********************************************************************
    ↪   ********************/
    const size_t Np = 220;                    /* Number of grid points
    ↪     */
    char const *fileName = "crudeB.csv";      /* Experimental data
    ↪     */
    realtype kb1 = 7.e-6,   /* From testLogNormalInitialCondition.cpp */
            kb2 = 2.e-4,    /* From testLogNormalInitialCondition.cpp */
            kc1 = 1.e-4,    /* From testLogNormalInitialCondition.cpp */
            kc2 = 3.e2;     /* From testLogNormalInitialCondition.cpp */
    kb1 = 8.e-6;
    kb2 = kb2;
    kc1 = 5.e-4;
    kc2 = 4.e2;

    size_t Npts = 10;
    std::vector<gsl_matrix *> SSEvector(Npts*Npts, nullptr);    /* 10x10
    ↪   matrix of matrices */
    std::vector<gsl_matrix *> wSSEvector(Npts*Npts, nullptr);   /* 10x10
    ↪   matrix of matrices */
    std::vector<gsl_matrix *> meanSSEvector(Npts*Npts, nullptr);   /*
    ↪   10x10 matrix of matrices */
    std::vector<PBModel *> m(Npts*Npts*Npts*Npts, nullptr);
    std::vector<realtype> kb1vec(Npts, 0);
    std::vector<realtype> kb2vec(Npts, 0);
    std::vector<realtype> kc1vec(Npts, 0);
    std::vector<realtype> kc2vec(Npts, 0);

    /* Set search space */
    realtype kb1lb = log10(1.e-7), kb1ub = log10(1.e-3);
    realtype kb2lb = log10(1.e-4), kb2ub = log10(1.e0);
    realtype kc1lb = log10(1.e-7), kc1ub = log10(1.e-3);
    realtype kc2lb = log10(1.e-2), kc2ub = log10(4.e3);
//    realtype kb1lb = log10(2.15e-6), kb1ub = log10(1.e-3);
//    realtype kb2lb = log10(1.29e-4), kb2ub = log10(2.15e-2);
//    realtype kc1lb = log10(1.67e-5), kc1ub = log10(1.e-3);
//    realtype kc2lb = log10(12.95), kc2ub = log10(4.e3);

    size_t i, j, q, r, w;
    for (i = 0; i < Npts; i++){
        kb1vec[i] = pow(10.0, (realtype) i / (Npts-1) * (kb1ub-kb1lb) +
         ↪  kb1lb);
        kb2vec[i] = pow(10.0, (realtype) i / (Npts-1) * (kb2ub-kb2lb) +
         ↪  kb2lb);
        kc1vec[i] = pow(10.0, (realtype) i / (Npts-1) * (kc1ub-kc1lb) +
         ↪  kc1lb);
        kc2vec[i] = pow(10.0, (realtype) i / (Npts-1) * (kc2ub-kc2lb) +
         ↪  kc2lb);
    }

    /* Choose which distributions to include in objective function */
    std::vector<size_t> dist_idx(27, 0);
    j = 2;
    for (i = 0; i < 26; i++){
        if (i < 20) { dist_idx[i] = i; }
        else {
            j++;
            dist_idx[i] = j*10;
        }
    }
    dist_idx[dist_idx.size()-1] = 88;
    /********************************************************************
     ↪   ********************/
    /* Instantiation and solution
     ↪                         */
    /********************************************************************
     ↪   ********************/
    /* Helper classes */
    Grid g = Grid(Np, 0, 1, 0, 0, 2);
    Fluid disp = Fluid(0.837e3, 22.0e-3, 16.88e-3); /* Oil       */
    Fluid cont = Fluid(1.0e3, 1, 1);                /* Water     */
    SystemProperties s = SystemProperties(500.0e-6, 725.0e-6, 0.366,
     ↪  disp);

    /* Looping on all parameters */
    size_t index_ijqr = 0, index_ij = 0, N = 0, M = 0;
    std::vector<size_t>::iterator it;
    realtype res = 0, wres = 0, meanres = 0, currentRes = 0, mid = 6.e-6,
     ↪  slack = 1.e-6;
    for (i = 0; i < Npts; i++){
        kb1 = kb1vec[i];
        for (j = 0; j < Npts; j++){
            kb2 = kb2vec[j];
            index_ij = Npts*i + j;
            SSEvector[index_ij] = gsl_matrix_calloc(Npts, Npts);
            wSSEvector[index_ij] = gsl_matrix_calloc(Npts, Npts);
            meanSSEvector[index_ij] = gsl_matrix_calloc(Npts, Npts);
            for (q = 0; q < Npts; q++) {
```

```cpp
                kc1 = kc1vec[q];
                for (r = 0; r < Npts; r++) {
                    kc2 = kc2vec[r];
                    index_ijqr = (size_t) (i * pow(Npts, 3) + j *
                     ↪  pow(Npts, 2) + q * pow(Npts, 1) + r * pow(Npts,
                     ↪  0));
                    std::cout << index_ijqr << std::endl;
                    m[index_ijqr] = new PBModel(fileName, kb1, kb2, kc1,
                     ↪  kc2, g, s, cont, disp, 0);
                    m[index_ijqr]->solvePBE();
                    M = dist_idx.size();
                    N = m[index_ijqr]->getN();
                    res = 0;
                    wres = 0;
                    meanres = 0;
                    for (it = dist_idx.begin(); it != dist_idx.end();
                     ↪  it++){
                        for (w = 0; w < N; w++){
                            realtype dropSize =
                             ↪  gsl_vector_get(m[index_ijqr]->getR(), w);
                            currentRes =
                             ↪  pow(m[index_ijqr]->getResidualij(*it, w),
                             ↪  2);
                            res += currentRes;
                            wres += currentRes *
                             ↪  m[index_ijqr]->getWeight(dropSize, mid,
                             ↪  slack);
                        }
                        meanres +=
                         ↪  pow(m[index_ijqr]->getResidualMean(*it), 2);
                    }
                    if (m[index_ijqr]->checkMassBalance()){
                        gsl_matrix_set(SSEvector[index_ij], q, r, res);
                        gsl_matrix_set(wSSEvector[index_ij], q, r, wres);
                        gsl_matrix_set(meanSSEvector[index_ij], q, r,
                         ↪  meanres);
                    }
                    else {
                        gsl_matrix_set(SSEvector[index_ij], q, r, NAN);
                        gsl_matrix_set(wSSEvector[index_ij], q, r, NAN);
                        gsl_matrix_set(meanSSEvector[index_ij], q, r,
                         ↪  NAN);
                    }
                    delete m[index_ijqr];
                }
            }
        }
    }
    /* Write to file */
    time_t rawtime;
    struct tm * timeinfo;
    char buffer[80];
    time (&rawtime);
    timeinfo = localtime(&rawtime);
    strftime(buffer,sizeof(buffer),"%d-%m-%Y-%I:%M:%S",timeinfo);
    std::string str(buffer);

    std::ofstream fbparams("../results/parameterFiles/paramsBreakage" +
     ↪  str + ".dat");
    std::ofstream fcparams("../results/parameterFiles/paramsCoalescence"
     ↪  + str + ".dat");
    std::ofstream fresiduals("../results/residualFiles/SSEallTimes" + str
     ↪  + ".dat");
    std::ofstream fwresiduals("../results/residualFiles/wSSEallTimes" +
     ↪  str + ".dat");
    std::ofstream
     ↪  fmeanresiduals("../results/residualFiles/meanSSEallTimes" + str +
     ↪  ".dat");
    fbparams << "#kb1,#kb2\n";
    for (i = 0; i < Npts; i++) {
        fbparams << kb1vec[i] << "," << kb2vec[i] << std::endl;
    }
    fcparams << "#kc1,#kc2\n";
    for (i = 0; i < Npts; i++) {
        fcparams << kc1vec[i] << "," << kc2vec[i] << std::endl;
    }
    fresiduals << "#SSE\n";
    for (i = 0; i < Npts; i++){
        for (j = 0; j < Npts; j++) {
            index_ij = i * Npts + j;
            gsl_matrix *currentMat = SSEvector[index_ij];
            for (q = 0; q < Npts; q++) {
                for (r = 0; r < Npts; r++) {
                    fresiduals << gsl_matrix_get(currentMat, q, r) << ",";
                }
                fresiduals << std::endl;
            }
            gsl_matrix_free(currentMat);
        }
    }
    fwresiduals << "#wSSE\n";
    for (i = 0; i < Npts; i++){
        for (j = 0; j < Npts; j++) {
            index_ij = i * Npts + j;
            gsl_matrix *currentMat = wSSEvector[index_ij];
            for (q = 0; q < Npts; q++) {
                for (r = 0; r < Npts; r++) {
                    fwresiduals << gsl_matrix_get(currentMat, q, r) <<
                     ↪  ",";
                }
                fwresiduals << std::endl;
            }
            gsl_matrix_free(currentMat);
        }
    }
    fmeanresiduals << "#meanSSE\n";
    for (i = 0; i < Npts; i++){
        for (j = 0; j < Npts; j++) {
            index_ij = i * Npts + j;
            gsl_matrix *currentMat = meanSSEvector[index_ij];
            for (q = 0; q < Npts; q++) {
                for (r = 0; r < Npts; r++) {
                    fmeanresiduals << gsl_matrix_get(currentMat, q, r) <<
                     ↪  ",";
                }
                fmeanresiduals << std::endl;
            }
            gsl_matrix_free(currentMat);
        }
    }

    fbparams.close();
    fcparams.close();
    fresiduals.close();
    fwresiduals.close();
    fmeanresiduals.close();
    return 0;
}
```

Listing B.3: bruteForceParamEstimation.cpp: Program that was used to chart the parameter space

```
//
// Created by Sindre Bakke Øyen on 30.04.2018.
//

#include <iostream>
#include <chrono>
#include <ctime>
#include <vector>

/* User-defined header files */
#include "PBModel.h"
```

```

int main() {
    /*****************************************************************************⌋
     ↪   ********************/
    /* Description of program
     ↪                        */
    /*****************************************************************************⌋
     ↪   ********************/
    /* This is the main program that solves the Population Balance
     ↪   Equation (PBE)
     * We have some classes to help us solve the model:
     *  - Grid                :: Contains all variables needed for Gaussian
 ↪   quadrature rule
     *  - Fluid               :: Contains density, surface tension and
 ↪   viscosity for a fluid
     *  - SystemProperties    :: Contains variables such as maximum radius,
 ↪   volume of tanks etc
     *  - Constants           :: Contains parameters such as
 ↪   k1,k2,k3,k4,kb1,kb2,kc1,kc2
     *  - Kernels             :: Contains kernels for breakage (KBB,kDB)
 ↪   and coalescence (KBC,KDC)
     *  - PBModel             :: Solves the entire model by the use of an
 ↪   ODE solver:
     *                           Utilizes all above classes
     */

    /*****************************************************************************⌋
     ↪   ********************/
    /* Declaration of variables
     ↪                        */
    /*****************************************************************************⌋
     ↪   ********************/
    const size_t Np = 200;                        /* Number of grid points
     ↪    */
    char const *fileName = "crudeB.csv";         /* Experimental data
     ↪    */
    realtype kb1 = 7.e-6,    /* From testLogNormalInitialCondition.cpp */
             kb2 = 2.e-4,    /* From testLogNormalInitialCondition.cpp */
             kc1 = 1.e-4,    /* From testLogNormalInitialCondition.cpp */
             kc2 = 3.e2;     /* From testLogNormalInitialCondition.cpp */
    kb1 = 8.e-6;
    kb2 = kb2;
    kc1 = 5.e-4;
    kc2 = 4.e2;

    size_t Npts = 50;
    std::vector<gsl_matrix *> residualVector(Npts*Npts, nullptr); /*
     ↪   19x19 matrix of matrices */
    std::vector<PBModel *> m(Npts*Npts, nullptr);
    std::vector<realtype> k1vec(Npts, 0);
    std::vector<realtype> k2vec(Npts, 0);
    realtype k1lb = log10(5.e-5), k1ub = log10(5.e-3);
    realtype k2lb = log10(40), k2ub = log10(4000);
    size_t i, j, q;
    for (i = 0; i < Npts; i++){
        k1vec[i] = pow(10.0, (realtype) i / (Npts-1) * (k1ub-k1lb) +
         ↪  k1lb);
        k2vec[i] = pow(10.0, (realtype) i / (Npts-1) * (k2ub-k2lb) +
         ↪  k2lb);
    }
    realtype k1, k2;

    /**********************************************************************⌋
     ↪   *********************/
    /* Instantiation and solution
     ↪                          */
    /**********************************************************************⌋
     ↪   *********************/
    /* Helper classes */
    Grid g = Grid(Np, 0, 1, 0, 0, 2);
    Fluid disp = Fluid(0.837e3, 22.0e-3, 16.88e-3); /* Oil      */
    Fluid cont = Fluid(1.0e3, 1, 1);                /* Water    */
    SystemProperties s = SystemProperties(500.0e-6, 725.0e-6, 0.366,
     ↪  disp);
    size_t index_ij = 0, N = 0, M = 0;
//    for (i = 0; i < Npts; i++){
//        k1 = k1vec[i];
//        for (j = 0; j < Npts; j++){
//            index_ij = i*Npts + j;
//            k2 = k2vec[j];
//            m[index_ij] = new PBModel(fileName, kb1, kb2, k1, k2, g, s,
↪   cont, disp, 0);
//            m[index_ij]->solvePBE();
//            M = m[index_ij]->getM();
//            N = m[index_ij]->getN();
//            residualVector[index_ij] = gsl_matrix_alloc(M, 1);
//            for (q = 0; q < M; q++) {
//                gsl_matrix_set(
//                        residualVector[index_ij], q, 0,
//                        m[index_ij]->getResidualMean(q)
//                );
//            }
//            delete m[index_ij];
//        }
//    }

```

```
    /* Write to file */
    time_t rawtime;
    struct tm * timeinfo;
    char buffer[80];
    time (&rawtime);
    timeinfo = localtime(&rawtime);
    strftime(buffer,sizeof(buffer),"%d-%m-%Y-%I:%M:%S",timeinfo);
    std::string str(buffer);

//    std::ofstream fparams("../results/paramsBreakage.dat");
    std::ofstream fparams("../results/parameterFiles/paramsCoalescence" +
    ↪  str + ".dat");
//    std::ofstream fresiduals("../results/residualFiles/residualsMeans"
↪  + str + ".dat");
//    std::ofstream fresiduals("../results/residualsNoDynamics.dat");
    fparams << "#k1,#k2\n";
    for (i = 0; i < Npts; i++) {
        fparams << k1vec[i] << "," << k2vec[i] << std::endl;
    }
//    fresiduals << "#residuals\n";
//    for (i = 0; i < Npts; i++){
//        for (j = 0; j < Npts; j++) {
//            index_ij = i * Npts + j;
//            gsl_matrix *currentMat = residualVector[index_ij];
//            for (q = 0; q < M; q++) {
//                fresiduals << gsl_matrix_get(currentMat, q, 0) << ",";
//            }
//            fresiduals << std::endl;
//        }
//    }
    fparams.close();
//    fresiduals.close();
    return 0;
}
```

Listing B.4: chartMeanSSE.cpp: Program that was used to chart the parameter space

```
//
// Created by Sindre Bakke Øyen on 25.04.2018.
//

#include <iostream>
#include <chrono>
#include <ctime>
#include <vector>

```

```
/* User-defined header files */
#include "PBModel.h"

int main() {
    /*********************************************************************⌋
     ↪   ********************/
    /* Description of program
     ↪                      */
    /*********************************************************************⌋
     ↪   ********************/
    /* This is the main program that solves the Population Balance
     ↪   Equation (PBE)
     * We have some classes to help us solve the model:
     *  - Grid              :: Contains all variables needed for Gaussian
↪   quadrature rule
     *  - Fluid             :: Contains density, surface tension and
↪   viscosity for a fluid
     *  - SystemProperties  :: Contains variables such as maximum radius,
↪   volume of tanks etc
     *  - Constants         :: Contains parameters such as
↪   k1,k2,k3,k4,kb1,kb2,kc1,kc2
     *  - Kernels           :: Contains kernels for breakage (KBB,kDB)
↪   and coalescence (KBC,KDC)
     *  - PBModel           :: Solves the entire model by the use of an
↪   ODE solver:
     *                         Utilizes all above classes
     */

    /*********************************************************************⌋
     ↪   ********************/
    /* Declaration of variables
     ↪                      */
    /*********************************************************************⌋
     ↪   ********************/
    const size_t Np = 200;                      /* Number of grid points
     ↪     */
    char const *fileName = "crudeB.csv";        /* Experimental data
     ↪     */
    realtype kb1 = 7.e-6,   /* From testLogNormalInitialCondition.cpp */
             kb2 = 2.e-4,   /* From testLogNormalInitialCondition.cpp */
             kc1 = 1.e-4,   /* From testLogNormalInitialCondition.cpp */
             kc2 = 3.e2;    /* From testLogNormalInitialCondition.cpp */
    kb1 = 8.e-6;
    kb2 = kb2;
    kc1 = 5.e-4;
    kc2 = 4.e2;

    size_t Npts = 100;
    std::vector<gsl_matrix *> residualVector(Npts*Npts, nullptr); /* 19x19 matrix of matrices */
    std::vector<PBModel *> m(Npts*Npts, nullptr);
    std::vector<realtype> k1vec(Npts, 0);
    std::vector<realtype> k2vec(Npts, 0);
    realtype k1lb = log10(5.e-5), k1ub = log10(5.e-3);
    realtype k2lb = log10(40), k2ub = log10(4000);
    size_t i, j, q, r;
    for (i = 0; i < Npts; i++){
        k1vec[i] = pow(10.0, (realtype) i / (Npts-1) * (k1ub-k1lb) + k1lb);
        k2vec[i] = pow(10.0, (realtype) i / (Npts-1) * (k2ub-k2lb) + k2lb);
    }
    realtype k1, k2;

    /*****************************************************************************************/
    /* Instantiation and solution                                                            */
    /*****************************************************************************************/
    /* Helper classes */
    Grid g = Grid(Np, 0, 1, 0, 0, 2);
    Fluid disp = Fluid(0.837e3, 22.0e-3, 16.88e-3); /* Oil       */
    Fluid cont = Fluid(1.0e3, 1, 1);                /* Water     */
    SystemProperties s = SystemProperties(500.0e-6, 725.0e-6, 0.366, disp);
    realtype summation = 0;
    size_t index_ij = 0, N = 0, M = 0;
    for (i = 0; i < Npts; i++){
        k1 = k1vec[i];
        for (j = 0; j < Npts; j++){
            index_ij = i*Npts + j;
            k2 = k2vec[j];
            m[index_ij] = new PBModel(fileName, kb1, kb2, k1, k2, g, s, cont, disp, 0);
            m[index_ij]->solvePBE();
            M = m[index_ij]->getM();
            N = m[index_ij]->getN();
            residualVector[index_ij] = gsl_matrix_alloc(M, N);
            for (q = 0; q < M; q++) {
                for (r = 0; r < N; r++) {
                    gsl_matrix_set(
                            residualVector[index_ij], q, r,
                            m[index_ij]->getResidualij(q, r)
                    );
                }
            }
            delete m[index_ij];
        }
    }

    /* Write to file */
    time_t rawtime;
    struct tm * timeinfo;
    char buffer[80];
    time (&rawtime);
    timeinfo = localtime(&rawtime);
    strftime(buffer,sizeof(buffer),"%d-%m-%Y-%I:%M:%S",timeinfo);
    std::string str(buffer);

//    std::ofstream fparams("../results/paramsBreakage.dat");
    std::ofstream fparams("../results/parameterFiles/paramsCoalescence" +
    ↪  str + ".dat");
    std::ofstream fresiduals("../results/residualFiles/residualsDynamics"
    ↪  + str + ".dat");
//    std::ofstream fresiduals("../results/residualsNoDynamics.dat");
    fparams << "#k1,#k2\n";
    for (i = 0; i < Npts; i++) {
        fparams << k1vec[i] << "," << k2vec[i] << std::endl;
    }
    fresiduals << "#residuals\n";
    for (i = 0; i < Npts; i++){
        for (j = 0; j < Npts; j++) {
            index_ij = i * Npts + j;
            gsl_matrix *currentMat = residualVector[index_ij];
            for (q = 0; q < M; q++) {
                for (r = 0; r < N; r++) {
                    fresiduals << gsl_matrix_get(currentMat, q, r) << ",";
                }
                fresiduals << std::endl;
            }
        }
    }
    fparams.close();
    fresiduals.close();
    return 0;
}
```

Listing B.5: checkSensitivity.cpp: Program that was used in the coalescence experiment to chart objective functions

```
//
// Created by Sindre Bakke Øyen on 18.04.2018.
```

```
//

#include <iostream>
#include <chrono>
#include <ctime>

/* User-defined header files */
#include "PBModel.h"

int main() {
    /**********************************************************************
     ↪   ********************/
    /* Description of program
     ↪                          */
    /**********************************************************************
     ↪   ********************/
    /* This is the main program that solves the Population Balance
     ↪   Equation (PBE)
     * We have some classes to help us solve the model:
     *  - Grid              :: Contains all variables needed for Gaussian
↪   quadrature rule
     *  - Fluid             :: Contains density, surface tension and
↪   viscosity for a fluid
     *  - SystemProperties  :: Contains variables such as maximum radius,
↪   volume of tanks etc
     *  - Constants         :: Contains parameters such as
↪   k1,k2,k3,k4,kb1,kb2,kc1,kc2
     *  - Kernels           :: Contains kernels for breakage (KBB,kDB)
↪   and coalescence (KBC,KDC)
     *  - PBModel           :: Solves the entire model by the use of an
↪   ODE solver:
     *                         Utilizes all above classes
     */

    /**********************************************************************
     ↪   ********************/
    /* Declaration of variables
     ↪                          */
    /**********************************************************************
     ↪   ********************/
    const size_t Np = 400;                      /* Number of grid points
     ↪     */
    char const *fileName = "crudeB.csv";        /* Experimental data
     ↪     */
//     realtype kb1 = 2.6426477281e-07,           /* Model fitted
↪   parameters  */
//              kb2 = 0,
//              kc1 = 1.0138643992e+00*kb1,
//              kc2 = 1.6791362386e-01;
    realtype kb1 = 7.e-6,
            kb2 = 5.e-4,
            kc1 = 5.e-4*0,
            kc2 = 6.e2;
    /*********************************************************************
    ↪   ********************/
    /* Instantiation and solution
    ↪                         */
    /*********************************************************************
    ↪   ********************/
    /* Helper classes */
    Grid g = Grid(Np, 0, 1, 0, 0, 2);
    Fluid disp = Fluid(0.837e3, 22.0e-3, 16.88e-3); /* Oil      */
    Fluid cont = Fluid(1.0e3, 1, 1);                /* Water    */
    SystemProperties s = SystemProperties(500.0e-6, 725.0e-6, 0.366,
    ↪   disp);
    PBModel m = PBModel(fileName, kb1, kb2, kc1, kc2, g, s, cont, disp,
    ↪   1);
    m.solvePBE();
    m.exportFv();
    return 0;
}
```

Listing B.6: testLogNormalInitialCondition.cpp: Program that was used to do dynamic simulations on log-normal initial conditions

```
#include <iostream>
#include <chrono>
#include <ctime>
#include <gsl/gsl_vector_double.h>

/* User-defined header files */
#include "PBModel.h"

int main() {
    /*********************************************************************
    ↪   ********************/
    /* Description of program
    ↪                         */
    /*********************************************************************
    ↪   ********************/
    /* This is the main program that solves the Population Balance
    ↪   Equation (PBE)
     * We have some classes to help us solve the model:
     *   - Grid              :: Contains all variables needed for Gaussian
↪   quadrature rule
```

```
     *  - Fluid            :: Contains density, surface tension and viscosity for a fluid
     *  - SystemProperties  :: Contains variables such as maximum radius, volume of tanks etc
     *  - Constants         :: Contains parameters such as k1,k2,k3,k4,kb1,kb2,kc1,kc2
     *  - Kernels           :: Contains kernels for breakage (KBB,kDB) and coalescence (KBC,KDC)
     *  - PBModel           :: Solves the entire model by the use of an ODE solver:
     *                         Utilizes all above classes
     */

    /**********************************************************************************/
    /* Declaration of variables                                                       */
    /**********************************************************************************/
    const size_t Np = 200;                    /* Number of grid points      */
    char const *fileName = "crudeB.csv";      /* Experimental data          */
    realtype kb1 = 8.e-6,
             kb2 = 2.e-4,
             kc1 = 1.e-4,
             kc2 = 3.e2;
    /* Guess for parameter estimating residuals */
//    kb1 = 2.99e-5, kb2 = 5.556421e-4, kc1 = 5.57417e-4, kc2 = 227.596; /* Refined */
//    kb1 = 4.64e-5, kb2 = 4.64e-4, kc1 = 1.00e-3, kc2 = 2.28e2; /* Coarse */
//    kb1*=1; kb2*=1; kc1*= 1; kc2*= 1;

    /* Guess for parameter estimating weighted residuals */
//    kb1 = 7.1905e-05, kb2 = 8.01876e-4, kc1 = 7.46537e-4, kc2 = 151.12; /* Refined */
//    kb1 = 4.64e-5, kb2 = 4.64e-4, kc1 = 1.00e-3, kc2 = 2.28e2; /* Coarse */

    /* Guess for parameter estimating means */
//    kb1 = 2.68e-4, kb2 = 4.98e-3, kc1 = 7.20e-5, kc2 = 2.94e1; /* Refined */
//    kb1 = 3.59381e-4, kb2 = 5.99484e-3, kc1 = 1.29155e-4, kc2 = 227.592; /* Coarse */



```

```

    /* Solve with parameters from optimizing residuals */
    kb1 = 3.90e-5, kb2 = 5.12e-4, kc1 = 9.89e-4, kc2 = 2.86e2;  /* Coarse
     ↪  parameter estimation */
//    kb1 = 2.92088e-05, kb2 = 0.000601003, kc1 = 0.000573217, kc2 =
↪  251.809; /* Refined  */

    /* Solve with parameters from optimizing weighted residuals */
//    kb1 = 3.92e-5, kb2 = 5.80e-4, kc1 = 9.06e-4, kc2 = 2.76e2;  /*
↪  Coarse parameter estimation */

    /* Solve with parameters from optimizing means */
//    kb1 = 2.48e-4, kb2 = 5.04e-3, kc1 = 6.28e-5, kc2 = 3.34e1;  /*
↪  Refined parameter estimation */

    /* Solve with parameters from Dakota */
//    kb1 = 3.e-6, kb2 = 2.e-4, kc1 = 3.77e-3, kc2 = 8.89e2;
    /*********************************************************************
     ↪  ********************/
    /* Instantiation and solution
     ↪                        */
    /*********************************************************************
     ↪  ********************/
    /* Helper classes */
    Grid g = Grid(Np, 0, 1, 0, 0, 2);
    Fluid disp = Fluid(0.837e3, 22.0e-3, 16.88e-3); /* Oil      */
    Fluid cont = Fluid(1.0e3, 1, 1);                /* Water    */
    SystemProperties s = SystemProperties(500.0e-6, 725.0e-6, 0.366,
     ↪  disp);
    PBModel m = PBModel(fileName, kb1, kb2, kc1, kc2, g, s, cont, disp,
     ↪  0);
//    m.paramesterEstimationSSE();
//    m.parameterEstimationMean();
    m.solvePBE();
//    m.exportFvSimulatedWithExperimental();
//    m.exportFv();
    m.exportMeans();
    return 0;
}
```

Listing B.7: testSolution.cpp: Program that was used to do dynamic simulations on a specific parameter combination

```
//
// Created by Sindre Bakke Øyen on 05.03.2018.
//
```

```cpp

#include "Fluid.h"

/* Constructors */
Fluid::Fluid() : rho(0), sigma(0), nu(0) {}

Fluid::Fluid(const Fluid &f) : rho(f.getRho()), sigma(f.getSigma()),
 ↪  nu(f.getNu()){}

Fluid::Fluid(realtype rho, realtype sigma, realtype nu) : rho(rho),
 ↪  sigma(sigma), nu(nu) {}

/* Getter methods */
realtype Fluid::getRho() const {
    return rho;
}

realtype Fluid::getSigma() const {
    return sigma;
}

realtype Fluid::getNu() const {
    return nu;
}

std::ostream &operator<<(std::ostream &os, const Fluid &fluid) {
    os << "rho: " << fluid.rho << " sigma: " << fluid.sigma << " nu: " <<
     ↪  fluid.nu;
    return os;
}

/* Destructors */
Fluid::~Fluid(){
}
```

Listing B.8: Fluid.cpp: C++ class to characterize fluids

```cpp
//
// Created by Sindre Bakke Øyen on 06.03.2018.
//

#include "Grid.h"
#include <iostream>

/* Constructors */
Grid::Grid() = default;
```

```cpp

Grid::Grid(const Grid &g):N(g.getN()), x0(g.getX0()), x1(g.getX1()),
 ↪ alpha(g.getAlpha()),
                          beta(g.getBeta()), mu0(g.getMu0()),
                          xi(gsl_vector_alloc(g.getN())),
                          w(gsl_vector_alloc(g.getN())),
                          D(gsl_matrix_alloc(g.getN(), g.getN())),
                          xipBB(gsl_matrix_alloc(g.getN(), g.getN())),
                          xipBC(gsl_matrix_alloc(g.getN(), g.getN())),
                          xippBC(gsl_matrix_alloc(g.getN(), g.getN()))
{
    /* Memory has been allocated, copy values (not pointers) */
    gsl_vector_memcpy(this->xi, g.getXi());
    gsl_vector_memcpy(this->w, g.getW());
    gsl_matrix_memcpy(this->D, g.getD());
    gsl_matrix_memcpy(this->xipBB, g.getXipBB());
    gsl_matrix_memcpy(this->xipBC, g.getXipBC());
    gsl_matrix_memcpy(this->xippBC, g.getXippBC());

}

Grid::Grid(size_t N, realtype x0, realtype x1, realtype alpha, realtype
 ↪ beta, realtype mu0)
        : N(N), x0(x0), x1(x1), alpha(alpha), beta(beta), mu0(mu0) {
    this->w = gsl_vector_alloc(N);
    this->xi = gsl_vector_alloc(N);
    this->xipBB = gsl_matrix_alloc(N, N);
    this->xipBC = gsl_matrix_alloc(N, N);
    this->xippBC = gsl_matrix_alloc(N, N);
    this->D = gsl_matrix_alloc(N, N);

    /* Set xi, w, remap to [x0, x1] and derivative */
    this->setQuadratureRule();  /* Sets xi and w */
    this->remapGrid();                      /* Remaps from [-1, 1] to
     ↪ [x0, x1] */
    this->setLagrangeDerivativeMatrix();    /* In domain [x0, x1] as set
     ↪ by constructor */

    /* Set rescaled xis */
    this->setInterpolatedXis();
}

void Grid::coefs(size_t j, realtype *r){
    /* Returns coefficients for three-term recurrence relationship for
     ↪ Jacobi polynomials
  * The coefficients are
  *       r[0] = aj
  *       r[1] = bj
 *      r[2] = cj
 */
    /* Note: j runs from 0 */
    realtype a, div2, div3;
    if ((alpha == beta) == -0.5){
        r[0] = 2;
        r[1] = 0;
        r[2] = 1;
    } else {
        a = (2*j+alpha+beta);
        r[0] = (a+1)*(a+2)/(2*(j+1)*(j+alpha+beta+1));
        div2 = (2*(j+1)*(j+alpha+beta+1)*a);
        if (div2 == 0){
            r[1] = 0;
        } else{
            r[1] = (a+1)*(SUNRpowerI(alpha,2)-SUNRpowerI(beta,2))/div2;
        }
        div3 = ((j+1)*(j+alpha+beta+1)*a);
        if (div3 == 0){
            r[2] = 0;
        } else{
            r[2] = (j+alpha)*(j+beta)*(a+2)/div3;
        }
    }

}

/* Setter methods */
void Grid::setQuadratureRule() {
    /* Computes the N-point Gauss Lobatto quadrature rule with Jacobi
     ↪  polynomials
     * The function uses the Golub Welsch algorithm,
     *      xi = eigenvalues of Jtilde, wi = mu0 * (eigenvector vi of
 ↪  Jtilde)^2
     * Input args:
     * alpha:: Coefficient to determine spacing of quadrature points
 ↪  (alpha=beta=0 means Legendre)
     * beta :: Coefficient to determine spacing of quadrature points
     * mu0  :: Integral of weight function from a to b (a=-1, b=1 for
 ↪  Legendre)
     * */
    /* Declare variables */

    gsl_matrix *J, *Jtilde; /* Matrices used to obtain xi and w
     ↪        */
    gsl_matrix *I;          /* Identity matrix (size NxN)
     ↪        */
    gsl_matrix *JmI, *JpI;  /* Matrices J-I and J+I
     ↪          */
    gsl_matrix *T;          /* Matrix to solve for gammap and taup+1
     ↪          */
    gsl_matrix_view subJt;  /* Submatrix of Jtilde
     ↪          */
    gsl_matrix *eVecs;      /* Matrix of eigenvectors of Jtilde
     ↪          */

    gsl_permutation *P1;    /* Permutation matrix for LU factorization
     ↪          */
    gsl_permutation *P2;    /* Permutation matrix for LU factorization
     ↪          */

    gsl_vector *ep;         /* Basis vector (zero except for pth element
     ↪          */
    gsl_vector *eta, *mu;   /* Eigenvectors of J+-I, lambdas = +-1
     ↪          */
    gsl_vector *rhs;        /* Right hand side for retrieving gammap and
     ↪   taup+1 */
    gsl_vector *taugamma;   /* Vector to hold gammaP and tauN+1
     ↪          */
    gsl_vector *eVals;      /* Vector of eigenvalues of Jtilde
     ↪          */

    size_t i, j;            /* Iterators
     ↪          */
    int k = 2;              /* Signum of permutation
     ↪          */
    realtype gammai;        /* Used to store value for super and
     ↪   subdiagonal    */
    realtype taui;          /* Used to store value for main diagonal
     ↪          */
    realtype gammaP;        /* Last value of gamma
     ↪          */
    realtype tauPp1;        /* Last value of tau
     ↪          */
    realtype etaP, muP;     /* Last values of vectors eta and mu
     ↪          */
    realtype r1[3] = {0};   /* Holds coefficients from coefs function
     ↪          */
    realtype r2[3] = {0};   /* Holds coefficients from coefs function
     ↪          */
    realtype tmp;           /* Temporary variable
     ↪          */

    /* Allocate memory for all needed matrices and vectors */
    J       = gsl_matrix_calloc(N-1, N-1);
    Jtilde   = gsl_matrix_calloc(N, N);
    I        = gsl_matrix_alloc(N-1, N-1);
    JmI      = gsl_matrix_alloc(N-1, N-1);
    JpI      = gsl_matrix_alloc(N-1, N-1);
    T        = gsl_matrix_alloc(2, 2);
    subJt    = gsl_matrix_submatrix(Jtilde, 0, 0, J->size1, J->size2);
    eVecs    = gsl_matrix_alloc(N, N);

    P1 = gsl_permutation_alloc(N-1);
    P2 = gsl_permutation_alloc(2);

    ep       = gsl_vector_alloc(N-1);
    eta      = gsl_vector_alloc(N-1);
    mu       = gsl_vector_alloc(N-1);
    rhs      = gsl_vector_alloc(2);
    taugamma= gsl_vector_alloc(2);
    eVals    = gsl_vector_alloc(N);

    /* Construct J (normally used to calculate the Gauss quadrature rule)
     ↪   */
    for (i = 0; i < N-2; i++){
        this->coefs(i, r1);
        this->coefs(i+1, r2);
        taui   = r1[1]/r1[0];
        gammai = sqrt(r2[2]/(r1[0]*r2[0]));
        gsl_matrix_set(J, i, i, taui);       // Diagonal
        gsl_matrix_set(J, i, i+1, gammai);   // Superdiagonal
        gsl_matrix_set(J, i+1, i, gammai);   // Subdiagonal
    }

    /* Create all vectors and matrices to create Jtilde */
    gsl_vector_set_basis(ep, ep->size-1); // (MxN)(Nx1) = (Mx1),
     ↪   (J-I)eta=ep => ep in R^(Mx1)
    gsl_matrix_set_identity(I);
    gsl_matrix_memcpy(JmI, J);
    gsl_matrix_memcpy(JpI, J);
    gsl_matrix_sub(JmI, I);
    gsl_matrix_add(JpI, I);

    gsl_vector_set(rhs, 0, -1);
    gsl_vector_set(rhs, 1, 1);

    gsl_linalg_LU_decomp(JmI, P1, &k);
    gsl_linalg_LU_solve(JmI, P1, ep, mu);
    gsl_linalg_LU_decomp(JpI, P1, &k);
    gsl_linalg_LU_solve(JpI, P1, ep, eta);

    etaP = gsl_vector_get(eta, eta->size-1);
    muP = gsl_vector_get(mu, mu->size-1);

    gsl_matrix_set(T, 0, 0, 1);
    gsl_matrix_set(T, 0, 1, -etaP);
    gsl_matrix_set(T, 1, 0, 1);
    gsl_matrix_set(T, 1, 1, -muP);

    /* Enforce xi0 = -1 and xiN = 1 */
    gsl_linalg_LU_decomp(T, P2, &k);
    gsl_linalg_LU_solve(T, P2, rhs, taugamma);

    tauPp1 = gsl_vector_get(taugamma, 0);
    gammaP = sqrt(gsl_vector_get(taugamma, 1));

    gsl_matrix_swap(&subJt.matrix, J);

    gsl_matrix_set(Jtilde, Jtilde->size1-2, Jtilde->size2-1, gammaP);
    gsl_matrix_set(Jtilde, Jtilde->size1-1, Jtilde->size2-2, gammaP);
    gsl_matrix_set(Jtilde, Jtilde->size1-1, Jtilde->size2-1, tauPp1);

    /* The eigenvalues of Jtilde are the quadrature points and
     * the first value of each eigenvector is used to obtain the
↪   quadrature weights
     * */
    gsl_eigen_symmv_workspace *ws;
    ws = gsl_eigen_symmv_alloc(N);
    gsl_eigen_symmv(Jtilde, eVals, eVecs, ws);

    /* Find quadrature points */
    gsl_vector_swap(xi, eVals);
    /* Find weights */
    for (i=0; i < N; i++){
        tmp = gsl_matrix_get(eVecs, 0, i);
        tmp *= tmp;
        gsl_vector_set(w, i, tmp*mu0);
    }

    /* xi and w are reversed. 3/4 of the xi and w are sorted. Sort with
     ↪  insertion sort */
    gsl_vector_reverse(xi);
    gsl_vector_reverse(w);
    i = 1;
    while (i < xi->size){
        j = i;
        while ((j > 0) && (gsl_vector_get(xi, j-1) > gsl_vector_get(xi,
         ↪  j))){
            /* Some value is less than previous: swap elements until it
             ↪  is not */
```

```
                gsl_vector_swap_elements(xi, j, j-1);
                gsl_vector_swap_elements(w, j, j-1);  /* Remember that w_i
                 ↪  corresponds to xi_i: swap accordingly */
                j--;
            }
            i++;
        }

        /* END OF FUNCTION: FREE ALLOCATED MEMORY */
        gsl_matrix_free(J);
        gsl_matrix_free(Jtilde);
        gsl_matrix_free(I);
        gsl_matrix_free(JmI);
        gsl_matrix_free(JpI);
        gsl_matrix_free(T);
        gsl_matrix_free(eVecs);

        gsl_permutation_free(P1);
        gsl_permutation_free(P2);

        gsl_vector_free(ep);
        gsl_vector_free(eta);
        gsl_vector_free(mu);
        gsl_vector_free(rhs);
        gsl_vector_free(taugamma);
        gsl_vector_free(eVals);
        gsl_eigen_symmv_free(ws);
    }

    void Grid::remapGrid(){
        /* Remap xi from [-1,1] to physical domain [this->x0, this->x1] */
        realtype a0 = gsl_vector_get(this->xi, 0);
        realtype b0 = gsl_vector_get(this->xi, this->N-1);

        gsl_vector_scale(this->xi, (this->x1-this->x0)/(b0-a0));
        gsl_vector_add_constant(this->xi, -a0*(this->x1-this->x0)/(b0-a0));

        gsl_vector_scale(this->w, (this->x1-this->x0)/(b0-a0));
    }

    void Grid::setLagrangeDerivativeMatrix(){
        realtype s = 1;
        realtype wi, wj;
        size_t i, j, k;
        for (i = 1; i < this->N+1; i++){        /* Loop on rows */
            s = 1;
            for (k = 0; k < this->N; k++){
                if (k != i-1){
```

```
                s /= gsl_vector_get(this->xi, i-1) -
                 ↪  gsl_vector_get(this->xi, k);
            }
        }
        wi = s;
        for (j = 1; j < this->N+1; j++){   /* Loop on columns (l_j(x_i))
         ↪   */
            if (i==j){ /* Diagonal */
                s = 0;
                for (k = 0; k < this->N; k++){
                    if (k != i-1){
                        s += 1 / (gsl_vector_get(this->xi, i-1) -
                         ↪  gsl_vector_get(this->xi, k));
                    }
                }
                gsl_matrix_set(D, i-1, j-1, s);
            } else{ /* Off-diagonal */
                s = 1;
                for (k = 0; k < this->N; k++){
                    if (k != j-1){
                        s /= gsl_vector_get(this->xi, j-1) -
                         ↪  gsl_vector_get(this->xi, k);
                    }
                }
                wj = s;
                gsl_matrix_set(D, j-1, i-1, wi /
                        (wj * (gsl_vector_get(this->xi,
                         ↪  j-1)-gsl_vector_get(this->xi, i-1))));
            }

        }
    }
}

void Grid::setInterpolatedXis(){
    size_t i, j;
    realtype ai, bj, tmp;
    realtype exp1, exp2;
    for (i = 0; i < this->N; i++){
        ai = gsl_vector_get(xi, i);
        for (j = 0; j < this->N; j++){
            bj   = gsl_vector_get(xi, j);
            exp1 = SUNRpowerR(bj, 3.0)/2;
            tmp  = 1.0 - exp1;
            exp2 = SUNRpowerR(tmp, 1.0/3);
            gsl_matrix_set(xipBB, i, j, (1.0-ai)*bj+ai);
             ↪     /* Breakage birth     */
```

```
                gsl_matrix_set(xipBC, i, j, SUNRpowerR(2.0, -1.0/3)*ai*bj);
                 ↪     /* Coalescence birth */
                gsl_matrix_set(xippBC, i, j, ai*exp2);
                 ↪     /* Coalescence birth */
            }
        }
}

/* Getter methods */
size_t Grid::getN() const {
    return N;
}

realtype Grid::getX0() const {
    return x0;
}

realtype Grid::getX1() const {
    return x1;
}

realtype Grid::getAlpha() const {
    return alpha;
}

realtype Grid::getBeta() const {
    return beta;
}

realtype Grid::getMu0() const {
    return mu0;
}

gsl_vector *Grid::getXi() const {
    return xi;
}

gsl_vector *Grid::getW() const {
    return w;
}

gsl_matrix *Grid::getD() const {
    return D;
}

gsl_matrix *Grid::getXipBB() const {
    return xipBB;
}
```

```cpp

gsl_matrix *Grid::getXipBC() const {
    return xipBC;
}

gsl_matrix *Grid::getXippBC() const {
    return xippBC;
}

std::ostream &operator<<(std::ostream &os, const Grid &grid) {
    size_t i, j, N = grid.getN();
    gsl_vector *xi  = grid.getXi();
    gsl_vector *w   = grid.getW();
    gsl_matrix *D   = grid.getD();
    gsl_matrix *xipBB = grid.getXipBB();
    gsl_matrix *xipBC = grid.getXipBC();
    gsl_matrix *xippBC = grid.getXippBC();

    os << "xi (Quadrature points):\n";
    for (i = 0; i < N; i++){
        os << std::setprecision(3) << gsl_vector_get(xi, i) << "\t";
    }
    os << "\n\nw (Quadrature weights):\n";
    for (i = 0; i < N; i++){
        os << std::setprecision(3) << gsl_vector_get(w, i) << "\t";
    }
    os << "\n\nD (Lagrange derivative matrix):\n";
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            os << std::setw(8) << std::setprecision(3) <<
             ↪  gsl_matrix_get(D, i, j) << "\t";
        }
        os << "\n";
    }
    os << "\n\nxipBB (Quadrature points birth breakage):\n";
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            os << std::setw(8) << std::setprecision(3) <<
             ↪  gsl_matrix_get(xipBB, i, j) << "\t";
        }
        os << "\n";
    }
    os << "\n\nxipBC (Quadrature points birth coalescence):\n";
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            os << std::setw(8) << std::setprecision(3) <<
             ↪  gsl_matrix_get(xipBC, i, j) << "\t";
        }
```

```
            os << "\n";
        }
        os << "\n\nxippBC (Quadrature points birth coalescence):\n";
        for (i = 0; i < N; i++){
            for (j = 0; j < N; j++){
                os << std::setw(8) << std::setprecision(3) <<
                 ↪  gsl_matrix_get(xippBC, i, j) << "\t";
            }
            os << "\n";
        }
        return os;
}

/* Destructors */
Grid::~Grid(){
    gsl_vector_free(this->xi);
    gsl_vector_free(this->w);
    gsl_matrix_free(this->D);
    gsl_matrix_free(this->xipBB);
    gsl_matrix_free(this->xipBC);
    gsl_matrix_free(this->xippBC);
}
```

Listing B.9: Grid.cpp: C++ class to set Gaussian grids with derivatives and integral weights

```
//
// Created by Sindre Bakke Øyen on 07.03.2018.
//

#include <gsl/gsl_matrix.h>
#include "Kernels.h"

/* Constructors */
Kernels::Kernels() = default;

Kernels::Kernels(const Kernels &k){
    size_t N;
    N = k.getKBB()->size1;

    /* Allocate memory for kernels */
    this->KBB = gsl_matrix_alloc(N, N);
    this->KBC = gsl_matrix_alloc(N, N);
    this->KDC = gsl_matrix_alloc(N, N);
    this->KDB = gsl_vector_alloc(N);
    /* Copy the values (not pointers) so we don't get memory leak */
    gsl_matrix_memcpy(this->KBB, k.getKBB());
    gsl_matrix_memcpy(this->KBC, k.getKBC());
    gsl_matrix_memcpy(this->KDC, k.getKDC());
    gsl_vector_memcpy(this->KDB, k.getKDB());
}

Kernels::Kernels(realtype kb1, realtype kb2, realtype kc1, realtype kc2,
 ↪  realtype tf,
                 const Grid &grid, const SystemProperties &sysProps,
                 const Fluid &cont, const Fluid &disp) :
    kb1(kb1), kb2(kb2), kc1(kc1), kc2(kc2), tf(tf), grid(grid),
    KBB(gsl_matrix_calloc(grid.getN(), grid.getN())),
    KBC(gsl_matrix_calloc(grid.getN(), grid.getN())),
    KDC(gsl_matrix_calloc(grid.getN(), grid.getN())),
    KDB(gsl_vector_calloc(grid.getN())){
    this->initializeKs(cont, disp, sysProps);
    this->setBreakageKernels();
    this->setCoalescenceKernels();
}

void Kernels::setBreakageKernels() {
    size_t i, j, N;
    realtype xi_i, xipBBij;
    gsl_vector *xi;
    gsl_matrix *xipBB;

    N = grid.getN();
    xi = grid.getXi();
    xipBB = grid.getXipBB();

    for (i = 1; i < N; i++){
        xi_i = gsl_vector_get(xi, i);

        /* Death breakage */
        gsl_vector_set(this->KDB, i,
                       this->k1
                       *1/SUNRpowerR(xi_i, (realtype) 2.0/3.0)
                       *SUNRexp(-this->k2/SUNRpowerR(xi_i, (realtype)
                        ↪  5.0/3.0)));
        for (j = 1; j < N; j++){
            xipBBij = gsl_matrix_get(xipBB, i, j);

            /* Birth breakage */
            gsl_matrix_set(this->KBB, i, j,
                           this->k1
                           *(2*1/(realtype)SUNRpowerR(xipBBij, (realtype)
                            ↪  2.0/3.0))
```

```
                                *SUNRexp(-this->k2/(realtype)SUNRpowerR(xipBBi⌋
                                 ↪  j, (realtype)
                                 ↪  5.0/3.0))
                                *(2.4/(realtype)SUNRpowerI(xipBBij, 3))
                                *SUNRexp(-4.5*(realtype)SUNRpowerI(
                                        2*(realtype)SUNRpowerI(xi_i, 3) -
                                         ↪  (realtype)SUNRpowerI(xipBBij, 3), 2
                                ) /(realtype)SUNRpowerI(xipBBij, 6)
                                )
                                *3*(realtype)SUNRpowerI(xi_i, 2)
                );
            }
        }
//      /* Set first value to 0, to avoid NaN */
//      gsl_vector_set(KDB, 0, 0);
//      gsl_matrix_set(KBB, 0, 0, 0);
}

void Kernels::setCoalescenceKernels() {
    size_t i, j, N;
    realtype xi_i, xi_j, xipBCij, xippBCij;

    gsl_vector *xi      = grid.getXi();
    gsl_matrix *xipBC   = grid.getXipBC();
    gsl_matrix *xippBC  = grid.getXippBC();

    N  = grid.getN();
    for (i = 1; i < N; i++){
        xi_i = gsl_vector_get(xi, i);
        for (j = 1; j < N; j++){
            xi_j     = gsl_vector_get(xi, j);
            xipBCij  = gsl_matrix_get(xipBC, i, j);
            xippBCij = gsl_matrix_get(xippBC, i, j);
            /* Birth coalescence */
            gsl_matrix_set(this->KBC, i, j,
                               this->k3
                               *(realtype)SUNRpowerI(xipBCij+xippBCij, 2)
                               *(realtype)SUNRpowerR(
                                       (realtype)SUNRpowerR(xipBCij,
                                        ↪  (realtype) 2.0/3.0)
                                       +(realtype)SUNRpowerR(xippBCij,
                                        ↪  (realtype) 2.0/3.0),
                                       (realtype) 1.0/2.0
                               )*(realtype)SUNRexp(
                                       -this->k4
                                       *(realtype)SUNRpowerR(1/xipBCij+1/xipp⌋
                                        ↪  BCij,
                                        ↪  (realtype)-5.0/6.0)
```

```
                              )
               );

               /* Death coalescence */
               gsl_matrix_set(this->KDC, i, j,
                              this->k3
                              *(realtype)SUNRpowerI(xi_j+xi_i, 2)
                              *(realtype)SUNRpowerR(
                                      (realtype)SUNRpowerR(xi_j,
                                       ↪  (realtype)2.0/3.0)
                                      +(realtype)SUNRpowerR(xi_i,
                                       ↪  (realtype)2.0/3.0),
                                      (realtype)1.0/2.0
                              )*(realtype)SUNRexp(
                                      -this->k4
                                      *(realtype)SUNRpowerR(1/xi_j + 1/xi_i,
                                       ↪  (realtype)-5.0/6.0)
                              )
               );
           }
       }
}

/* Setter methods */
void Kernels::initializeKs(const Fluid &cont, const Fluid &disp, const
 ↪  SystemProperties &s){
    realtype Rm = s.getRm();
    realtype eps = s.getEps();
    realtype rhoc = cont.getRho();
    realtype rhod = disp.getRho();
    realtype sigma = disp.getSigma();
    realtype Vm = s.getVm();

    realtype R23 = SUNRpowerR(Rm, 2.0/3);
    realtype R53 = SUNRpowerR(Rm, 5.0/3);
    realtype R73 = SUNRpowerR(Rm, 7.0/3);
    realtype R56 = SUNRpowerR(Rm, 5.0/6);
    realtype e13 = SUNRpowerR(eps, 1.0/3);
    realtype e23 = SUNRpowerR(eps, 2.0/3);
    realtype t13 = SUNRpowerR(2.0, 1.0/3);
    realtype t23 = SUNRpowerR(2.0, 2.0/3);
    realtype t53 = SUNRpowerR(2.0, 5.0/3);
    realtype rho12 = SUNRpowerR(rhoc, 1.0/2);
    realtype sigma12 = SUNRpowerR(sigma, 1.0/2);

    this->k1 = tf*kb1*e13/(t23*R23)*SUNRsqrt(rhod/rhoc);
    this->k2 = kb2*sigma / (rhod*t53*e23*R53);
    this->k3 = tf/Vm*R73*4*t13*kc1*e13;
    this->k4 = kc2*R56*rho12*e13/(2*sigma12);
}

void Kernels::setTf(realtype tf) {
    this->k1 *= tf / this->tf;
    this->k3 *= tf / this->tf;
    Kernels::tf = tf;
    this->setBreakageKernels();
    this->setCoalescenceKernels();
}

void Kernels::setKb1(realtype kb1) {
    Kernels::kb1 = kb1;
}

void Kernels::setKb2(realtype kb2) {
    Kernels::kb2 = kb2;
}

void Kernels::setKc1(realtype kc1) {
    Kernels::kc1 = kc1;
}

void Kernels::setKc2(realtype kc2) {
    Kernels::kc2 = kc2;
}

void Kernels::setNewK1(realtype kb1) {
    this->k1 *= kb1 / this->kb1;
    this->setKb1(kb1);
}

void Kernels::setNewK2(realtype kb2) {
    this->k2 *= kb2 / this->kb2;
    this->setKb2(kb2);
}

void Kernels::setNewK3(realtype kc1) {
    this->k3 *= kc1 / this->kc1;
    this->setKc1(kc1);
}

void Kernels::setNewK4(realtype kc2) {
    this->k4 *= kc2 / this->kc2;
    this->setKc2(kc2);
}

```

```
void Kernels::setNewKs(realtype kb1, realtype kb2, realtype kc1, realtype
 ↪  kc2){
    this->setNewK1(kb1);
    this->setNewK2(kb2);
    this->setNewK3(kc1);
    this->setNewK4(kc2);
    /* Also get new kernels because of new k's */
    this->setBreakageKernels();
    this->setCoalescenceKernels();
}

/* Getter methods */
gsl_matrix *Kernels::getKBB() const {
    return KBB;
}

gsl_matrix *Kernels::getKBC() const {
    return KBC;
}

gsl_matrix *Kernels::getKDC() const {
    return KDC;
}

gsl_vector *Kernels::getKDB() const {
    return KDB;
}

realtype Kernels::getK1() const {
    return k1;
}

realtype Kernels::getK2() const {
    return k2;
}

realtype Kernels::getK3() const {
    return k3;
}

realtype Kernels::getK4() const {
    return k4;
}

realtype Kernels::getKb1() const {
    return kb1;
}

```

```
realtype Kernels::getKb2() const {
    return kb2;
}

realtype Kernels::getKc1() const {
    return kc1;
}

realtype Kernels::getKc2() const {
    return kc2;
}

realtype Kernels::getTf() const {
    return tf;
}

const Grid &Kernels::getGrid() const {
    return grid;
}

/* Relational operators */
Kernels& Kernels::operator=(const Kernels &rhs){
    this->k1 = rhs.getK1();
    this->k2 = rhs.getK2();
    this->k3 = rhs.getK3();
    this->k4 = rhs.getK4();
    this->kb1 = rhs.getKb1();
    this->kb2 = rhs.getKb2();
    this->kc1 = rhs.getKc1();
    this->kc2 = rhs.getKc2();
    this->tf = rhs.getTf();
    size_t N = rhs.getGrid().getN();
    this->KBB = gsl_matrix_calloc(N, N);
    this->KBC = gsl_matrix_calloc(N, N);
    this->KDC = gsl_matrix_calloc(N, N);
    this->KDB = gsl_vector_calloc(N);
    return *this;
}

std::ostream& operator<<(std::ostream &os, const Kernels &kernels) {
    size_t i, j, N;
    gsl_matrix *KBB = kernels.getKBB();
    gsl_vector *KDB = kernels.getKDB();
    gsl_matrix *KBC = kernels.getKBC();
    gsl_matrix *KDC = kernels.getKDC();

    N = KDB->size;
    os << "KBB (Kernel birth breakage):\n";
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            os << std::setw(8) << std::setprecision(3) <<
             ↪  gsl_matrix_get(KBB, i, j) << "\t";
        }
        os << "\n";
    }
    os << "\n\nKDB (Kernel death breakage):\n";
    for (i = 0; i < N; i++){
            os << std::setw(8) << std::setprecision(3) <<
             ↪  gsl_vector_get(KDB, i) << "\t";
    }
    os << "\n\nKBC (Kernel birth coalescence):\n";
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            os << std::setw(8) << std::setprecision(3) <<
             ↪  gsl_matrix_get(KBC, i, j) << "\t";
        }
        os << "\n";
    }
    os << "\n\nKDC (Kernel death coalescence):\n";
    for (i = 0; i < N; i++){
        for (j = 0; j < N; j++){
            os << std::setw(8) << std::setprecision(3) <<
             ↪  gsl_matrix_get(KDC, i, j) << "\t";
        }
        os << "\n";
    }
    return os;
}

Kernels::~Kernels() {
    gsl_vector_free(this->KDB);
    gsl_matrix_free(this->KBB);
    gsl_matrix_free(this->KBC);
    gsl_matrix_free(this->KDC);
}
```

Listing B.10: Kernels.cpp: C++ class to characterize kernels

```
//
// Created by Sindre Bakke Øyen on 18.03.2018.
//

#include <gsl/gsl_vector_double.h>
#include <gsl/gsl_matrix.h>
```

```
#include <gsl/gsl_multifit_nlinear.h>
#include "PBModel.h"

/* Constructors */
PBModel::PBModel() = default;

PBModel::PBModel(char const *f, realtype kb1, realtype kb2, realtype kc1,
 ↪  realtype kc2,
                  const Grid &g, const SystemProperties &s,
                  const Fluid &cont, const Fluid &disp,
                  size_t decision):
        filename(f), grid(g), sysProps(s), cont(cont), disp(disp),
         ↪  cvode_mem(nullptr),
        kerns(Kernels(kb1, kb2, kc1, kc2, 1, g, s, cont, disp)){
    int flag = 0;
    /* Get rows and columns of csv file and initialize M and N
     ↪  respectively */
    this->getRowsAndCols();

    /* Allocate memory for member variables */
    this->t      = gsl_vector_alloc(this->M);
    this->r      = gsl_vector_alloc(this->N);
    this->fv     = gsl_matrix_alloc(this->M, this->N);
    this->tau    = gsl_vector_alloc(this->M);
    this->fvSim  = gsl_matrix_calloc(this->M, this->N);

    /* These must be on another domain (xi, not r) */
    this->psi    = gsl_matrix_calloc(this->M, grid.getN());
    this->NPsi   = N_VNew_Serial(grid.getN());

    /* Set experimental data: r, t, fv and rescale */
    this->getDistributions();
    if (decision) {
        std::ifstream fin("../results/logNormal_mean_30_sd_10.txt");
        std::string line;
        getline(fin, line);
        size_t i = 0;
        while (getline(fin, line)){
            i++;
        }
        gsl_vector_free(this->r);
        this->r = gsl_vector_calloc(i);
        gsl_matrix_free(this->fv);
        this->fv = gsl_matrix_calloc(this->M, i);
        gsl_matrix_free(this->fv);
        this->fvSim = gsl_matrix_calloc(this->M, i);
        fin.clear();
        fin.seekg(0, fin.beg);
        getline(fin, line);
        realtype val1 = 0, val2 = 0;
        size_t j = 0;
        while (std::getline(fin, line)){
            std::stringstream linestream(line);
            linestream >> val1 >> val2;
            gsl_vector_set(r, j, val1);
            gsl_matrix_set(this->fv, 0, j, val2);
            j++;
        }
        fin.close();
        this->N = i;
        gsl_vector_view fvj0 = gsl_matrix_row(this->fv, 0);
        for (j = 1; j < this->M; j++){
            gsl_vector_view fvjj = gsl_matrix_row(this->fv, j);
            gsl_vector_memcpy(&fvjj.vector, &fvj0.vector);
        }
    }

    this->rescaleInitial();

    /* Set final time of experiment and update kernels */
    realtype tf = gsl_vector_get(this->t, this->M - 1);
    this->kerns.setTf(tf);
    /* Assign nondimensional time tau = t / tf */
    gsl_vector_memcpy(this->tau , this->t);
    gsl_vector_scale(this->tau, 1/this->kerns.getTf());
    this->tRequested = gsl_vector_get(this->tau, 1);

    /* Assign nondimensional initial distribution */
    this->psiN    = gsl_matrix_row(this->psi, 0);

    /* Prepare CVode memory */
    flag = this->prepareCVMemory();
    if (flag == 1) perror("Failed to prepare ODE memory");
}


/* Helper methods */
void PBModel::getRowsAndCols(){
    FILE *f = fopen(filename, "r");
    if (f != nullptr) {
        // Initialize variables
        size_t rows = 0, cols = 0;
        size_t i=0, j=1000, tmp = 0;
        bool flag = false;
        realtype val = 0;
        const char s[2] = ",";
        const int bufSize = 1000;
        char line[bufSize], *toFree;

        // Now loop over lines
        while(fgets(line, sizeof line, f) != nullptr) {
            rows++;
            tmp = 0;
            i = 0;
            toFree = strdup(line);
            while ((strsep(&toFree, s)) != nullptr) {
                tmp++;
                if (rows > TRASHROWS && tmp > TRASHCOLS){
                    if (toFree != nullptr) {
                        val = strtod(toFree, nullptr);
                    } else val = 1;
                    if (val != 0) flag = true;
                    else if (flag && val == 0){
                        i++;
                    }
                }
            }
            flag = false;
            if (rows > TRASHROWS && j > i) j = i;
            if (tmp > cols) {
                cols = tmp;
            }
        }
        if (j > TRUNCATETHRESHOLD){
            cols = cols - TRASHCOLS - j + TRUNCATETHRESHOLD;
        } else cols = cols - TRASHCOLS;
        rows = rows - TRASHROWS;
        this->M = rows;
        this->N = cols;
        fclose(f);
    } else {
        perror(filename);
    }
}

void PBModel::getDistributions() {
    /* Declare needed variables */
    char *hours, *minutes;
    realtype h, m;
    size_t i = 0, k = 0; // Index variables

    const char s[2] = ",";
    const size_t bufSize = 10*(this->N);
    char line[bufSize], *token, *toFree, *timeStr;

    /* Open file and start reading */
    FILE *f = fopen(filename, "r");
    if (f != nullptr) {
        while (fgets(line, sizeof line, f) != nullptr) {
            if (i == 0) {
                i++;
                continue;
            }
            toFree = strdup(line);
            k = 0;
            while((token = strsep(&toFree, s)) != nullptr){
                if (k < TRASHCOLS){
                    if (k == 2){
                        if (i==1){
                            k++;
                            continue;
                        } else {
                            // Fetch time column
                            timeStr = strsep(&token, " ");
                            timeStr = token;
                            hours = strsep(&timeStr, ":");
                            minutes = timeStr;
                            h = strtod(hours, nullptr);
                            m = strtod(minutes, nullptr);
                            gsl_vector_set(t, i-TRASHROWS, h*3600+m*60);
                            k++;
                            continue;
                        }
                    } else {
                        k++;
                        continue;
                    }
                }
                if (k-TRASHCOLS < this->N) {
                    switch (i) {
                        case 1: {
                            /* Given sizes are diameter; we need radii.
                             ↪  Also they should be microns */
                            gsl_vector_set(r, k - TRASHCOLS,
                             ↪  strtod(token, nullptr) / 2 * 1.e-6);
                            break;
                        }
                        default: {
                            gsl_matrix_set(fv, i - TRASHROWS, k -
                             ↪  TRASHCOLS, strtod(token, nullptr));
                            break;
                        }
                    }
                    k++;
                } else continue;
            }
            i++;
        }
        fclose(f);
        gsl_vector_add_constant(t, -gsl_vector_get(t, 0));
        for (i = 1; i < t->size; i++){
            if (gsl_vector_get(t, i) - gsl_vector_get(t, i-1) == 0){
                gsl_vector_set(t, i, gsl_vector_get(t, i) + 30);
                // t_data[i] += 30;
            }
        }
        gsl_vector_set(t, 0, 0);
        if (gsl_vector_get(t, 1) - gsl_vector_get(t, 0) == 0){
            gsl_vector_set(t, 1, gsl_vector_get(t, 1) + 30);
        }
    } else perror(filename);
}

void PBModel::rescaleInitial(){
    /* Takes distributions, corresponding radii and phase fraction and
     ↪  rescales the distributions
     * fv = phi/I * f0
     * Approximate I by trapezoids: I = sum([r(i+1)-r(i)] *
 ↪  [f(i+1)+f(i)]) from i=0 to N-1
     */
    size_t i, j;
    realtype I, rj, rjj, fij, fijj;

    for (i = 0; i < M; i++){ /* Loop over rows */
        I  = 0;
        for ( j = 0; j < N-1; j++ ){ /* Loop over columns */
            rj  = gsl_vector_get(r, j); rjj = gsl_vector_get(r, j+1);
            fij = gsl_matrix_get(fv, i, j); fijj = gsl_matrix_get(fv, i,
             ↪  j+1);
            I  += ( rjj-rj ) * ( fijj+fij );
        }
        I = I/2;
        gsl_vector_view rowJ = gsl_matrix_row(fv, i);
        gsl_vector_scale(&rowJ.vector, PHI/I);
    }
}

int PBModel::preparePsi(){
    realtype xN, yN;
    realtype *psiData = NV_DATA_S(this->NPsi);
    /* Allocate memory for Steffen spline on experimental psi */
    gsl_vector_view fv0 = gsl_matrix_row(this->fv, 0);
    gsl_vector *psi0 = gsl_vector_alloc(this->N);
    gsl_vector_memcpy(psi0, &fv0.vector);
    gsl_vector_scale(psi0, this->sysProps.getRm());

    /* Create temporary experimental radius / Rm */
    gsl_vector *tmp = gsl_vector_alloc(this->N);
    gsl_vector_memcpy(tmp, this->r);
    gsl_vector_scale(tmp, 1/sysProps.getRm());

    gsl_interp_accel *acc = gsl_interp_accel_alloc();
    gsl_spline *spline = gsl_spline_alloc(gsl_interp_steffen, this->N);
    gsl_spline_init(spline, &tmp->data[0], &psi0->data[0], this->N);

    for (size_t i = 0; i < grid.getN(); i++){
        xN = gsl_vector_get(grid.getXi(), i);
        /* We are not allowed to "interpolate" outside of experimental
         ↪  radius */
        if (xN > gsl_vector_get(tmp, 0) && xN < gsl_vector_get(tmp,
         ↪  this->N-1)) {
            yN = gsl_spline_eval(spline, xN, acc);
        } else yN = 0; /* If we are outside of experimental radius, 0 our
         ↪  distribution */
        gsl_vector_set(&psiN.vector, i, yN);
        /* Assign initial condition to solution vector */
        psiData[i] = yN;
    }

    gsl_vector_free(psi0);
    gsl_vector_free(tmp);
    gsl_spline_free (spline);
    gsl_interp_accel_free (acc);

    return 0;
}

int PBModel::prepareCVMemory(){
    int flag = 0;
    /* Call CVodeCreate to create the solver memory and specify the
     * Backward Differentiation Formula and the use of a Newton iteration
↪  */
    this->cvode_mem = CVodeCreate(CV_BDF, CV_NEWTON);
    if (checkFlag((void *)this->cvode_mem, "CVodeCreate", 0)) return(1);

    /* Call CVodeInit to initialize the integrator memory and specify the
     * user's right hand side function in y'=f(t,y), the inital time T0,
↪  and
```

```
     * the initial dependent variable vector y. */
    flag = CVodeInit(this->cvode_mem, dydt, gsl_vector_get(this->tau, 0),
    ↪  this->NPsi);
    if (checkFlag(&flag, "CVodeInit", 1)) return(1);

    /* Call CVodeSStolerances to specify the scalar relative tolerance
     * and scalar absolute tolerance */
    flag = CVodeSStolerances(this->cvode_mem, RTOL, ATOL);
    if (checkFlag(&flag, "CVodeSStolerances", 1)) return(1);

    /* Set the pointer to user-defined data */
    flag = CVodeSetUserData(cvode_mem, this);
    if(checkFlag(&flag, "CVodeSetUserData", 1)) return(1);

    /* Create dense SUNMatrix for use in linear solves */
    this->A = SUNDenseMatrix(this->grid.getN(), this->grid.getN());
    if(checkFlag((void *)this->A, "SUNDenseMatrix", 0)) return(1);

    /* Create dense SUNLinearSolver object for use by CVode */
    this->LS = SUNDenseLinearSolver(this->NPsi, this->A);
    if(checkFlag((void *)this->LS, "SUNDenseLinearSolver", 0)) return(1);

    /* Call CVDlsSetLinearSolver to attach the matrix and linear solver
    ↪  to CVode */
    flag = CVDlsSetLinearSolver(this->cvode_mem, this->LS, this->A);
    if(checkFlag(&flag, "CVDlsSetLinearSolver", 1)) return(1);
    return flag;
}

int PBModel::releaseCVMemory(){
    CVodeFree(&this->cvode_mem);
    SUNMatDestroy(this->A);
    SUNLinSolFree(this->LS);
    return 0;
}

int PBModel::checkFlag(void *flagvalue, const char *funcname, int opt){
    int *errflag;

    /* Check if SUNDIALS function returned NULL pointer - no memory
    ↪  allocated */
    if (opt == 0 && flagvalue == NULL) {
        fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed - returned NULL
        ↪  pointer\n\n",
                funcname);
        return(1); }

        /* Check if flag < 0 */
    else if (opt == 1) {
        errflag = (int *) flagvalue;
        if (*errflag < 0) {
            fprintf(stderr, "\nSUNDIALS_ERROR: %s() failed with flag =
            ↪  %d\n\n",
                    funcname, *errflag);
            return(1); }}

        /* Check if function returned NULL pointer - no memory allocated
        ↪  */
    else if (opt == 2 && flagvalue == NULL) {
        fprintf(stderr, "\nMEMORY_ERROR: %s() failed - returned NULL
        ↪  pointer\n\n",
                funcname);
        return(1); }

    return(0);
}

bool PBModel::checkMassBalance() {
    /* Only to be called after solvePBE method */
    size_t i = 0;
    realtype phaseFraction = 0;
    realtype phasef = 0;
    for (i = 0; i < this->M; i++){
        gsl_vector_view psii = gsl_matrix_row(this->psi, i);
        gsl_vector_view fvi  = gsl_matrix_row(this->fvSim, i);
        gsl_blas_ddot(this->grid.getW(), &psii.vector, &phaseFraction);
        gsl_blas_ddot(this->r, &fvi.vector, &phasef);
        if ((realtype) SUNRabs(phaseFraction - PHI)/PHI * 100 > 5){
            /* More than relative 5% deviation. Mass not conserved */
            return false;
        }
    }
    /* At no time the mass was not conserved --> mass was conserved,
    ↪  return true */
    return true;
}


/* Solver methods */
int PBModel::getRHS(N_Vector y, N_Vector ydot){
    size_t i = 0, j = 0;
    double integralBB; /* Stores integral value for BB */
    double integralBC; /* Stores integral value for BC */
    double integralDC; /* Stores integral value for DC */
    realtype *ydata = NV_DATA_S(y);
    realtype *ydotdata = NV_DATA_S(ydot);

    /* Fetch different grids */
    size_t Np           = this->grid.getN();
    gsl_vector *w       = grid.getW();
    gsl_vector *xi      = grid.getXi();
    gsl_matrix *xipBB   = grid.getXipBB();
    gsl_matrix *xipBC   = grid.getXipBC();
    gsl_matrix *xippBC  = grid.getXippBC();

    /* Fetch different kernels */
    gsl_matrix *KBB     = kerns.getKBB();
    gsl_vector *KDB     = kerns.getKDB();
    gsl_matrix *KBC     = kerns.getKBC();
    gsl_matrix *KDC     = kerns.getKDC();

    /* Point psiN.vector to ydata */
    this->psiN.vector.data = ydata;

    /* Allocate memory for our interpolated distributions */
    gsl_matrix *psipBB   = gsl_matrix_alloc(Np, Np);
    gsl_matrix *psipBC   = gsl_matrix_alloc(Np, Np);
    gsl_matrix *psippBC = gsl_matrix_alloc(Np, Np);

    /* Interpolate from (x,y)-pairs to (xx,yy)-pairs */
    interpolatePsi(xi, &this->psiN.vector, xipBB, psipBB);
    interpolatePsi(xi, &this->psiN.vector, xipBC, psipBC);
    interpolatePsi(xi, &this->psiN.vector, xippBC, psippBC);

    /* Allocate memory for integrands (and integrals?) */
    /* IBB, IBC, IDC (and BB, DB, BC, DC?) */
    gsl_matrix *IBB = gsl_matrix_calloc(Np, Np);
    gsl_matrix *IBC = gsl_matrix_calloc(Np, Np);
    gsl_matrix *IDC = gsl_matrix_calloc(Np, Np);
    gsl_vector *B   = gsl_vector_calloc(Np);
    gsl_vector *C   = gsl_vector_calloc(Np);

    /* Open two files to write B and C to file */
    std::ofstream bin("../results/breakage.dat", std::fstream::app);
    std::ofstream cin("../results/coalescence.dat", std::fstream::app);
    bin << this->tRequested * gsl_vector_get(this->t, this->M-1) << "\t";
    cin << this->tRequested / gsl_vector_get(this->t, this->M-1) << "\t";
    /* B = IBB*w - DB, C = IBC*w - IDC*w */
    /* ydot[i] = B[i] + C[i] */
    /* Loop over rows and columns and evaluate RHS */
    for (i = 1; i < Np; i++){
        realtype kdbi = gsl_vector_get(KDB, i);
        realtype xii = gsl_vector_get(xi, i);
        realtype psii = gsl_vector_get(&this->psiN.vector, i);
        for (j = 0; j < Np; j++){
            /* Fetch indexed variables for easier typing */
            realtype xipbbij = gsl_matrix_get(xipBB, i, j);
            realtype xipbcij = gsl_matrix_get(xipBC, i, j);
            realtype xippbcij = gsl_matrix_get(xippBC, i, j);
            realtype xij = gsl_vector_get(xi, j);
            realtype kbbij = gsl_matrix_get(KBB, i, j);
            realtype kbcij = gsl_matrix_get(KBC, i, j);
            realtype kdcij = gsl_matrix_get(KDC, i, j);
            realtype psipbbij = gsl_matrix_get(psipBB, i, j);
            realtype psipbcij = gsl_matrix_get(psipBC, i, j);
            realtype psippbcij = gsl_matrix_get(psippBC, i, j);
            realtype psij = gsl_vector_get(&this->psiN.vector, j);

            /* Set integrand birth breakage */
            realtype denomBB = SUNRpowerI(xipbbij, 3);
            if (denomBB == 0) { gsl_matrix_set(IBB, i, j, 0); }
            else {
                gsl_matrix_set(IBB, i, j,
                               kbbij * psipbbij
                               / denomBB
                               * (1 - xii)
                );
            }
            /* Set integrand birth coalescence */
            realtype denomBC1 = SUNRpowerI(xipbcij, 3);
            realtype denomBC2 = SUNRpowerI(xippbcij, 3);
            if (denomBC1 == 0 || denomBC2 == 0){ gsl_matrix_set(IBC, i,
             ↪  j, 0); }
            else {
                gsl_matrix_set(IBC, i, j,
                               kbcij
                               * psipbcij / denomBC1
                               * psippbcij / denomBC2
                               * (realtype) SUNRpowerI(xii / xippbcij, 2)
                               * xii / (realtype) SUNRpowerR(2.0, 1.0 /
                               ↪  3.0)
                );
            }
            /* Set integrand death coalescence */
            realtype denomDB = SUNRpowerI(xij, 3);
            if (denomDB == 0) { gsl_matrix_set(IDC, i, j, 0); }
            else {
                gsl_matrix_set(IDC, i, j,
                               kdcij * psij / denomDB
                );
            }
        }
        /* Now do inner product of integrand and weights to evaluate
         ↪  integrals */
        /* Populate B by BB - DB
         * BB[i] =  xii^3 * IBB[i, :] * w = xii^3 * ddot(IBB[i, :], w)
         */
        gsl_vector_view IBBrow = gsl_matrix_row(IBB, i);
        gsl_vector_view IBCrow = gsl_matrix_row(IBC, i);
        gsl_vector_view IDCrow = gsl_matrix_row(IDC, i);

        /* Breakage */
        gsl_blas_ddot(w, &IBBrow.vector, &integralBB);
        /* B[i] = Birth breakage[i] - Death breakage[i] */
        gsl_vector_set(B, i,
                       (realtype)SUNRpowerI(xii, 3)
                       *(realtype)integralBB
                       -kdbi*psii
        );

        /* Coalescence */
        gsl_blas_ddot(w, &IBCrow.vector, &integralBC);
        gsl_blas_ddot(w, &IDCrow.vector, &integralDC);
        /* C[i] = Birth coalescence[i] - Death coalescence[i] */
        gsl_vector_set(C, i,
                       (realtype)SUNRpowerI(xii, 3)
                       *(realtype)integralBC
                       -psii*(realtype)integralDC
        );
        ydotdata[i] = (realtype) (gsl_vector_get(B, i) +
         ↪  gsl_vector_get(C, i));

        /* Write coalescence and breakage to file */
        bin << gsl_vector_get(B, i) << "\t";
        cin << gsl_vector_get(C, i) << "\t";
    }
    ydotdata[0] = 0;
    bin << std::endl;
    cin << std::endl;
    bin.close();
    cin.close();
    /* Free allocated memory that is only used in current scope */
    gsl_matrix_free(psipBB);
    gsl_matrix_free(psipBC);
    gsl_matrix_free(psippBC);
    gsl_matrix_free(IBB);
    gsl_matrix_free(IBC);
    gsl_matrix_free(IDC);
    gsl_vector_free(B);
    gsl_vector_free(C);
    return 0;
}

int PBModel::interpolatePsi(const gsl_vector *x, const gsl_vector *y,
↪  const gsl_matrix *xx, gsl_matrix *yy){
    /* x and y is original data, xx and yy is interpolated data */
    size_t i=0, j=0;
    realtype xN, yN;
    realtype x0   = gsl_vector_get(x, 0);
    realtype xend = gsl_vector_get(x, x->size - 1);
    gsl_interp_accel *acc = gsl_interp_accel_alloc();

    gsl_spline *spline = gsl_spline_alloc(gsl_interp_steffen, x->size);
    gsl_spline_init(spline, x->data, y->data, x->size);
    /* Interpolate onto xx domain and get yy values */
    for (i = 0; i < xx->size1; i++) {      /* Loop over rows        */
        for (j = 0; j < xx->size2; j++) { /* Loop over columns     */
            xN = gsl_matrix_get(xx, i, j);
            if (xN > x0 && xN < xend) {
                yN = gsl_spline_eval(spline, xN, acc);
            } else yN = 0; /* If we are outside of experimental radius, 0
             ↪  our distribution */
            gsl_matrix_set(yy, i, j, yN);
        }
    }
    gsl_spline_free (spline);
    gsl_interp_accel_free (acc);
    return 0;
}

int PBModel::interpolateFv(const gsl_vector *x, const gsl_vector *y,
↪  const gsl_vector *xx, gsl_vector *yy){
    /* x and y is original data. xx and yy is interpolated data */
    size_t i=0;
    realtype xN, yN;
    realtype x0   = gsl_vector_get(x, 0);
    realtype xend = gsl_vector_get(x, x->size -1);
    gsl_interp_accel *acc = gsl_interp_accel_alloc();

    gsl_spline *spline = gsl_spline_alloc(gsl_interp_steffen, x->size);
    gsl_spline_init(spline, x->data, y->data, x->size);
    /* Interpolate onto xx domain and get yy values */
    for (i = 0; i < xx->size; i++){
        xN = gsl_vector_get(xx, i);
        if (xN > x0 && xN < xend){
            yN = gsl_spline_eval(spline, xN, acc);
        } else yN = 0;
        gsl_vector_set(yy, i, yN);
    }
    gsl_spline_free (spline);
    gsl_interp_accel_free(acc);
    return 0;
}

int PBModel::timeIterate() {
    int flag = CVode(this->cvode_mem, this->tRequested, this->NPsi,
    ↪   &(this->tout), CV_NORMAL);
    if(checkFlag(&flag, "CVode", 1)) return 1;
//    std::cout << "time requested: " << this->tRequested << ", time
↪   produced: " << this->tout << std::endl;
    return 0;
}

int PBModel::solvePBE(){
    int flag = 0;
    /* Prepare psi for CVode */
    flag = this->preparePsi();
    if (flag != 0) perror("Failed to interpolate fv onto psi");
    /* Prepare memory for integration */
    flag = this->releaseCVMemory();
    if (flag == 1) perror("Failed to release ODE memory");
    flag = this->prepareCVMemory();
    if (flag == 1) perror("Failed to prepare ODE memory");

    size_t i = 0;
    /* Write breakage and coalescence contributions to file */
    std::ofstream bin("../results/breakage.dat");
    std::ofstream cin("../results/coalescence.dat");
    bin << "xi\n";
    cin << "xi\n";
    for (i = 0; i < this->grid.getN(); i++){
        bin << gsl_vector_get(this->grid.getXi(), i) << "\t";
        cin << gsl_vector_get(this->grid.getXi(), i) << "\t";
    }
    bin << std::endl;
    cin << std::endl;
    bin.close();
    cin.close();

    for (i = 1; i < this->M; i++){
        /* Request new return time for ODE solver */
        this->tRequested = gsl_vector_get(this->tau, i);

        /* Take one time iteration at solving the ODE */
        flag = this->timeIterate();
        if (flag == 1) { break; }

        /* Copy data of current row into psi (psiN->data points to
         ↪  solution data) */
        gsl_vector_view row = gsl_matrix_row(this->psi, i);
        gsl_vector_memcpy(&row.vector, &(this->psiN.vector));
    }

    /** Interpolate solution back onto experimental radial domain   **/
    /** i.e. psi(xi, tau) --> fv(r, t)                              **/
    /* Matrix fvSim will hold fv on experimental domain             */
    /* Create temporary matrix to hold fv on discretized domain     */
    gsl_matrix *tmpPsi   = gsl_matrix_alloc(this->M, this->grid.getN());
    /* Create temporary vector to hold simulated radii on [0, Rm]   */
    gsl_vector *tmpR     = gsl_vector_alloc(this->grid.getN());
    /* Copy original data */
    gsl_vector_memcpy(tmpR, this->grid.getXi());
    gsl_matrix_memcpy(tmpPsi, this->psi);
    /* Scale original data */
    gsl_vector_scale(tmpR, sysProps.getRm());
    gsl_matrix_scale(tmpPsi, 1/sysProps.getRm());

    for (i = 0; i < this->M; i++) {
        gsl_vector_view tmpPsii   = gsl_matrix_row(tmpPsi, i);
        gsl_vector_view fvSimi = gsl_matrix_row(this->fvSim, i);
        interpolateFv(tmpR, &tmpPsii.vector, this->r, &fvSimi.vector);
    }
//    this->printFvSimulated();
    gsl_matrix_free(tmpPsi);
    gsl_vector_free(tmpR);
    return 0;
}


realtype PBModel::getResidualij(size_t i, size_t j){
    /* fvSim was set in solvePBE method and should by now
     * hold simulated fv on experimental radial domain
     */
    realtype fvSimVal = gsl_matrix_get(this->fvSim, i, j);
    realtype fvExpVal = gsl_matrix_get(this->fv, i, j);
    return (fvSimVal - fvExpVal);
}

double PBModel::getModeledMean(size_t t){
    double s, rdfvsim, dr;
    s = 0;
    size_t i = 0;
    for (i = 1; i < this->N; i++){
        dr = gsl_vector_get(this->r, i) - gsl_vector_get(this->r, i-1);
```

```cpp
        rdfvsim = gsl_vector_get(this->r, i)
                  * (gsl_matrix_get(this->fvSim, t, i-1) +
                  ↪  gsl_matrix_get(this->fvSim, t, i));
        s += dr * rdfvsim / 2;
    }
    return (s / PHI * 1.e6);
}

double PBModel::getExperimentalMean(size_t t) {
    double s, rdfvsim, dr;
    s = 0;
    size_t i = 0;
    for (i = 1; i < this->N; i++){
        dr = gsl_vector_get(this->r, i) - gsl_vector_get(this->r, i-1);
        rdfvsim = gsl_vector_get(this->r, i)
                  * (gsl_matrix_get(this->fv, t, i-1) +
                  ↪  gsl_matrix_get(this->fv, t, i));
        s += dr * rdfvsim / 2;
    }
    return (s / PHI * 1.e6);
}

realtype PBModel::getResidualMean(size_t t){
    /* t is time instant */
    return (this->getModeledMean(t) - this->getExperimentalMean(t));
}

double PBModel::getWeightedResidual(size_t i, size_t j, double m, double
 ↪  s){
    double x = gsl_vector_get(this->r, j);
    double weight = this->getWeight(x, m, s);
    return (this->getResidualij(i, j) * weight);

    return 1.0;
}

double PBModel::getWeight(double x, double m, double s) {
    return (1.0 / (1.0+exp((m-x)/s)));
}


/* Levenberg-Marquardt parameter estimation */
int PBModel::costFunctionSSE(const gsl_vector *x, gsl_vector *f) {
    /* Evaluates the cost function at x
     * x is the vector of parameters kb1, kb2, kc1, kc2 */
    realtype kb1 = gsl_vector_get(x, 0) / 1.e5;
    realtype kb2 = gsl_vector_get(x, 1) / 1.e4;
    realtype kc1 = gsl_vector_get(x, 2) / 1.e4;
    realtype kc2 = gsl_vector_get(x, 3) / 1.e-2;
    this->kerns.setNewKs(kb1, kb2, kc1, kc2);
    this->solvePBE();
    double s = 1.e-6, m = 6.e-6;
    size_t i = 0, j = 0;
    size_t times = f->size / this->N;
    for (i = 0; i < times-1; i++){
        for (j = 0; j < this->N; j++){
            size_t idx = i * this->N + j;
//            gsl_vector_set(f, idx, this->getResidualij(i, j));
            gsl_vector_set(f, idx, this->getWeightedResidual(i, j, m, s));
        }
    }
    for (j = 0; j < this->N; j++){
        size_t idx = (times-1)*this->N + j;
//        gsl_vector_set(f, idx, this->getResidualij(M-1, j));
        gsl_vector_set(f, idx, this->getWeightedResidual(M-1, j, m, s));
    }
    return GSL_SUCCESS;
}

int PBModel::costFunctionMean(const gsl_vector *x, gsl_vector *f){
    double kb1 = gsl_vector_get(x, 0) / 1.e4;
    double kb2 = gsl_vector_get(x, 1) / 1.e3;
    double kc1 = gsl_vector_get(x, 2) / 1.e5;
    double kc2 = gsl_vector_get(x, 3) / 1.e-1;
    this->kerns.setNewKs(kb1, kb2, kc1, kc2);
    this->solvePBE();
    size_t i = 0;
    size_t nRes = f->size; /* Number of residual means */
    for (i = 0; i < nRes; i++){
        gsl_vector_set(f, i, this->getResidualMean(i));
    }
    return GSL_SUCCESS;
}

int PBModel::paramesterEstimationSSE() {
    const size_t Ntmin = 81;     /* Minimum number of distributions chosen
     ↪     */
    const size_t Ntmax = 90;     /* Maximum number of distributions chosen
     ↪     */
    size_t Nt = Ntmin;           /* Number of distributions chosen
     ↪     */
    const size_t p = 4;          /* Number of parameters
     ↪     */
    const realtype kb1 = this->getKerns().getKb1();
    const realtype kb2 = this->getKerns().getKb2();
    const realtype kc1 = this->getKerns().getKc1();
    const realtype kc2 = this->getKerns().getKc2();
    do {
        size_t N = Nt * this->N;       /* Number of residuals */
        size_t n = N;

        const gsl_multifit_nlinear_type *T = gsl_multifit_nlinear_trust;
        gsl_multifit_nlinear_workspace *w;
        gsl_multifit_nlinear_fdf fdf;
        gsl_multifit_nlinear_parameters fdf_params =
                gsl_multifit_nlinear_default_parameters();
        fdf_params.h_df = 1.e-2;

        std::cout << "Number of distributions: " << Nt << std::endl;

        gsl_vector *f;  /* Function */
        gsl_matrix *J;  /* Jacobian */
        gsl_matrix *covar = gsl_matrix_alloc(p, p);

        PBModel *d = this;
        /* starting values */
        double x1_scaling = 1.e5, x2_scaling = 1.e4, x3_scaling = 1.e4,
         ↪  x4_scaling = 1.e-2;
        double x_init[4] = {kb1 * x1_scaling, kb2 * x2_scaling,
                            kc1 * x3_scaling, kc2 * x4_scaling};
        gsl_vector_view x = gsl_vector_view_array(x_init, p);
        double chisq, chisq0;
        int status, info;

        const double xtol = 1e-8;
        const double gtol = 1e-8;
        const double ftol = 1.e-4;

        /* define the function to be minimized */
        fdf.f = gatewayCostSSE;
        fdf.df = NULL;      /* set to NULL for finite-difference Jacobian
         ↪  */
        fdf.fvv = NULL;     /* not using geodesic acceleration */
        fdf.n = n;
        fdf.p = p;
        fdf.params = d;

        /* allocate workspace with default parameters */
        w = gsl_multifit_nlinear_alloc(T, &fdf_params, n, p);

        /* initialize solver with starting point and weights */
        gsl_multifit_nlinear_init(&x.vector, &fdf, w);

        /* compute initial cost function */
        f = gsl_multifit_nlinear_residual(w);
        gsl_blas_ddot(f, f, &chisq0);

        /* solve the system with a maximum of 200 iterations */
        status = gsl_multifit_nlinear_driver(200, xtol, gtol, ftol,
                                             paramEstimationCallbackSSE,
                                              ↪  NULL, &info, w);

        /* compute covariance of best fit parameters */
        J = gsl_multifit_nlinear_jac(w);
        gsl_multifit_nlinear_covar(J, 0.0, covar);

        /* compute final cost */
        gsl_blas_ddot(f, f, &chisq);

#define FIT(i) gsl_vector_get(w->x, i)
#define ERR(i) sqrt(gsl_matrix_get(covar,i,i))

        time_t rawtime;
        struct tm *timeinfo;
        char buffer[80];
        time(&rawtime);
        timeinfo = localtime(&rawtime);
        strftime(buffer, sizeof(buffer), "%d-%m-%Y-%I:%M:%S", timeinfo);
        std::string str(buffer);
        std::stringstream ss;
        ss << Nt;
        std::string outputFilename =
         ↪  "../results/parameterEstimation/wSSE/refined_initial_guess/"
         ↪  + ss.str() + "_dists_" + str + ".dat";
        std::ofstream outfile;
        outfile.open(outputFilename);

        fprintf(stderr, "summary from method '%s/%s'\n",
                gsl_multifit_nlinear_name(w),
                gsl_multifit_nlinear_trs_name(w));
        fprintf(stderr, "number of iterations: %zu\n",
                gsl_multifit_nlinear_niter(w));
        fprintf(stderr, "function evaluations: %zu\n", fdf.nevalf);
        fprintf(stderr, "Jacobian evaluations: %zu\n", fdf.nevaldf);
        fprintf(stderr, "reason for stopping: %s\n",
                (info == 1) ? "small step size" : "small gradient");
        fprintf(stderr, "initial |f(x)| = %f\n", sqrt(chisq0));
        fprintf(stderr, "final   |f(x)| = %f\n", sqrt(chisq));

        {
            double dof = n - p;
            double c = GSL_MAX_DBL(1, sqrt(chisq / dof));

            fprintf(stderr, "chisq/dof = %g\n", chisq / dof);

            fprintf(stderr, "kb1      = %.3g +/- %.3g\n", FIT(0), c *
            ↪  ERR(0));
            fprintf(stderr, "kb2      = %.3g +/- %.3g\n", FIT(1), c *
            ↪  ERR(1));
            fprintf(stderr, "kc1      = %.3g +/- %.3g\n", FIT(2), c *
            ↪  ERR(2));
            fprintf(stderr, "kc2      = %.3g +/- %.3g\n", FIT(3), c *
            ↪  ERR(3));

            outfile << "#kb1,kb2,kc1,kc2,kb10,kb20,kc10,kc20,chisq/dof,#d⌋
            ↪  ists,initial,final,iter\n";
            outfile << FIT(0) / x1_scaling << "," << FIT(1) / x2_scaling
                    << "," << FIT(2) / x3_scaling << "," << FIT(3) /
                    ↪  x4_scaling
                    << "," << x_init[0] / x1_scaling << "," << x_init[1]
                    ↪  / x2_scaling
                    << "," << x_init[2] / x3_scaling << "," << x_init[3]
                    ↪  / x4_scaling
                    << "," << chisq / dof << "," << Nt
                    << "," << sqrt(chisq0) << "," << sqrt(chisq)
                    << "," << gsl_multifit_nlinear_niter(w) << "\n";
            outfile << c * ERR(0) / x1_scaling << "," << c * ERR(1) /
            ↪  x2_scaling
                    << "," << c * ERR(2) / x3_scaling << "," << c *
                    ↪  ERR(3) / x4_scaling << "\n";
            outfile << fdf_params.h_df << "\n";
        }
        outfile.close();
        fprintf(stderr, "status = %s\n", gsl_strerror(status));

        gsl_multifit_nlinear_free(w);
        gsl_matrix_free(covar);
        Nt++;
    } while (Nt < Ntmax);
    return 0;
}

int PBModel::parameterEstimationMean() {
    const size_t N = this->M;
    const size_t p = 4;
    const size_t n = N;

    const gsl_multifit_nlinear_type *T = gsl_multifit_nlinear_trust;
    gsl_multifit_nlinear_workspace *w;
    gsl_multifit_nlinear_fdf fdf;
    gsl_multifit_nlinear_parameters fdf_params =
            gsl_multifit_nlinear_default_parameters();
    fdf_params.h_df = 1.e-2;

    gsl_vector *f;  /* Function */
    gsl_matrix *J;  /* Jacobian */
    gsl_matrix *covar = gsl_matrix_alloc(p, p);

    PBModel *d = this;
    /* starting values */
    double x1_scaling = 1.e4, x2_scaling = 1.e3, x3_scaling = 1.e5,
     ↪  x4_scaling = 1.e-1;
    double x_init[4] = {this->kerns.getKb1() * x1_scaling,
     ↪  this->kerns.getKb2() * x2_scaling,
                        this->kerns.getKc1() * x3_scaling,
                         ↪  this->kerns.getKc2() * x4_scaling};
    gsl_vector_view x = gsl_vector_view_array(x_init, p);
    double chisq, chisq0;
    int status, info;

    const double xtol = 1e-8;
    const double gtol = 1e-8;
    const double ftol = 1.e-4;

    /* define the function to be minimized */
    fdf.f = gatewayCostMean;
    fdf.df = NULL;      /* set to NULL for finite-difference Jacobian */
    fdf.fvv = NULL;     /* not using geodesic acceleration */
    fdf.n = n;
    fdf.p = p;
    fdf.params = d;

    /* allocate workspace with default parameters */
    w = gsl_multifit_nlinear_alloc(T, &fdf_params, n, p);

    /* initialize solver with starting point and weights */
    gsl_multifit_nlinear_init(&x.vector, &fdf, w);

    /* compute initial cost function */
    f = gsl_multifit_nlinear_residual(w);
    gsl_blas_ddot(f, f, &chisq0);

    /* solve the system with a maximum of 200 iterations */
    status = gsl_multifit_nlinear_driver(200, xtol, gtol, ftol,
                                         paramEstimationCallbackMean,
                                          ↪  NULL, &info, w);

    /* compute covariance of best fit parameters */
    J = gsl_multifit_nlinear_jac(w);
    gsl_multifit_nlinear_covar(J, 0.0, covar);

    /* compute final cost */
    gsl_blas_ddot(f, f, &chisq);

#define FIT(i) gsl_vector_get(w->x, i)
#define ERR(i) sqrt(gsl_matrix_get(covar,i,i))

    time_t rawtime;
    struct tm *timeinfo;
    char buffer[80];
    time(&rawtime);
    timeinfo = localtime(&rawtime);
    strftime(buffer, sizeof(buffer), "%d-%m-%Y-%I:%M:%S", timeinfo);
    std::string str(buffer);
    std::string outputFilename =
    ↪  "../results/parameterEstimation/means_all_dists" + str + ".dat";
    std::ofstream outfile;
    outfile.open(outputFilename);

    fprintf(stderr, "summary from method '%s/%s'\n",
            gsl_multifit_nlinear_name(w),
            gsl_multifit_nlinear_trs_name(w));
    fprintf(stderr, "number of iterations: %zu\n",
            gsl_multifit_nlinear_niter(w));
    fprintf(stderr, "function evaluations: %zu\n", fdf.nevalf);
    fprintf(stderr, "Jacobian evaluations: %zu\n", fdf.nevaldf);
    fprintf(stderr, "reason for stopping: %s\n",
            (info == 1) ? "small step size" : "small gradient");
    fprintf(stderr, "initial |f(x)| = %f\n", sqrt(chisq0));
    fprintf(stderr, "final   |f(x)| = %f\n", sqrt(chisq));

    {
        double dof = n - p;
        double c = GSL_MAX_DBL(1, sqrt(chisq / dof));

        fprintf(stderr, "chisq/dof = %g\n", chisq / dof);

        fprintf(stderr, "kb1      = %.3g +/- %.3g\n", FIT(0), c * ERR(0));
        fprintf(stderr, "kb2      = %.3g +/- %.3g\n", FIT(1), c * ERR(1));
        fprintf(stderr, "kc1      = %.3g +/- %.3g\n", FIT(2), c * ERR(2));
        fprintf(stderr, "kc2      = %.3g +/- %.3g\n", FIT(3), c * ERR(3));

        outfile << "#kb1,kb2,kc1,kc2,kb10,kb20,kc10,kc20,chisq/dof,initia⌋
        ↪  l,final,iter\n";
        outfile << FIT(0) / x1_scaling << "," << FIT(1) / x2_scaling
```

```
                << "," << FIT(2) / x3_scaling << "," << FIT(3) / x4_scaling
                << "," << x_init[0] / x1_scaling << "," << x_init[1] / x2_scaling
                << "," << x_init[2] / x3_scaling << "," << x_init[3] / x4_scaling
                << "," << chisq / dof
                << "," << sqrt(chisq0) << "," << sqrt(chisq)
                << "," << gsl_multifit_nlinear_niter(w) << "\n";
        outfile << c * ERR(0) / x1_scaling << "," << c * ERR(1) / x2_scaling
                << "," << c * ERR(2) / x3_scaling << "," << c * ERR(3) / x4_scaling << "\n";
        outfile << fdf_params.h_df << "\n";
    }
    outfile.close();
    fprintf(stderr, "status = %s\n", gsl_strerror(status));

    gsl_multifit_nlinear_free(w);
    gsl_matrix_free(covar);
    return 0;
}

///* Fletcher-Reeves constrained optimization (parameter estimation) */
//double PBModel::fletcherReevesCostFunction(const gsl_vector *v) {
//    realtype kb1 = gsl_vector_get(v, 0);
//    realtype kb2 = gsl_vector_get(v, 1);
//    realtype kc1 = gsl_vector_get(v, 2);
//    realtype kc2 = gsl_vector_get(v, 3);
//    this->kerns.setNewKs(kb1, kb2, kc1, kc2);
//    this->solvePBE();
//    double result = 0;
//    size_t i = 0, j = 0;
//    for (i = 0; i < this->M; i++){
//        for (j = 0; j < this->N; j++){
//            result += pow(this->getResidualij(i, j), 2);
//        }
//    }
//    return result;
//}
//void PBModel::fletcherReevesParamEstimation(){
//    size_t iter = 0;
//    int status;
//
//    const gsl_multimin_fdfminimizer_type *T;
//    gsl_multimin_fdfminimizer *s;
//
//    PBModel *m = this;
```

```
//
//    gsl_vector *x;
//    gsl_multimin_function_fdf func;
//    func.n = 4;
//    func.f = fletcherReevesGatewayCost;
//    func.df = NULL;
//    func.fdf = NULL;
//    func.params = m;
//
//    /* Starting point */
//    x = gsl_vector_alloc(4);
//    double x_init[4] = { this->kerns.getKb1(), this->kerns.getKb2(),
//                         this->kerns.getKc1(), this->kerns.getKc2() };
//    gsl_vector_set(x, 0, this->kerns.getKb1());
//    gsl_vector_set(x, 1, this->kerns.getKb2());
//    gsl_vector_set(x, 2, this->kerns.getKc1());
//    gsl_vector_set(x, 3, this->kerns.getKc2());
//
//    T = gsl_multimin_fdfminimizer_conjugate_fr;
//    s = gsl_multimin_fdfminimizer_alloc (T, 4);
//}

/* Getter methods */
gsl_matrix *PBModel::getFv() const {
    return fv;
}

gsl_vector *PBModel::getR() const {
    return r;
}

gsl_vector *PBModel::getT() const {
    return t;
}

size_t PBModel::getM() const {
    return M;
}

size_t PBModel::getN() const {
    return N;
}

const Grid &PBModel::getGrid() const {
    return grid;
}

const Kernels &PBModel::getKerns() const {
    return kerns;
}

const SystemProperties &PBModel::getSysProps() const {
    return sysProps;
}

const Fluid &PBModel::getCont() const {
    return cont;
}

const Fluid &PBModel::getDisp() const {
    return disp;
}


/* Printer methods */
void PBModel::printExperimentalDistribution(){
    size_t i, j;
    std::cout << "The droplet size density distribution:" << std::endl;
    for (i=0;i<M;i++){
        for(j=0;j<N;j++){
            std::cout << std::setw(8) << std::setprecision(3) <<
             ↪  gsl_matrix_get(fv, i, j) << "\t";
        }
        std::cout << std::endl;
    }
}

void PBModel::printSizeClasses() {
    size_t i = 0;
    std::cout << "Measured size classes: " << std::endl;
    for (i = 0; i < this->N; i++){
        std::cout << std::setw(10) << std::setprecision(7) <<
         ↪  gsl_vector_get(this->r, i);
    }
    std::cout << std::endl;
}

void PBModel::printCurrentPsi(){
    realtype *data = NV_DATA_S(this->NPsi);
    size_t i = 0;
    std::cout << "Psi for the current time iteration is: " << std::endl;
    for (i = 0; i < grid.getN(); i++){
        std::cout << std::setw(8) << std::setprecision(2) << data[i];
    }
    std::cout << std::endl;
}
```

```

void PBModel::printPsi(){
    size_t i = 0, j = 0;
    std::cout << "Nondimensionalized droplet size density distribution:"
     ↪  << std::endl;
    for (i = 0; i < this->M; i++){
        for (j = 0; j < grid.getN(); j++){
            std::cout << std::setw(12) << std::setprecision(3) <<
             ↪  gsl_matrix_get(psi, i, j);
        }
        std::cout << std::endl;
    }
}

void PBModel::printFvSimulated(){
    size_t i = 0, j = 0;
    std::cout << "Droplet size density distribution fvSim(r, t):" <<
     ↪  std::endl;
    for (i = 0; i < this->M; i++){
        for (j = 0; j < this->N; j++){
            std::cout << std::setw(12) << std::setprecision(3) <<
             ↪  gsl_matrix_get(this->fvSim, i, j);
        }
        std::cout << std::endl;
    }
}

void PBModel::printDimensions(){
    std::cout << "Rows: " << this->M << ", Columns: " << this->N <<
     ↪  std::endl;
}

void PBModel::printTime() {
    size_t i = 0;
    std::cout << "Time of measurement: " << std::endl;
    for (i = 0; i < this->M; i++){
        std::cout << std::setw(5) << std::setprecision(4) <<
         ↪  gsl_vector_get(this->t, i);
    }
    std::cout << std::endl;
}

void PBModel::printTau(){
    size_t i = 0;
    std::cout << "Nondimensionalized time vector: " << std::endl;
    for (i = 0; i < this->M; i++){
        std::cout << std::setw(6) << std::setprecision(2) <<
         ↪  gsl_vector_get(this->tau, i);
```

```cpp
    }
    std::cout << std::endl;
}


/* Exporter methods */
int PBModel::exportFvSimulatedWithExperimental() {
    time_t rawtime;
    struct tm * timeinfo;
    char buffer[80];
    time (&rawtime);
    timeinfo = localtime(&rawtime);
    strftime(buffer,sizeof(buffer),"%d-%m-%Y-%I:%M:%S",timeinfo);
    std::string str(buffer);
    std::string outputFilename = "../results/solutionFiles/pbe-" + str +
     ↪  ".dat";
    std::ofstream outfile;

    size_t i = 0, j = 0;
    outfile.open(outputFilename);
    outfile << "#r,#fv\n";
    for (i = 0; i < this->N; i++) {
        outfile << gsl_vector_get(this->r, i) << ",";
        for (j = 0; j < this->M; j++) {
            outfile << gsl_matrix_get(this->fvSim, j, i) << ",";
        }
        for (j = 0; j < this->M; j++){
            outfile << gsl_matrix_get(this->fv, j, i) << ",";
        }
        outfile << std::endl;
    }
    outfile.close();
    return 0;
}

int PBModel::exportFv(){
    /* Create new matrix and vector to dimensionalize results */
    gsl_matrix *tmpfv = gsl_matrix_alloc(this->M, this->grid.getN());
    gsl_matrix_memcpy(tmpfv, psi);
    gsl_matrix_scale(tmpfv, 1/this->sysProps.getRm());

    gsl_vector *tmpr = gsl_vector_alloc(this->grid.getN());
    gsl_vector_memcpy(tmpr, this->grid.getXi());
    gsl_vector_scale(tmpr, this->sysProps.getRm());

    /* Export to file */
    time_t rawtime;
    struct tm * timeinfo;
    char buffer[80];
    time (&rawtime);
    timeinfo = localtime(&rawtime);
    strftime(buffer,sizeof(buffer),"%d-%m-%Y-%I:%M:%S",timeinfo);
    std::string str(buffer);
    std::string outputFilename = "../results/solutionFiles/pbe-" + str +
     ↪  ".dat";
    std::ofstream outfile;

    size_t i = 0, j = 0;
    outfile.open(outputFilename);
    outfile << "#r,#fv\n";
    for (i = 0; i < this->grid.getN(); i++) {
        outfile << gsl_vector_get(tmpr, i) << ",";
        for (j = 0; j < this->M; j++) {
            outfile << gsl_matrix_get(tmpfv, j, i) << ",";
        }
        outfile << std::endl;
    }
    outfile.close();

    /* Free temporary variables */
    gsl_matrix_free(tmpfv);
    gsl_vector_free(tmpr);
    return 0;
}

int PBModel::exportPsi() {
    time_t rawtime;
    struct tm * timeinfo;
    char buffer[80];
    time (&rawtime);
    timeinfo = localtime(&rawtime);
    strftime(buffer,sizeof(buffer),"%d-%m-%Y-%I:%M:%S",timeinfo);
    std::string str(buffer);
    std::string outputFilename = "../results/solutionFiles/pbe-" + str +
     ↪  ".dat";
    std::ofstream outfile;

    size_t i = 0, j = 0;
    outfile.open(outputFilename);
    outfile << "#xi,#psi\n";
    for (i = 0; i < this->grid.getN(); i++) {
        outfile << gsl_vector_get(this->grid.getXi(), i) << ",";
        for (j = 0; j < this->M; j++) {
            outfile << gsl_matrix_get(this->psi, j, i) << ",";
        }
        outfile << std::endl;
    }
    outfile.close();
    return 0;
}

int PBModel::exportMeans(){
    time_t rawtime;
    struct tm * timeinfo;
    char buffer[80];
    time (&rawtime);
    timeinfo = localtime(&rawtime);
    strftime(buffer,sizeof(buffer),"%d-%m-%Y-%I:%M:%S",timeinfo);
    std::string str(buffer);
    std::string outputFilename = "../results/solutionFiles/means-" + str
     ↪  + ".dat";
    std::ofstream outfile(outputFilename);
    if (!outfile.good()) return 1;

    outfile << "#modeled,#experimental" << std::endl;
    size_t t;
    for (t = 0; t < this->M; t++){
        outfile << this->getModeledMean(t) << "," <<
         ↪  this->getExperimentalMean(t) << std::endl;
    }
    return 0;
}

/* Destructors */
PBModel::~PBModel(){
    gsl_matrix_free(this->fv);
    gsl_matrix_free(this->psi);
    gsl_vector_free(this->r);
    gsl_vector_free(this->t);
    gsl_vector_free(this->tau);
    /* NPsi->data points to a row in psi. psi is freed, so we cannot free
     ↪  NPsi yet.
     * Point NPsi->data to nullptr before freeing, so we don't encounter
↪   memory issues.
     */
    this->releaseCVMemory();
    NV_DATA_S(this->NPsi) = nullptr;
    N_VDestroy_Serial(this->NPsi);
}
```

Listing B.11: PBModel.cpp: C++ class to represent the entire population balance equation

```
//
// Created by Sindre Bakke Øyen on 05.03.2018.
//

#include "SystemProperties.h"

/* Constructors */
SystemProperties::SystemProperties() = default;

SystemProperties::SystemProperties(const SystemProperties &s) :
        Rm(s.getRm()), Vl(s.getVl()), Vm(s.getVm()), P(s.getP()),
         ↪  eps(s.getEps()) {}

SystemProperties::SystemProperties(
        realtype Rm, realtype Vl, realtype P, const Fluid &disp) :
        Rm(Rm), Vl(Vl), P(P)
{
    this->Vm = 4.0/3 * M_PI * SUNRpowerI(this->Rm, 3);
    this->eps = this->P / (disp.getRho() * this->Vl);
}

/* Getter methods */
realtype SystemProperties::getRm() const {
    return Rm;
}

realtype SystemProperties::getVl() const {
    return Vl;
}

realtype SystemProperties::getVm() const {
    return Vm;
}

realtype SystemProperties::getP() const {
    return P;
}

realtype SystemProperties::getEps() const {
    return eps;
}

/* Setter methods */

/* Friend methods */
std::ostream &operator<<(std::ostream &os, const SystemProperties
 ↪  &properties) {
    os << "Rm: " << properties.Rm << " Vl: " << properties.Vl << " Vm: "
    ↪  << properties.Vm
       << " P: " << properties.P << " eps: " << properties.eps;
    return os;
}

/* Destructors */
SystemProperties::~SystemProperties(){}
```

Listing B.12: SystemProperties.cpp: C++ class to characterize the environment

B.2 Header Files

```
//
// Created by Sindre Bakke Øyen on 05.03.2018.
//

#ifndef MASTERPROJECTCPP_FLUID_H
#define MASTERPROJECTCPP_FLUID_H
/* Built-in header files */
#include <iostream>                   /* Used for input/output to console
↪     */
#include <ostream>                    /* Used for overloading print
↪   operator  */

/* External library header files */
#include <sundials/sundials_types.h>

class Fluid {
private:
    realtype rho, sigma, nu;
public:
    /* Constructors */
    Fluid();
    Fluid(const Fluid &f);
    Fluid(realtype rho, realtype sigma, realtype nu);

    /* Getter methods */
    realtype getRho() const;
    realtype getSigma() const;
    realtype getNu() const;

    friend std::ostream &operator<<(std::ostream &os, const Fluid &fluid);

    /* Destructors */
    ~Fluid();
};


#endif //MASTERPROJECTCPP_FLUID_H
```

Listing B.13: Fluid.h: C++ header file for Fluid class

```
//
// Created by Sindre Bakke Øyen on 06.03.2018.
//
```

```

#ifndef MASTERPROJECTCPP_GRID_H
#define MASTERPROJECTCPP_GRID_H
/* Built-in header files */
#include <ostream>                      /* Print to console
↪         */
#include <iomanip>                      /* Manipulate output format
↪         */

/* External library header files */
#include <sundials/sundials_types.h>    /* Datatypes from sundials
↪         */
#include <sundials/sundials_math.h>     /* Math functions, power etc
↪         */
#include <gsl/gsl_vector_double.h>      /* Vectors
↪         */
#include <gsl/gsl_matrix_double.h>      /* Matrices
↪         */
#include <gsl/gsl_linalg.h>             /* Linear algebra
↪         */
#include <gsl/gsl_blas.h>               /* Basic linear algebraic
↪   subprograms   */
#include <gsl/gsl_eigen.h>              /* Eigenvectors and values
↪         */

/* User-defined header files */

class Grid {
private:
    gsl_vector *xi, *w;
    gsl_matrix *D, *xipBB, *xipBC, *xippBC;
    size_t N;
    realtype x0, x1, alpha, beta, mu0;
    /* Variables:
     * xi    :: Quadrature points
     * w     :: Quadrature weights
     * D     :: Lagrange derivative matrix
     * xip's:: Interpolated quadrature points for birth terms
     * N     :: Number of grid points
     * x0    :: Left boundary
     * x1    :: Right boundary
     * alpha:: Chooses Jacobi polynomial
     * beta :: Chooses Jacobi polynomial
     * mu0   :: The integral of the weight function in the domain [-1,1] */
public:
    /* Constructors */
    Grid();
    Grid(const Grid &g);
    Grid(size_t N, realtype x0, realtype x1, realtype alpha, realtype
     ↪  beta, realtype mu0);

    /* Setter methods */
    void coefs(size_t j, realtype *r);
    void setQuadratureRule();   /* Gauss Lobatto rule */
    void remapGrid();           /* Remaps grid to [x0, x1] domain */
    void setLagrangeDerivativeMatrix();
    void setInterpolatedXis();

    /* Getter methods */
    size_t getN() const;
    realtype getX0() const;
    realtype getX1() const;
    realtype getAlpha() const;
    realtype getBeta() const;
    realtype getMu0() const;

    gsl_vector *getXi() const;
    gsl_vector *getW() const;
    gsl_matrix *getD() const;
    gsl_matrix *getXipBB() const;
    gsl_matrix *getXipBC() const;
    gsl_matrix *getXippBC() const;

    friend std::ostream &operator<<(std::ostream &os, const Grid &grid);

    /* Destructors */
    ~Grid();
};

#endif //MASTERPROJECTCPP_GRID_H
```

Listing B.14: Grid.h: C++ header file for Grid class

```
//
// Created by Sindre Bakke Øyen on 07.03.2018.
//

#ifndef MASTERPROJECTCPP_KERNELS_H
#define MASTERPROJECTCPP_KERNELS_H
/* User-defined header files */
#include "Grid.h"
#include "SystemProperties.h"

class Kernels {
```

```
private:
    gsl_matrix *KBB, *KBC, *KDC; /* KDC is symmetric */
    gsl_vector *KDB;

    realtype k1, k2, k3, k4, kb1, kb2, kc1, kc2, tf;
    Grid grid;
public:
    /* Constructors */
    Kernels();
    Kernels(const Kernels &k);
    Kernels(realtype kb1, realtype kb2, realtype kc1, realtype kc2,
     ↪  realtype tf,
            const Grid &grid, const SystemProperties &sysProps,
            const Fluid &cont, const Fluid &disp);

    /* Setter methods */
    void initializeKs(const Fluid &cont, const Fluid &disp, const
     ↪  SystemProperties &s);

    void setTf(realtype tf);

    void setKb1(realtype kb1);
    void setKb2(realtype kb2);
    void setKc1(realtype kc1);
    void setKc2(realtype kc2);

    void setNewK1(realtype kb1);
    void setNewK2(realtype kb2);
    void setNewK3(realtype kc1);
    void setNewK4(realtype kc2);

    void setNewKs(realtype kb1, realtype kb2, realtype kc1, realtype kc2);

    // TODO: (Optional) Avoid double for loops and use elementwise
     ↪  operations
    void setBreakageKernels();
    void setCoalescenceKernels();

    /* Getter methods */
    gsl_matrix *getKBB() const;
    gsl_matrix *getKBC() const;
    gsl_matrix *getKDC() const;
    gsl_vector *getKDB() const;
    realtype getK1() const;
    realtype getK2() const;
    realtype getK3() const;
    realtype getK4() const;
    realtype getKb1() const;
    realtype getKb2() const;
    realtype getKc1() const;
    realtype getKc2() const;
    realtype getTf() const;
    const Grid &getGrid() const;

    /* Relational operators */
    Kernels &operator=(const Kernels &rhs);

    /* Friend methods */
    friend std::ostream& operator<<(std::ostream &os, const Kernels
     ↪  &kernels);

    /* Destructors */
    ~Kernels();
};


#endif //MASTERPROJECTCPP_KERNELS_H
```

Listing B.15: Kernels.h: C++ header file for Kernels class

```
//
// Created by Sindre Bakke Øyen on 18.03.2018.
//

#ifndef MASTERPROJECTCPP_MODEL_H
#define MASTERPROJECTCPP_MODEL_H
/***********************************************************************⌋
 ↪   ****************/
/* Preamble
 ↪                    */
/***********************************************************************⌋
 ↪   ****************/
/* Built-in header files */
#include <cmath>
#include <ostream>
#include <fstream>
#include <sstream>
#include <cstdio>
#include <cstdlib>
#include <cstring>
#include <sys/stat.h>

/* External library header files */
#include <sundials/sundials_math.h>     /* Math functions, power etc
 ↪                         */
```

```
#include <sundials/sundials_types.h>    /* Data types such as realtype
↪                      */
#include <cvode/cvode.h>                /* prototypes for CVODE fcts.,
↪   consts.                 */
#include <nvector/nvector_serial.h>     /* access to serial N_Vector
↪                      */
#include <sunmatrix/sunmatrix_dense.h>  /* access to band SUNMatrix
↪                      */
#include <sunlinsol/sunlinsol_dense.h>  /* access to band SUNLinearSolver
↪                      */
#include <cvode/cvode_direct.h>         /* access to CVDls interface
↪                      */

#include <gsl/gsl_spline.h>             /* Spline interpolation from GSL
↪                      */
#include <gsl/gsl_interp.h>             /* Interpolation header from GSL
↪                      */
#include <gsl/gsl_multifit_nlinear.h>   /* Non-linear multifit regression
↪                      */
#include <gsl/gsl_multimin.h>           /* Multidimensional minimization
↪                      */
#include <gsl/gsl_matrix.h>

/* User-defined header files */
#include "Kernels.h"                    /* Contains kernels, override to
↪   use other kernels   */
#include "Grid.h"                       /* Contains Gaussian quadrature
↪   rule                  */
#include "SystemProperties.h"           /* Contains data from
↪   experimental setup + fluid    */

/* Define constants for program to run */
#define RTOL                RCONST(1.0e-4)              /* Relative
↪   integration tolerance               */
#define ATOL                RCONST(1.0e-8)              /* Absolute
↪   integration tolerance               */
#define PHI                 RCONST(0.7e-2)              /* Phase fraction
↪   of oil in water                 */
#define TRASHROWS           RCONST(2)                   /* Rows in csv
↪   not containing relevant data     */
#define TRASHCOLS           RCONST(9)                   /* Columns in csv
↪   not containing relevant data */
#define TRUNCATETHRESHOLD   RCONST(6)                   /* Truncate 0's
↪   in csv if many consecutive 0's */

/*************************************************************************↩
↪   ***************/
```

```
/* Class declaration                                                    */
/****************************************************************************************/
class PBModel {
private:
    char const *filename;
    size_t M, N;                /* Only used for experimental data */

    /* Experimental data */
    gsl_matrix *fv;             /* size MxN           */
    gsl_vector *r, *t;          /* size N, size M     */

    /* Modeled data */
    realtype tout, tRequested;
    gsl_matrix *psi;            /* size Mxgrid.getN()                         */
    gsl_vector_view psiN;       /* size grid.getN()                           */
    gsl_vector *tau;            /* size M                                     */
    gsl_matrix *fvSim;          /* Holds modeled fv on experimental radial domain (size MxN)*/

    /* Sundials variables for evaluating ODE */
    SUNMatrix A;
    N_Vector NPsi;              /* size grid.getN() */
    SUNLinearSolver LS;
    void *cvode_mem;

    /* Classes to help evaluate model */
    const Grid grid;
    Kernels kerns;
    const SystemProperties sysProps;
    const Fluid cont, disp;
public:
    /* Constructors */
    PBModel();
    PBModel(char const *f, realtype kb1, realtype kb2, realtype kc1, realtype kc2,
            const Grid &g, const SystemProperties &s, const Fluid &cont, const Fluid &disp,
            size_t decision);

    /* Helper methods */
    void getRowsAndCols();
    void getDistributions();
    void rescaleInitial();
    int preparePsi();
    int prepareCVMemory(); /* Allocates memory for ODE solver and
     ↪  prepares it for solution */
    int releaseCVMemory();
    int checkFlag(void *flagvalue, const char *funcname, int opt);
    bool checkMassBalance();


    /* Solver methods */
    int getRHS(N_Vector y, N_Vector ydot);
    static int interpolatePsi(const gsl_vector *x, const gsl_vector *y,
     ↪  const gsl_matrix *xx, gsl_matrix *yy);
    static int interpolateFv(const gsl_vector *x, const gsl_vector *y,
     ↪  const gsl_vector *xx, gsl_vector *yy);
    int timeIterate();
    int solvePBE();
    realtype getResidualij(size_t i, size_t j);
    double getModeledMean(size_t t);
    double getExperimentalMean(size_t t);
    realtype getResidualMean(size_t t);
    double getWeightedResidual(size_t i, size_t j, double m, double s);
    double getWeight(double x, double m, double s);


    /* Non-linear least squares parameter estimation */
    int costFunctionSSE(const gsl_vector *x, gsl_vector *f);
    int costFunctionMean(const gsl_vector *x, gsl_vector *f);

    int paramesterEstimationSSE();
    int parameterEstimationMean();


//    /* Fletcher-Reeves constrained optimization (parameter estimation)
 ↪  */
//    double fletcherReevesCostFunction(const gsl_vector *v);
//    void fletcherReevesParamEstimation();


    /* Setter methods */


    /* Getter methods */
    gsl_matrix *getFv() const;
    gsl_vector *getR() const;
    gsl_vector *getT() const;
    size_t getM() const;
    size_t getN() const;

    const Grid &getGrid() const;
    const Kernels &getKerns() const;
    const SystemProperties &getSysProps() const;
    const Fluid &getCont() const;
    const Fluid &getDisp() const;


    /* Print methods */
    void printExperimentalDistribution();
    void printSizeClasses();
    void printCurrentPsi();
    void printPsi();
    void printFvSimulated();
    void printTime();
    void printTau();
    void printDimensions();


    /* Exporter methods */
    int exportFvSimulatedWithExperimental();
    int exportFv();
    int exportPsi();
    int exportMeans();


    inline static bool fileExists(const std::string &fileName){
        struct stat buf;
        return (stat(fileName.c_str(), &buf) != -1);
    }

    /* Destructors */
    ~PBModel();
};
/* CVode trick */
inline int dydt(realtype t, N_Vector y, N_Vector ydot, void *user_data){
    PBModel *obj = static_cast<PBModel *> (user_data);
    int err = obj->getRHS(y, ydot);
    return err;
}

/* Levenberg-Marquardt residuals trick */
inline int gatewayCostSSE(const gsl_vector *x, void *data, gsl_vector *f){
    PBModel *obj = static_cast<PBModel *> (data);
    int err = obj->costFunctionSSE(x, f);
    return err;
}

/* Levenberg-Marquardt mean trick */
```

```
inline int gatewayCostMean(const gsl_vector *x, void *data, gsl_vector
 ↪  *f){
    PBModel *obj = static_cast<PBModel *> (data);
    int err = obj->costFunctionMean(x, f);
    return err;
}

inline void paramEstimationCallbackSSE(const size_t iter, void *params,
                                       const
                                        ↪  gsl_multifit_nlinear_workspace
                                        ↪  *w){
    gsl_vector *f = gsl_multifit_nlinear_residual(w);
    gsl_vector *x = gsl_multifit_nlinear_position(w);
    gsl_matrix *J = gsl_multifit_nlinear_jac(w);
    double rcond;

    /* compute reciprocal condition number of J(x) */
    gsl_multifit_nlinear_rcond(&rcond, w);

    fprintf(stderr, "iter %2zu: kb1 = %.10g, kb2 = %.10g, kc1 = %.10g,
     ↪  kc2, = %.10g,"
                    " cond(J) = %8.4f, |f(x)| = %.4f\n",
            iter,
            gsl_vector_get(x, 0),
            gsl_vector_get(x, 1),
            gsl_vector_get(x, 2),
            gsl_vector_get(x, 3),
            1.0 / rcond,
            gsl_blas_dnrm2(f));
}

inline void paramEstimationCallbackMean(const size_t iter, void *params,
                                       const
                                        ↪  gsl_multifit_nlinear_workspace
                                        ↪  *w){
    gsl_vector *f = gsl_multifit_nlinear_residual(w);
    gsl_vector *x = gsl_multifit_nlinear_position(w);
    gsl_matrix *J = gsl_multifit_nlinear_jac(w);
    double rcond;

    /* compute reciprocal condition number of J(x) */
    gsl_multifit_nlinear_rcond(&rcond, w);

    fprintf(stderr, "iter %2zu: kb1 = %.10g, kb2 = %.10g, kc1 = %.10g,
     ↪  kc2, = %.10g,"
                    " cond(J) = %8.4f, |f(x)| = %.4f\n",
            iter,
            gsl_vector_get(x, 0),
                gsl_vector_get(x, 1),
                gsl_vector_get(x, 2),
                gsl_vector_get(x, 3),
                1.0 / rcond,
                gsl_blas_dnrm2(f));
}
///* Fletcher-Reeves trick */
//inline double fletcherReevesGatewayCost(const gsl_vector *v, void
 ↪  *params){
//    PBModel *obj = static_cast<PBModel *> (params);
//    double err = obj->fletcherReevesCostFunction(v);
//    return err;
//}
#endif //MASTERPROJECTCPP_MODEL_H
```

Listing B.16: PBModel.h: C++ header file for PBModel class

```
//
// Created by Sindre Bakke Øyen on 05.03.2018.
//

#ifndef MASTERPROJECTCPP_SYSTEMPROPERTIES_H
#define MASTERPROJECTCPP_SYSTEMPROPERTIES_H
/* Built-in header files */
#include <cmath>
#include <ostream>

/* External library header files */
#include <sundials/sundials_math.h>     /* Math functions, power etc
 ↪          */

/* User-defined header files */
#include "Fluid.h"

class SystemProperties {
private:
    realtype Rm, Vl, Vm, P, eps;
public:
    /* Constructors */
    SystemProperties();
    SystemProperties(const SystemProperties &s);
    SystemProperties(
            realtype Rm, realtype Vl, realtype P, const Fluid &disp);

    /* Getter methods */
    realtype getRm() const;
    realtype getVl() const;
    realtype getVm() const;
    realtype getP() const;
    realtype getEps() const;

    friend std::ostream &operator<<(std::ostream &os, const
    ↪  SystemProperties &properties);

    /* Destructors */
    ~SystemProperties();
};


#endif //MASTERPROJECTCPP_SYSTEMPROPERTIES_H
```

Listing B.17: SystemProperties.h: C++ header file for SystemProperties class

Appendix C

MATLAB Program

C.1 MATLAB Source Files

```
% Title: Set Distribution
% Author: Sindre Bakke Oyen
% Date (started): 07.06.2017
% Description: Sets a log normal distribution with mean mu
%              and standard deviation sigma. The program plots the
%              distribution and writes it to a tab separated textfile.
%% Set initials and create normal distribution
mean = 30e-6;
st = 10e-6;
var = st^2;
mu = log(mean/sqrt(1+var/mean^2));
sigma = sqrt(log(1+var/mean^2));
x = 0:1e-6:10e-6;
x = [x, 10.1e-6:0.1e-6:80e-6];
x = [x, 81e-6:5e-6:500e-6];
f = 1./(x*sigma*sqrt(2*pi)).*exp(-(log(x)-mu).^2/(2*sigma^2));
f(1) = 0;
%% Plot and check conservations
fig = figure();
fontProps.FontName = 'Calibri';
fontProps.FontSize = 14;
fontProps.FontWeight = 'bold';
hAxes = axes('Xscale', 'log');
set(hAxes, fontProps);
box(hAxes, 'on');
hold(hAxes, 'on')
title('Log Normal Distribution')
```

```
xlabel('radius, R [m]')
ylabel('Number Density, f [-]')
xlim([1e-6 500e-6])
plot(x, f, 'Color', 'r', 'LineWidth', 2, 'Marker',...
'o', 'MarkerEdgeColor','r', 'MarkerFaceColor', 'none',...
'DisplayName', 'f(r)')
legend('s.t. = 60, mean = 350')
hLegend = legend(hAxes, 'show');
set(hLegend, fontProps, 'Location', 'NorthEastOutside');
editFigureProperties(fig);
%saveas(hFig, 'intial_logNormal2', 'epsc') 52
%% Write to normal distribution with radii to file
result_matrix = [x; f];
fid = fopen('raw_logNormal4.txt', 'w');
fprintf(fid, '%1s %10s \n', 'r', 'f_0');
fprintf(fid, '%1.9f %10.6f \n', result_matrix);
log(mean/sqrt(1+var/mean^2));
fclose(fid);
% END OF PROGRAM
```

Listing C.1: setLogNormal.m: Program that was used to set a log-normal distribution as initial condition

```
function [idx1, idx2, idx3, idx4, opt] =
  ↪ sensitivityAroundOptimum(SSEfile, breakFile, coalFile)

A = importdata(SSEfile);
B = importdata(breakFile);
C = importdata(coalFile);

SSEs = A.data;
pb = B.data;
pc = C.data;

kb1 = pb(:, 1);
kb2 = pb(:, 2);
kc1 = pc(:, 1);
kc2 = pc(:, 2);

n1 = length(kb1);
n2 = length(kb2);
n3 = length(kc1);
n4 = length(kc2);

opt = 1e10;
for i = 1:n1
```

```
    for j = 1:n2
        for q = 1:n3
            for r = 1:n4
                idx = (i-1)*n2*n3 + (j-1)*n3 + q;
                currentVal = SSEs(idx, r);
                if (currentVal < opt)
                    opt = currentVal;
                    idx1 = i;
                    idx2 = j;
                    idx3 = q;
                    idx4 = r;
                end
            end
        end
    end
end

end
```

Listing C.2: sensitivityAroundOptimum.m: Function that was used to find the optimal parameter combinations and the objective function value

```
function [ Vl, rhoc, rhod, sigma, Vmax, P, nu ] = setParams( Rmax, flag )
% Set all parameters needed for the program to function

Vl     = 725e-6;       % Volume of liquid in tank              [m^3]
rhoc   = 1e3;          % Density continuous phase (water)    [kg/m^3]
%rhod   = 0.837e3;     % Density of dispersed phase (oil)    [kg/m^3]
%nu     = 16.88e-3;    % Kinematic viscosity                  [m^2/s]
%sigma = 22e-3;        % Surface tension                      [N/m]
%Rmax   = 120e-6;       % Maximum allowed radius of bubbles   [m]
Vmax   = 4/3*Rmax^3;   % Maximum volume                       [m^3]
%P      = 0.366;       % Power usage                          [W]

switch flag
    case 1
        % We chose crude oil B
        rhod  = 0.837e3;
        nu    = 16.88e-3;
        sigma = 22e-3;
        P     = 0.366;
    case 2
        % We chose crude oil C
        rhod  = 0.911e3;
        nu    = 81.67e-3;
        sigma = 19e-3;
        P     = 0.152;
    otherwise
        ME = MException('MATLAB:IllegalFlag', ...
            ['You have passed in an illegal argument. '...
            'Flag must be either 1 or 2. Received %i.\n'], flag);
        throw(ME);
% Crude B:
%   rhod  = 0.837e3
%   nu    = 16.88e-3
%   sigma = 22e-3
%   P     = 0.366
% Crude C:
%   rhod  = 0.911e3
%   nu    = 81.67e-3
%   sigma = 19e-3
%   P     = 0.152
%
% eps = P / (rhod * Vl)
end
```

Listing C.3: setParams.m: Function that was used to set the values of the environment vector

```
function [ fn, fv, r, t ] = rescaleInitial( f0, phi, flag )
% Title: Rescale Initial
% Author: Sindre Bakke Oyen
% Date (started): 07.06.2017
% Description: This function should rescale an initial distribution
%              and return both the number distribution function
%              and its corresponding volumetric number distribution
%              function.
%
% Output args:
%       fn (array)   :: number distribution function [1/m^3*m]
%       fv (array)   :: volumetric number distribution function [1/m]
%       r  (array)   :: radial discretization
% Input args:
%       f0 (csv)     :: initial distribution with radii
%       phi (scalar) :: volume fraction of dispersed phase
%       flag (bool)  :: whether f0 is a volumetric
%                       or a normal number distribution
%                       flag == 0 means f0 is a fv and
%                       flag == 1 means f0 is a fn
%
% This function should rescale the integral since
% I = integral (fv*dr) from rmin to rmax
% I = integral (v(r)*fn*dr) from rmin to rmax
```

```matlab
% Rescale: fi = phi/I * f0, where i is either n or v
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% We shall utilize the trapezoidal rule:
% integral (f(x)dx) from a to b = h/2*sum(f(x_(k+1))+f(x_k))
% = (b-a)/2N * (f(x_1)+2f(x_2)+2f(x_3)+...+2f(x_N)+f(x_(N+1)))

%% Fetching out data from table
A = importdata(f0);
% Find index of density distribution
r = A.data(1, :) * 1e-6; % r is in micrometers
f = A.data(2:end, :);

r = r(1:80);        % After this f-values are irrelevant
f = f(:, 1:80);     % After this f-values are irrelevant

r = r / 2; % originally in diameters, now in radii

t       = split(A.textdata(3:end, 3));
t       = split(t(:, 2), ':');
hours   = str2double(t(:, 1));
mins    = str2double(t(:, 2));
hours   = hours - hours(1);
mins    = mins - mins(1);
t       = hours * 3600 + mins * 60;
for i   = 1:length(t)-1
    if t(i+1)  == t(i)
        t(i+1) = t(i+1) + 30; %add 25 seconds to make them unique
    end %if
end %for

%% Setting variables and preparing for integrating
rmin = r(1);
rmax = r(end);
N = length(r) - 1;

[rows, cols] = size(f);
fv = zeros(rows, cols);
fn = zeros(rows, cols);
switch flag
    case 0
    %% The initial distribution was a fv
    for j=1:rows
        I = 0;
        for i = 1:N
            I = I + ( r(i+1) - r(i) ) * ( f(j, i+1) + f(j, i));
        end %for i
```

```
            I = I / 2;
%             trapz(r, f(j, :))
            fv(j, :) = phi / I * f(j,:);
            for i=1:cols
                if r(i) == 0
                    fn(j, i) = 0; % There are no particles of radius = 0
                else
                fn(j, i) = fv(j, i)/(4/3*pi*r(i)^3);
                end %if
            end %for i
        end %for j
        case 1
            %% Initial distribution was fn
            % The integrand is v(r)*fn
            v = 4/3*pi*r.^3;
            integrand = v.*f;
            I = integrand(1) + integrand(end);
            for i=2:N
                I = I + 2*integrand(i);
            end %for
            I = I * (rmax - rmin) / (2 * N);

            fn = phi / I * f;
            fv = fn*4/3*pi.*r.^3;

        otherwise
            %% The received f0 was neither fv nor fn
            ME = MException('MATLAB:IllegalFlag',...
                ['You have passed in an illegal argument. '...
                'Flag must be either 0 or 1. Received %i.\n'], flag);
            throw(ME);

end % switch
end % function
```

Listing C.4: rescaleInitial.m: Function that was used to rescale the experimental measurements to the phase fraction

```
function [ kBB, kDB, kBC, kDC ] = evalKernels(k2, k4, xis, flag)
% Title: evalKernels
% Author: Sindre Bakke Oyen
% Date (started): 14.06.2017
% Description: This function evaluates all breakage and coalescence
 ↪  kernels
%              associated with them. The kernels for birth and death are
%              evaluated individually as they are over different domains.
```

```
%
% Output args:
%       kBB  (2D array):: Birth breakage rate of breakage
%       kDB  (1D array):: Death breakage rate of breakage
%       kBC  (2D array):: Birth coalescence rate of aggregation
%       kDC  (2D array):: Death coalescence rate of aggregation
%
% Input args:
%       kg2  (scalar)  :: Parameter for brekage frequency
%       k4   (scalar)  :: Parameter for coalescence exponent
%       flag (scalar)  :: contains information about which algorithm to
 ↪  run
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%|
 ↪  %%
xi = xis.xi;
xipBB = xis.xipBB;
xipBC = xis.xipBC;
xippBC = xis.xippBC;
xipDC = xis.xipDC;
N = length(xi);
switch flag
    case 0 % Double for loop
        % Preallocate space for kernels
        kBB = zeros(N, N);
        kDB = zeros(N, 1);
        kBC = zeros(N, N);
        kDC = zeros(N, N);
        for row=2:N
            % DB
            kDB(row) = 1/xi(row)^(2/3) * exp(-k2/xi(row)^(5/3));
            for col=1:N
                % BB
                kBB(row, col) = 2 * 1/xipBB(row, col)^(2/3) ...
                    *exp(-k2/xipBB(row, col)^(5/3)) ...
                    *2.4/xipBB(row, col)^3 ...
                    *exp(-4.5 * (2*xi(row)^3-xipBB(row, col)^3)^2 ...
                    /xipBB(row, col)^6)...
                    *3*xi(row)^2;

                % BC
                kBC(row, col) = (xipBC(row,col)+xippBC(row,col))^2 ...
                    *(xipBC(row,col)^(2/3)+xippBC(row,col)^(2/3))^(1/2)
                     ↪  ...
                    *exp(-k4*(1/xipBC(row,col) + 1/xippBC(row,col)) ...
                    ^(-5/6));

                % DC
                kDC(row,col) = (xipDC(col)+xi(row))^2 ...
```

```
                    *(xipDC(col)^(2/3)+xi(row)^(2/3))^(1/2) ...
                    *exp(-k4*(1/xipDC(col) + 1/xi(row))^(-5/6));
            end %col
        end %row

    case 1 % Single for loop
        % Preallocate space for kernels
        kBB = zeros(N, N);
        kBC = zeros(N, N);
        kDC = zeros(N, N);
        for row = 2:N
        kBB(row, :) = 2 * 1./xipBB(row,:).^(2/3) ...
                .*exp(-k2./xipBB(row,:).^(5/3)) ...
                .*2.4./xipBB(row,:).^3 ...
                .*exp(-4.5*(2.*xi(row)^3-xipBB(row,:).^3).^2 ...
                ./xipBB(row,:).^6) ...
                .*3.*xi(row)^2;

        kBC(row,:) = (xipBC(row, :)+xippBC(row, :)).^2 ...
                .*(xipBC(row, :).^(2/3)+xippBC(row, :).^(2/3)).^(1/2) ...
                .*exp(-k4*(1./xipBC(row, :) + 1./xippBC(row, :)).^(-5/6));

        kDC(row, :) = (xipDC+xi(row)).^2 ...
                .*(xipDC.^(2/3)+xi(row).^(2/3)).^(1/2) ...
                .*exp(-k4*(1./xipDC + 1/xi(row)).^(-5/6));
        end %row
        kDB = 1./xi.^(2/3) .* exp(-k2./xi.^(5/3));

    case 2 % No for loops
        xir = repmat(xi, 1, N);
        xiprDC = repmat(xipDC, 1, N)';

        kBB = 2 * 1./xipBB.^(2/3) ...
                .*exp(-k2./xipBB.^(5/3)) ...
                .*2.4./xipBB.^3 ...
                .*exp(-4.5*(2.*xir.^3-xipBB.^3).^2./xipBB.^6) ...
                .*3.*xir.^2;

        kBC = (xipBC+xippBC).^2 ...
                .*(xipBC.^(2/3)+xippBC.^(2/3)).^(1/2) ...
                .*exp(-k4*(1./xipBC + 1./xippBC).^(-5/6));

        kDC = (xiprDC+xir).^2 ...
                .*(xiprDC.^(2/3)+xir.^(2/3)).^(1/2) ...
                .*exp(-k4*(1./xiprDC + 1./xir).^(-5/6));
        kDB = 1./xi.^(2/3) .* exp(-k2./xi.^(5/3));

    otherwise
```

```
        ME = MException('MATLAB:IllegalFlag', ...
            ['The flag received is illegal. '...
            'Supported: 0, 1 or 2. Received: %i'], flag);
        throw(ME);

end %function
```

Listing C.5: evalKernels.m: Function that was used to set the kernels

```
function RHS = evalSource( tau, psi, kern, const, xis, flag )
% Title: Evaluate Source
% Author: Sindre Bakke Oyen
% Date (started): 20.06.2017
% Description: This function should evaluate the right hand side of the
%              nondimensionalized PBE. It will evaluate it for each radial
%              discretization, meaning it will return an array of
%              length = number of discretization points.
%
% Output args:
%       RHS    (array)  :: source of bubbles of radius xi
% Input args:
%       tau    (array)  :: dimensionless time
%       psi    (array)  :: dimensionless volumetric density distribution
%       kern   (struct) :: contains all birth and death kernels
%       const  (struct) :: contains all constants, k1, k2, k3, k5 and phi
%       flag   (scalar) :: contains information about which algorithm to
 ↪  run
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
 ↪  %%
xi = xis.xi;
xipBB = xis.xipBB;
xipBC = xis.xipBC;
xippBC = xis.xippBC;
xipDC = xis.xipDC;
N = length(xi);

% Fetch all kernels
kBB = kern.BB.k;
kDB = kern.DB.k;
kBC = kern.BC.k;
kDC = kern.DC.k;

% Fetch all constants
k1 = const.k1;
k3 = const.k3;
w  = const.w;
```

```

% Interpolate onto domains in terms of volume
psipBB   = pchip(xi, psi, xipBB);
psipBC  = pchip(xi, psi, xipBC);
psippBC = pchip(xi, psi, xippBC);
psipDC  = pchip(xi, psi, xipDC);

switch flag
    case 0 % Double for loop
        % Preallocate space for integrands
        IBB = zeros(N, N);
        IBC = zeros(N, N);
        IDC = zeros(N, N);

        % Preallocate space for the source terms
        BB = zeros(N, 1);
        DB = zeros(N, 1);
        BC = zeros(N, 1);
        DC = zeros(N, 1);
        for row=2:N % loop on xi
            for col=1:N % loop on xi' and xi''
                % BB
                IBB(row, col) = kBB(row,col) ...
                    *psipBB(row, col)/xipBB(row,col)^3 ...
                    *(1-xi(row));
                % BC
                IBC(row, col) = kBC(row, col) ...
                    *psipBC(row, col)/xipBC(row,col)^3 ...
                    *psippBC(row,col)/xippBC(row,col)^3 ...
                    *xi(row)^2/xippBC(row, col)^2 ...
                    *xi(row)*2^(-1/3);
                % DC
                IDC(row, col) = kDC(row, col) * psipDC(col)/xipDC(col)^3;
            end %col
            BB(row) = k1 * xi(row)^3 * IBB(row, :)*w;
            DB(row) = k1*kDB(row)*psi(row);
            BC(row) = k3*xi(row)^3 * IBC(row, :)*w;
            DC(row) = k3*psi(row) * IDC(row, :)*w;
        end %row

        B = BB - DB; % Net breakage
        C = BC - DC; % Net coalescence

        RHS = B + C;

    case 1 % Single for loop
        % Preallocate space for integrands
        IBB = zeros(N, N);
        IBC = zeros(N, N);
        IDC = zeros(N, N);
        for row=2:N
            % BB
            IBB(row, :) = kBB(row,:) ...
                .*psipBB(row, :)./xipBB(row,:).^3 ...
                 *(1-xi(row));
            % BC
            IBC(row, :) = kBC(row, :) ...
                .*psipBC(row, :)./xipBC(row, :).^3 ...
                .*psippBC(row, :)./xippBC(row, :).^3 ...
                 *xi(row)^2./xippBC(row, :).^2 ...
                 *xi(row)*2^(-1/3);
            % DC
            IDC(row, :) = kDC(row, :) .* (psipDC./xipDC.^3)';
        end %row
        BB = k1 * xi.^3. .* (IBB*w);
        DB = k1*kDB.*psi;
        BC = k3*xi.^3 .* (IBC*w);
        DC = k3*psi .* (IDC*w);

        B = BB - DB; % Net breakage
        C = BC - DC; % Net coalescence

        RHS = B + C;

    case 2 % No for loops
        xir = repmat(xi, 1, N); %xi without loops must have same
         ↪  dimensions

        IBB = kBB.*psipBB./xipBB.^3.*(1-xir);
        IBC = kBC.* ...
            psipBC./xipBC.^3 ...
            .*psippBC./xippBC.^3 ...
            .*xir.^2./xippBC.^2 ...
            .*xir*2^(-1/3);
        IDC = kDC .* repmat((psipDC./xipDC.^3)', N, 1);
        IDC(1, :) = 0;

        BB = k1 *  xi.^3 .* (IBB*w);
        DB = k1*kDB.*psi;
        BC = k3*xi.^3 .* (IBC*w);
        DC = k3*psi .* (IDC*w);

        B = BB - DB; % Net breakage
        C = BC - DC; % Net coalescence

        RHS = B + C;
        RHS;
    otherwise
        ME = MException('MATLAB:IllegalFlag', ...
            ['The flag received is illegal. '...
            'Supported: 0, 1 or 2. Received: %i'], flag);
        throw(ME);

end %function
```

Listing C.6: evalSource.m: Function that was used to find the right hand side of the population balance equation

```
% Title: Solution of the transient nondimensionalized PBE
% Author: Sindre Bakke Oyen
% Date (started): 13.06.2017
% Description: Main script for solving the transient nondimensionalized
%              PBE. It rescales initial distribution, f*, to fv and sets
%              the collocation points at the roots of Jacobi polynomials.
%              The points are orthogonally collocated in xi, xi' and xi''.
%
% Notation:
%   BB  :: Birth brekage
%   DB  :: Death brekage
%   BC  :: Birth coalescence
%   DC  :: Death coalescence
%   vp  :: Generic variable v prime (v')
%   vpp :: Generic variable v double prime (v'')
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%↩
↪   %%

clc

rng default;

%% Get initial distribution and discretize
phi = 0.7e-2;
f0 = 'Experimental/august/crudeB.csv';
[fn, fv, r, t] = rescaleInitial(f0, phi, 0);
Rmax = 120e-6;

[xi, A, B, w] = Collocation(198,1,1);
xi(1) = 1e-10;
N = length(xi);
alpha = xi;
gamma = xi;

```

```
% BB
xipBB = (1-xi)*gamma'+xi*ones(1, N);

% DC
xipDC  = xi;

%BC
xipBC  = 2^(-1/3)*xi*alpha';
xippBC = xi*(1-alpha.^3/2).^(1/3)';
%% Setting parameters and constants
kb1 = 1.5e-6;        % Model fitted parameter 1, g        [-]
                     % Dynamics, breakage
kb2 = 1e-2;          % Model fitted parameter 2, g        [-]
                     % Steady state settlement, breakage
kc1 = 1.5e-5;        % Model fitted parameter, probability[-]
                     % Dynamics, coalescence
kc2 = 5e2;           % Model fitted parameter, efficiency [-]
                     % Steady state settlement, coalescence
ratio = kb1/kc1;

kb1 = 1e0;
kb2 = 1e3;
kc1 = 1e0;
kc2 = 1e3;

consts.Rmax        = Rmax;
consts.xis.xi      = xi;
consts.xis.xipBB   = xipBB;
consts.xis.xipBC   = xipBC;
consts.xis.xippBC  = xippBC;
consts.xis.xipDC   = xipDC;
consts.fv          = fv;
consts.t           = t;
consts.tf          = t(end);
consts.r           = r;
consts.Rmax        = Rmax;
consts.w           = w;
consts.phi         = phi;

%% What search area to chart?
Niter = 40;     % Number of iterations (Niter x Niter SSE matrix produced)
stepSize = 2; % The parameters charted will be multiplied by this
fprintf('1 : kb1, kb2\n')
fprintf('2 : kb1, kc1\n')
fprintf('3 : kb1, kc2\n')
fprintf('4 : kb2, kc1\n')
fprintf('5 : kb2, kc2\n')
fprintf('6 : kc1, kc2\n')
```

```
flag = input('Which parameters would you explore? ');
steps = zeros(Niter, 1);
steps(end) = 1;
for i = Niter:-1:2
    steps(i-1) = steps(i) / stepSize;
end %for
switch flag
    case 1
        k1_vec = kb1 * steps;
        k2_vec = kb2 * steps;
        p1    = kc1;
        p2    = kc2;
    case 2
        k1_vec = kb1 * steps;
        p1    = kb2;
        k2_vec = kc1 * steps;
        p2    = kc2;
    case 3
        k1_vec = kb1 * steps;
        p1    = kb2;
        p2    = kc1;
        k2_vec = kc2 * steps;
    case 4
        p1    = kb1;
        k1_vec = kb2 * steps;
        k2_vec = kc1 * steps;
        p2    = kc2;
    case 5
        p1    = kb1;
        k1_vec = kb2 * steps;
        p2    = kc1;
        k2_vec = kc2 * steps;
    case 6
        p1    = kb1;
        p2    = kb2;
        k1_vec = kc1 * steps;
        k2_vec = kc2 * steps;
    otherwise
        error('Flag does not match any of the given');
end %switch

%% Solve and chart sensitivity
eSquared = zeros(Niter, Niter);
parfor i = 1:Niter
    k1 = k1_vec(i);
    tmpSquared = zeros(Niter, 1);
    for j = 1:Niter
        k2 = k2_vec(j);
        tmpSquared(j) = getSSE(k1, k2, p1, p2, consts, flag);
    end %for
    eSquared(i, :) = tmpSquared;
end %parfor

createFigure();
ax = axes();
hold(ax, 'on');
set(ax, 'Xscale', 'log');
set(ax, 'Yscale', 'log');
set(ax, 'Zscale', 'log');
surf(k1_vec, k2_vec, eSquared)
xlabel('k_{b,1}')
ylabel('k_{b,2}')
zlabel('SSE')
view(3);
%% Set plot properties
greekeps = char(949); % Greek letter epsilon
greekphi = char(966);     % Greek letter phi

colorMatrix{1} = sprintf('b');
colorMatrix{2} = sprintf('r');
colorMatrix{3} = sprintf('g');
colorMatrix{4} = sprintf('m');

fontProps.FontName = 'Calibri';
fontProps.FontSize = 14;
fontProps.FontWeight = 'bold';

%% Save variables for later plotting
now  = strsplit(char(datetime())); % Cell of date and time
time = strsplit(now{2}, ':');
now  = strcat('Results/Experimental/crudeB/', ...
     now{1}, '-', time{1}, '_', time{2}, '_', time{3});
save(now)

% END PROGRAM
```

Listing C.7: main.m: The main program that solved the population balance equation

```
function [eSquared] = getSSE(k_1, k_2, p1, p2, consts, flag)
% Input flag :: decides what pair of parameters received
%                1 : kb1, kb2
%                2 : kb1, kc1
%                3 : kb1, kc2
%                4 : kb2, kc1
%                5 : kb2, kc2
%                6 : kc1, kc2
Rmax = consts.Rmax;
r    = consts.r;
xi   = consts.xis.xi;
w    = consts.w;
fv   = consts.fv;
t    = consts.t;
tf   = consts.tf;
phi  = consts.phi;
%% Fetch betas and experimental values
switch flag
    case 1 % chosen kb1, kb2
        kb1 = k_1;
        kb2 = k_2;
        kc1 = p1;
        kc2 = p2;
    case 2 % chosen kb1, kc1
        kb1 = k_1;
        kb2 = p1;
        kc1 = k_2;
        kc2 = p2;
    case 3 % chosen kb1, kc2
        kb1 = k_1;
        kb2 = p1;
        kc1 = p2;
        kc2 = k_2;
    case 4 % chosen kb2, kc1
        kb1 = p1;
        kb2 = k_1;
        kc1 = k_2;
        kc2 = p2;
    case 5 % chosen kb2, kc2
        kb1 = p1;
        kb2 = k_1;
        kc1 = p2;
        kc2 = k_2;
    case 6 % chosen kc1, kc2
        kb1 = p1;
        kb2 = p2;
        kc1 = k_1;
        kc2 = k_2;
    otherwise
        error('Flag does not match any of the given\n');
end %switch
fv0 = fv(1, :);

%% Set the parameters and constants needed
```

```
[ Vl, rhoc, rhod, sigma, Vmax, P, ~ ] = setParams( Rmax, 1 ); % Crude B
eps = P / (rhod * Vl);

% Final constants
k1 = tf*kb1*eps^(1/3)/(2^(2/3)*Rmax^(2/3))*sqrt(rhod/rhoc);
k2 = kb2*sigma/(rhod*2^(5/3)*eps^(2/3)*Rmax^(5/3));
k3 = tf/Vmax * Rmax^(7/3)*4*2^(1/3)*kc1*eps^(1/3);
k4 = kc2*Rmax^(5/6)*rhoc^(1/2)*eps^(1/3)/(2*sigma^(1/2));

%% Solve program
% Find kernels
[kern.BB.k, kern.DB.k, kern.BC.k, kern.DC.k] =...
    evalKernels(k2, k4, consts.xis, 2);

% Store some constants needed for the source evaluation
const.k1 = k1;
const.k3 = k3;
const.w = w;

tau  = t / tf;
psi0 = pchip(r/Rmax, fv0*Rmax, xi);

% Setting ODE options
options = odeset();
tic
[~, psi] = ...
    ode15s(@evalSource, tau, psi0, options, kern, const, consts.xis, 2);
t_ode = toc
deviation = abs((psi(end, :)*w - phi) / phi * 100);
if deviation > 5
    fprintf('The mass is not conserved. Phase fraction deviation
     ↪  %4.3f\n', deviation)
end %if

if size(psi) ~= size(fv)
    eSquared = NaN;
else
    fv_modeled = psi / Rmax;
    fv_modeled = pchip(xi*Rmax, fv_modeled, r);
    eSquared   = sum(sum((fv_modeled - fv).^2));
end
end %function
```

Listing C.8: getSSE.m: Function was used to get the sum of squared errors between experimental data and numerical simulations

```matlab
clc

x0 = 1;
x1 = 5;
Np = 5;
xI = linspace(x0, x1, Np);

N = 100;
x = linspace(x0, x1, N);
ell = ones(N, N);

coefs = zeros(Np, Np);
for i = 1:Np
    polynom=1;
    denominator=1;
    for j = 1:Np
        if j~=i
           polynom=conv(polynom,[1 -xI(j)]);
           denominator=denominator*(xI(i)-xI(j));
        end
    end
    coefs(i,:)=polynom/denominator;

end

fig = figure();
ax = gca();
hold(ax, 'on');
legendCell = cell(Np, 1);
for i = 1:Np
    plot(x, polyval(coefs(i, :), x), 'LineWidth', 2)
    legendCell{i} = sprintf('l_%i(x)', i-1);
    hold on;
end
plot(xI, zeros(length(xI), 1), 'o', 'LineWidth', 1.5, ...
    'MarkerSize', 6, 'MarkerEdgeColor', 'k')
plot(xI, ones(length(xI), 1), 'o', 'LineWidth', 1.5, ...
    'MarkerSize', 6, 'MarkerEdgeColor', 'k')
xlabel('x')
ylabel('y')
title('Lagrange Interpolating Polynomials')
legend(legendCell)
editFigureProperties(fig);
```

Listing C.9: lagrangeInterpShowcase.m: Program that was used to showcase Lagrange inteprolating polynomials