University of Nottingham
UK | CHINA | MALAYSIA

Coupling granular size segregation and cyclic-loading-induced crushing effects in a continuum model

Thesis submitted to the University of Nottingham for the degree of **Doctor of Philosophy, September 2023.**

Jiaxin Zhang

14342441

Supervised by

Charles Heron
Barbara Turnbull

Signature _____

Date _____ / _____ / _____

Abstract

The kinetic sieving mechanism (Gray and Thornton, 2005; Gray, 2018) is a well-known continuum method for modelling granular size segregation phenomena, typically solved for steady-state chute flows.

In this thesis, we develop a model to include normal confining pressure dependence, time-dependence and evolving velocity profiles through the grain body. This enabled more sophisticated granular behaviours and feed-backs to be explored - in particular, those relevant to cyclic loading of soils through e.g. wind turbine foundations. An iterative expansion approach is proposed to self-expand the classic bi-disperse or tri-disperse segregation problem towards arbitrarily poly-disperse systems, with minimal change of input parameters and data structures.

Under pair-wise stress partition and segregation relationships, behaviour can be directly linked to size ratios. Simulations show the dependence on particle size distribution alongside the controlling non-dimensional parameters: inter-particle drag $\mathcal{C}$, diffusion rate $\mathcal{D}$ and confining pressure $\mathcal{P}_0$.

The final goal of this project was to couple particle crushing with the validated poly-disperse segregation model. After exploring possible ways of incorporating breakage, solid volume fraction re-distribution was tested in a 10-population problem with clear feed-backs from the prescribed crushing.

Acknowledgements

I would like to express my most sincere and deepest gratitude towards my supervisors Professor Barbara Turnbull and Professor Charles Heron for their continued patience and support through the length of my PhD project. I would also like to thank my colleague William Webb for his tremendous help running the poly-disperse simulations. Lastly, I want to thank my friends and loved ones for understanding and cheering for me during the hardest days and darkest nights.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this Chapter, we briefly explain the motivation behind modelling granular size segregation effects using continuum models as well as provide an overview of the development process of our models. An outline of the project is given at the end of this chapter.

Contents

1.1 The geotechnical motivation

Systems of granular materials, from geophysical phenomena such as dry snow avalanches (Jomelli and Bertran, 2001; Bartelt and McArdell, 2009; Pudasaini and Hutter, 2007; Issler et al., 2018) to industrial applications such as pharmaceutical manufacture (Bae et al., 2018; Muzzio et al., 2002), exhibit segregation effects. Also known as the 'Brazil nut effect' (Rosato et al., 1987; Ottino and Khakhar, 2000), when a mixture of granules is energetically excited, granules differing in size tend to migrate and form highly-concentrated regions of particles sorted by size. As shown in Figure 1.1(a), the largest granules (the Brazil nuts) are lifted to the top of the layer where the smallest (the milk powder) collect at the base. Interestingly, reversed Brazil nut effect have also been predicted and observed where large particles fall through the swarm of small particles under specific settings (Hong et al., 2001; Quinn et al., 2002) as shown in Figure 1.1(b). We are particularly interested in modelling the typical Brazil nut effect.

A number of driving mechanisms can explain this re-arranging effect under various circumstances. For the chute flow scenario, where granular mixture is moving along an inclined slope driven by gravity (e.g. debris flows), kinetic-sieving (Scott and Bridgwater, 1975; Savage and Lun, 1988) appears to dominate. Kinetic-sieving comprises a two-step process of 'granular percolation' and 'squeeze expulsion'. As the granular mixture propagates down-slope, local void spaces are created and eliminated due to the bulk movement. This enables a subset of the grains (usually of smaller size) to percolate through temporary gaps driven by gravity. The closer packing of these smaller particles collecting at the base of the flow then in turn levers other grains upwards through squeeze expulsion.

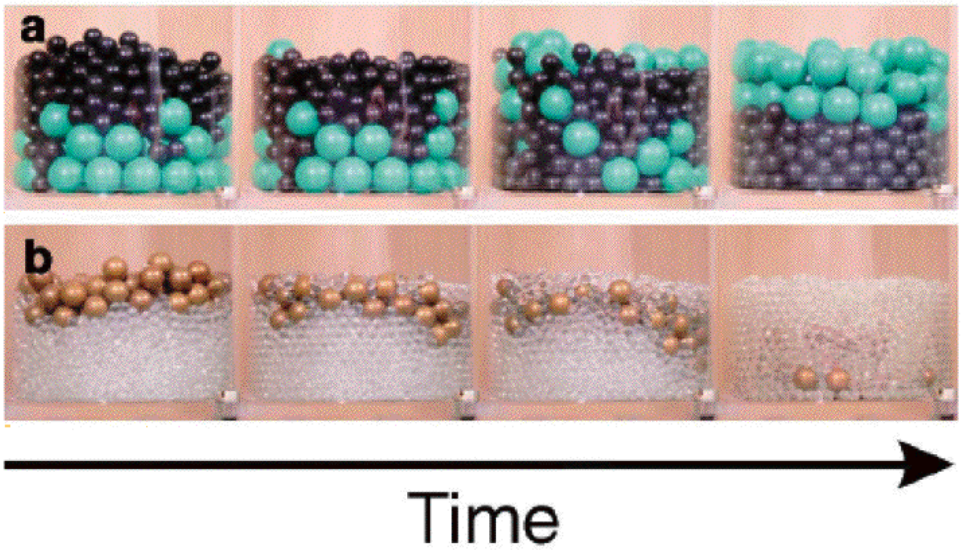


Figure 1.1: (a) Typical Brazil nut effect observed with 8 mm glass beads on top of 15 mm polypropylene; (b) Reverse Brazil nut effect observed with 10 mm bronze spheres on 4 mm glass beads (Breu et al., 2003).

1.2 Granular size segregation

In this thesis we are interested in the behaviour of granular material surrounding structures in the ground, such as pipelines, turbine foundations and underground tunnels. These structures typically exert cyclic loads onto the surrounding soil leading to size segregation which alters the macro behaviour of the soil such as drainage and bearing capacity. Further, the confinement and high loads associated with such infrastructure can lead to the soil granules being crushed into smaller fragments (Mullin, 2000; Pihler-Puzović and Mullin, 2013). Hence, the load characteristics are important if we are to develop an understanding of the interplay between cyclic behaviour and normal loading in the development of size segregated regions.

Although Discrete Element Methods (DEM) are frequently used for more realistic granular simulations due to their ability in specifying meso-scopic particle-to-particle interaction parameters and laws (Brandao et al., 2020a;

Combarros et al., 2014), a continuum model poses far lower computing demand and hence provides much room for extension towards highly polydiperse mixtures. Good qualitative and quantitative agreement with shear box (Golick and Daniels, 2009) and Couette cell (May et al., 2010) experiments show the adaptability of a continuum framework and these features will help pave the way for the future incorporation of particle crushing or breakage mechanics into the continuum model.

1.3 Cyclic loading shear box

To date, a number of granular size-segregation experiments have been carried out for chute flows going down slopes, rotating drums with an avalanching free surface (Johnson et al., 2012; Golick and Daniels, 2009; May et al., 2010) and side-shearing in annular shear cells (Scott and Bridgwater, 1975; van der Vaart et al., 2015). However, measuring and recording data for the granular mixture proves to be difficult especially when the bulk motions are still taking place (Leadbeater et al., 2012). DEM and continuum models are useful methods to provide insight and predictions on what exactly is going on inside the mixture based on macroscopic measurements and hence facilitate the extrapolation of experimental insight to large-scale geotechnical processes.

The specific cases of geo-physical processes that motivate our exploration of combining granular segregation and particle breakage is the cyclic loading environment for soils. For example, the soil underneath railway tracks (as shown in Figure 1.2) or a wind turbine foundation (as shown in Figure 1.3) experience cyclic shearing exerted from above. These excitements of grains can possibly encourage the soil particles to segregate in size. Addi-

tionally, the confining stress acting on the soil can often crush the brittle particles into smaller pieces or dust. We are particularly interested in exploring whether we can capture these phenomena and their interactions via modified version of classic granular segregation models.

Here, we employ a modified continuum modelling approach following (Thornton and Gray, 2008; Gajjar and Gray, 2014). This translates chute flow continuum segregation models to the geometry of a top-driven cyclic shear box (Forterre and Pouliquen, 2008), incorporating normal loads. This is a computationally low-cost and scale-able method of developing our understanding of how cyclic loading and confining pressures influence the dynamics of granular mixture undergoing kinetic-sieving type segregation.

After developing time-dependence in the governing equations the role of the evolving velocity profile through the cyclic shear cell is explored. An evolving velocity profile is proposed that originates from Bagnold-type profiles in published experiments and DEM simulations for shear boxes (Lo et al., 2010; Brewster et al., 2005). The strong variations in velocity gradient through the shear box lead to different segregation behaviours in those regions. Finally, a normal confining pressure acting on the shear cell is incorporated.

With time-dependence and absolute local stresses included, we look to generalise the methodology such that subsequent work can extend to highly poly-dispersed systems with stress-dependent, localised particle breakage.

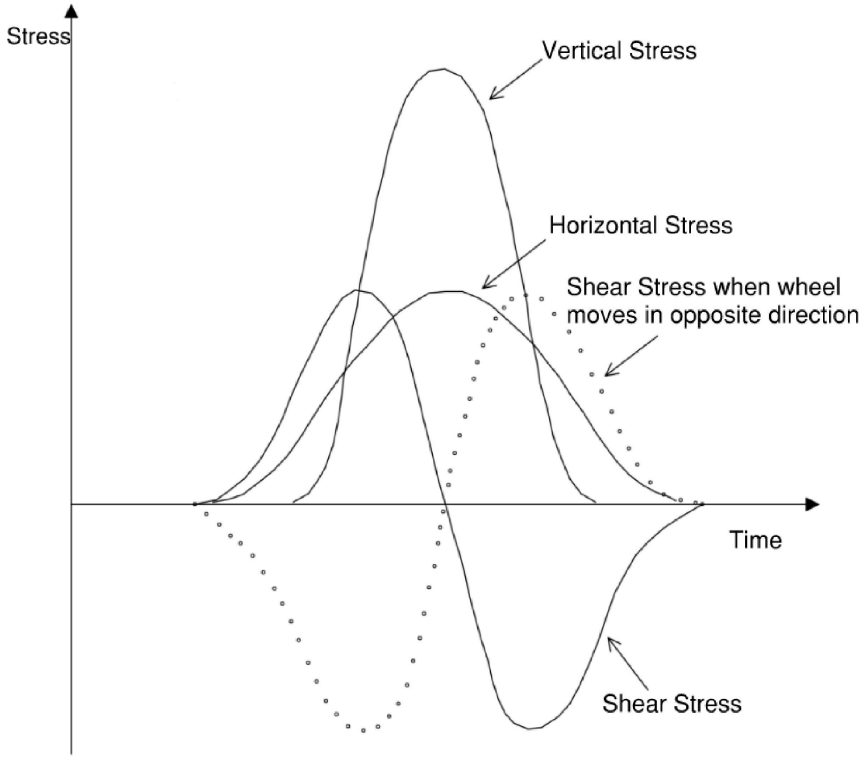


Figure 1.2: Stress conditions under a single moving wheel load (Thevakumar et al., 2021).

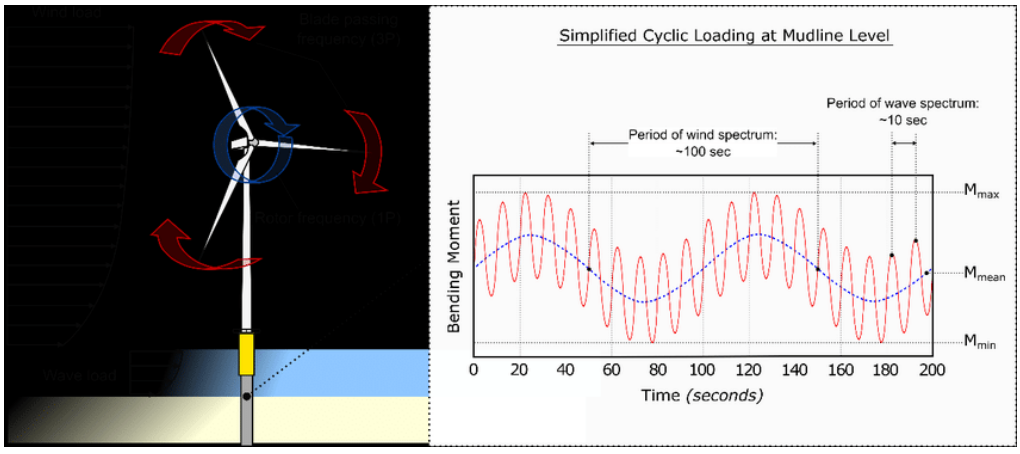


Figure 1.3: Loads acting on a typical offshore wind turbine foundation and typical mud-line moment (Nikitas et al., 2017).

1.4 Outline of this project

Starting from Chapter 2, we go through the theoretical framework and critical assumptions needed to construct classic kinetic-sieving model. Starting from Section 2.3, we start diverting from the classic model in terms of confining pressure dependence and velocity profile. These modifications are introduced with the purpose of capturing more distinct granular characteristics in a cyclic loading environment. In Chapter 4, a new approach of effectively self-assembling granular segregation problems involving arbitrarily many populations of size categories is proposed. This approach has the appealing quality of requiring minimal prescription of parameters as well as being able to adjust and record complex poly-disperse relationships via iterative expansion and symbolic calculation. After validating the poly-disperse results against known classic kinetic-sieving results, a number of test designs are also discussed before showing their simulation results. In Chapter 5, we explore the possibility of incorporating granular breakage, especially corner crushing effects into poly-disperse models.

Chapter 2

Shear Box Model Derivation

This chapter goes through the mathematical structure of the kinetic sieving model. We depart from a classical model designed around chute flow, starting by incorporating a normal, confining pressure. We finish by describing the numerical implementation of the segregation problem, solving by converting the group of partial differential equations into ordinary differential equations for each time-step.

Contents

2.1 Mixture theory framework

Our approach is built on the basis of an existing chute flow model (Thornton and Gray, 2008) which offers a robust foundation thanks to the physically interpretable mathematical framework, compatibility with semi-discrete numerical solvers and its rich potential of further modifications (such as poly-dispersion and time-dependence). This framework adopts a mixture theory (Truesdell and Truesdell, 1984; Thornton et al., 2006) model, with space being perceived to be simultaneously occupied by different phases of granular material. Each phase solely corresponds to the population of a certain type of granular particle categorised by size. According to standard mixture theory, partial quantities can be derived to represent the characteristics of individual phases within the granular mixture. Solid volume fractions ϕ^ν describe the spatial distribution of the particles within this continuum system (Gray and Thornton, 2005).

$$\sum_{\nu \in S} \phi^\nu = 1 \tag{2.1}$$

As shown in equation 2.1, the volume fractions must sum to unity. For example, a bi-disperse mixture of small and large particles satisfies the solid volume fraction relation $\phi^l + \phi^s = 1$, where ϕ^s stands for small particles and ϕ^l stands for large particles. Hence, the partial density ρ^ν of each population ν is defined based on how ϕ^ν partitions solid volume.

$$\rho = \sum_{\nu \in S} \rho^\nu \quad \text{where } \rho^\nu = \phi^\nu \rho^{\nu *}. \tag{2.2}$$

Here, $\rho^{\nu *}$ stands for the intrinsic density of granular species ν, which is in fact the mean solids fraction times the bulk solid density (Gray and Thornton, 2005).

The mixture theory framework provides a reasonable way to scale and partition physical quantities such as density and stress for each size population within a bulk mixture. In the context of granular segregation, we are interested in the relative motion of each size population against the bulk. Therefore, it is crucial to define partial velocity profiles $\mathbf{u}^\nu$ and compare with the bulk velocity profile $\mathbf{u}$. Therefore, it is helpful to assume intrinsic and partial velocity fields to be equal (Gray and Thornton, 2005).

$$\mathbf{u}^\nu = \mathbf{u}^{\nu *}. \tag{2.3}$$

However, stresses are not partitioned directly by volume fraction because, as a granular mixture segregates in size, small particles are more prone to fall through local void spaces and in turn detach from the local force chains, meaning that large particles would need to support a bigger portion of the 'geostatic' pressure inside the granular matrix (Thornton and Gray, 2008; Gajjar and Gray, 2014). An alternative stress partition function f^ν is defined to replace the typical solid volume fraction partition. Hence, the Cauchy stress tensor results in partial stresses

$$\mathbf{T}^\nu = f^\nu \mathbf{T}^{\nu *} \tag{2.4}$$

with vertical z component

$$p^\nu = f^\nu p^{\nu *}$$

leading to a bulk 'geostatic' stress

$$p = \sum_{\nu \in S} p^\nu = \sum_{\nu \in S} f^\nu p^{\nu *}. \tag{2.5}$$

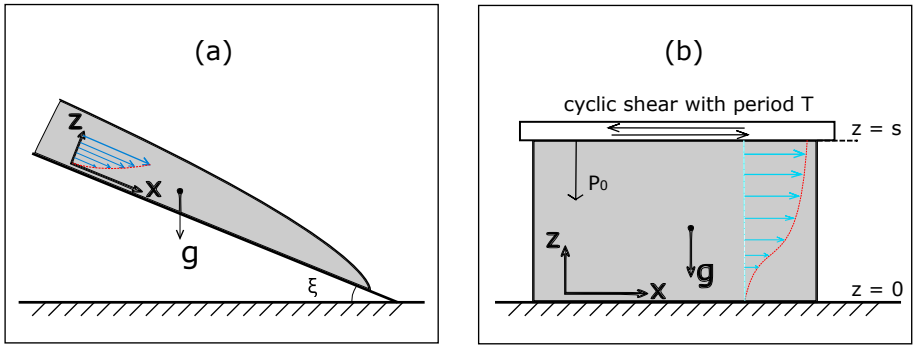


Figure 2.1: Illustrations of chute flow (a) and shear box (b) set-ups. For chute flows, it is convenient to set x axis along the inclined slope and a component of gravity drives the kinetic sieving mechanism in slope-normal z. The shear box's configuration has a level bottom surface and top plate subject to a confining pressure P_0 and driving cyclic shear with period T.

2.2 General kinetic sieving equations for granular segregation

Mass and momentum are conserved for each phase in the granular mixture following

$$\frac{\partial \rho^\nu}{\partial t} + \nabla \cdot (\rho^\nu \mathbf{u}^\nu) = 0 \qquad \forall \nu \in S \tag{2.6}$$

and

$$\frac{\partial}{\partial t}(\rho^\nu \mathbf{u}^\nu) + \nabla \cdot (\rho^\nu \mathbf{u}^\nu \otimes \mathbf{u}^\nu) = \nabla \cdot \mathbf{T}^\nu + \rho^\nu \mathbf{g} + \boldsymbol{\beta}^\nu \tag{2.7}$$

$$\forall \nu \in S$$

respectively. Here $\boldsymbol{\beta}^\nu$ denotes interaction force exerted onto phase ν collectively by all other phases. By Newton's Third Law $\boldsymbol{\beta}^\nu$ should sum to zero for all phases, i.e. $\sum_{\nu \in S} \boldsymbol{\beta}^\nu = 0$. The Cauchy stress tensor is also assumed to take a simple form $\mathbf{T}^\nu = -p^\nu \mathbf{I} + \sigma^\nu$, where the first term represents a spherical inwards pressure and the latter a deviatoric stress.

Here we will contrast the slight differences in modelling conditions between chute flows and the cyclic driven shear cells we aim to explore, shown in Figure 2.1. Since the model was originally devised to simulate steady-

state chute flows, the flow was assumed to be quasi-steady, and the lateral transport and deviatoric stress are assumed to be marginal. After summing over all phases under these two assumptions, the chute flow in z-direction momentum balance equation 2.7 becomes

$$0 + 0 = -\frac{\partial p}{\partial z} + \rho g \cos \xi, \tag{2.8}$$

where ξ denotes the inclination angle of the slope as shown in Figure 2.1. However, in the cyclic shear box set-up, the net granular motion to be horizontal with $\cos \xi = 1$ and

$$0 + 0 = -\frac{\partial p}{\partial z} + \rho g. \tag{2.9}$$

The chute flow has a free surface where the pressure $p = 0$. However, the cyclic loading shear box may have a normal load, such that the boundary condition at the top of the granular medium $p = P_0$ at $z = s$, where s denotes height of the top surface. The new top boundary condition yields a pressure distribution through the bulk

$$p = \rho g(s - z) + P_0, \tag{2.10}$$

in the form of a geo-static stress.

Switching to the cyclic shearing set-up, it makes sense to be mindful about whether the chute flow assumptions should still be employed. For example, deriving equation 2.8 requires quasi-steadiness assumption of the flow, which may not appear suitable for a top-shearing scenario with cyclic reversal taking place. However, we decided to keep this assumption in place mainly for two reasons. First, retaining the time derivative term from momentum balance equation would lead to additional complexity of pressure

formulation compared to a simpler geo-static format (as shown in equation 2.10). Since pressure dependence plays an active role in our modified convection-diffusion equation (as shown in equation 2.14), we found it sensible to start with a mathematically simpler formulation and leave room for improvement for future works. Secondly, we can still interpret retaining quasi-steadiness assumption in cyclic shearing scenario as this: the cyclic reversal timescale is significantly larger than the timescale of segregation, making it appropriate to assume that the mass flux of each granular population caused by segregation does not change significantly in time. Therefore, it makes sense to assume that at least the z-component of $\partial(\rho^\nu \mathbf{u}^\nu)/\partial t$ is sufficiently small, making equation 2.8 still hold.

The resultant geo-static pressure profile will be used throughout this thesis. In future works, a more complex pressure profile with time-dependence seems the right step towards a more general model.

2.3 Normal pressure dependence

By design, the interaction drag $\boldsymbol{\beta}^\nu$ in momentum balance equation originally takes the form (Thornton and Gray, 2008)

$$\boldsymbol{\beta}^\nu = p\nabla f^\nu - \rho^\nu c(\mathbf{u}^\nu - \mathbf{u}) - \rho d\nabla\phi^\nu, \tag{2.11}$$

where c denotes coefficient of inter-particle drag, d stands for coefficient of diffusive remixing and $\mathbf{u}$ stands for the bulk velocity profile. The first term $p\nabla f^\nu$ was designed to cancel out the second half of $\nabla \cdot \mathbf{T}^\nu$ in momentum balance equation to ensure that segregation process is driven by bulk (or

intrinsic) stress $p = p^{\nu *}$ rather than partial stress p^{ν}:

$$\nabla \cdot \mathbf{T}^{\nu} = -\nabla p^{\nu} = -f^{\nu} \nabla p - p \nabla f^{\nu}. \tag{2.12}$$

However, when considering the effects of normal stress the term $-p\nabla f^{\nu}$ must be retained since it carries the normal stress information caused by top plate. Substituting eqn. 2.11 into the momentum balance eqn. 2.7 and once again assuming quasi-steadiness and marginal deviatoric motion yields

$$0 = -f^{\nu} \nabla p - p \nabla f^{\nu} + \rho^{\nu} \mathbf{g} - \rho^{\nu} c(\mathbf{u}^{\nu} - \mathbf{u}) - \rho d \nabla \phi^{\nu}. \tag{2.13}$$

Specific information on how one phase of particle migrates within the mixture can be obtained when taking the z-component of eqn. 2.13 , the density mixture partition eqn. 2.2, the pressure distribution through the bulk eqn. 2.10 and further rearranging them into an expression of the vertical velocity of that phase relative to the bulk

$$w^{\nu} - w = \frac{g}{c}\left(\frac{f^{\nu}}{\phi^{\nu}} - 1\right) - \frac{d}{c}\frac{1}{\phi^{\nu}}\frac{\partial \phi^{\nu}}{\partial z} - \frac{p}{\rho c}\frac{1}{\phi^{\nu}}\frac{\partial f^{\nu}}{\partial \phi^{\nu}}\frac{\partial \phi^{\nu}}{\partial z}. \tag{2.14}$$

where $w^{\nu} - w$ can be seen as the segregation speed of the phase.

The stress partition function f^{ν} is also a modelling choice. For the function to remain physical, pure phase conditions must hold. In other words, if the granular mixture contained only one phase, that sole phase must carry all the load; conversely, if a particle species is not present in the mixture, that

phase can carry no load:

$$f^\nu = 1 \quad \text{when} \quad \phi^\nu = 1$$
$$f^\nu = 0 \quad \text{when} \quad \phi^\nu = 0.$$

For chute flow, Thornton and Gray (2008) proposed a form of f^ν that can satisfy this pure phase condition

$$f^\nu = \phi^\nu + \sum_{\forall \mu} B_{\nu\mu}\phi^\nu\phi^\mu, \tag{2.15}$$

where $B_{\nu\mu}$ is a pairwise segregation parameter describing the segregation effect phase ν receives caused by its pairwise interaction with phase μ.

In order to satisfy the unity-summation condition $\sum f^\nu = 1$, it is required to enforce $B_{\nu\nu} = 0 \quad \forall\nu$ and $B_{\nu\mu} = -B_{\mu\nu} \quad \forall\nu \neq \mu$. Therefore the matrix $\mathbf{B}$ storing all the pairwise segregation relation terms $B_{\nu\mu}$ is anti-symmetric i.e. $\mathbf{B} = -\mathbf{B}^T$; the segregation effect of phase μ on phase ν is equal and opposite to the segregation effect of phase ν on phase μ. The derivative of this stress partition function f^ν with respect to ϕ^ν is

$$\frac{\partial f^\nu}{\partial \phi^\nu} = 1 + \sum_{\forall \mu} B_{\nu\mu}\phi^\mu. \tag{2.16}$$

The choice of stress partition function can strongly influence the behaviour of the simulated flow (Tunuguntla et al., 2017). Choice of stress partition function can be is a method to prioritize which granular behaviour to capture. For example, the pairwise function approach proposed by Thornton and Gray (2008) has the advantage of mathematical simplicity and allows for well-validated parameter extraction via pairwise segregation experiments. In contrast, the quotient form proposed by Marks et al. (2012a) rather focuses on how large particles are being levered upwards due to local

'granular buoyancy'. For simplicity, we adopt the well-established pairwise approach, recognising that different aspects of our system could be brought out with a different choice of partition function.

Substituting the pairwise stress partition equations 2.15 and 2.16 into the segregation speed equation 2.14 for our cyclic shear cell with normal applied stress yields

$$w^\nu - w = \sum_{\forall \mu} \frac{g}{c} B_{\nu\mu}\phi^\mu - \frac{1}{c}\left[d + \frac{p}{\rho}\left(1 + \sum_{\forall \mu} B_{\nu\mu}\phi^\mu\right)\right]\frac{1}{\phi^\nu}\frac{\partial \phi^\nu}{\partial z} \tag{2.17}$$

where p comprises the normal confining pressure alongside the geo-static load, as defined in equation 2.10.

As a comparison, the segregation speed for the kinetic sieving model for a chute flow is

$$w^\nu - w = \sum_{\forall \mu} \frac{g}{c} B_{\nu\mu}\phi^\mu - \frac{d}{c}\frac{\partial}{\partial z}\left(\ln \phi^\nu\right), \tag{2.18}$$

and we see that the effect of introducing the normal applied load is to create an additional term in the segregation speed, acting similarly to the diffusion flux. Note, that since we are restricting our study to size-segregation, neglecting the role of density differences, ρ is considered constant.

2.4 Non-dimensionalisation

The mass and momentum balance equations 2.6 and 2.7 can be non-dimensionalised

$$z = H\hat{z}, \quad x = L\hat{x}, \quad t = T\hat{t} \tag{2.19}$$

where H is a typical height of the granular mixture in shear box, L is a length of the shear box and T is the period of the cyclic shear motion. Therefore, a velocity scale in x-direction can be devised using L/T and H/T can function as a normal velocity scale. After substitution of these scaling choices the mass balance equation 2.6 becomes

$$\frac{1}{T}\frac{\partial \phi^{\nu}}{\partial \hat{t}} + \frac{1}{T}\frac{\partial}{\partial \hat{x}}\left(\phi^{\nu}\hat{u}\right) + \frac{1}{T}\frac{\partial}{\partial \hat{z}}\left[\phi^{\nu}\left(\hat{w}^{\nu} - \hat{w}\right)\right] = 0 \tag{2.20}$$

where $u = \hat{u}L/T$ and $w = \hat{w}H/T$, and the length scales conveniently cancel out. The non-dimensionalised form of relative normal velocity (i.e. segregation speed) $\hat{w}^{\nu} - \hat{w}$. As defined in 2.2 and 2.4, ϕ^{ν} denotes solid volume fraction of a certain class of granular population labelled as ν while f^{ν} is the stress partition function indicating how much stress such class of granular material is sharing among all classes. Note that both ϕ^{ν}, the solid volume fraction granular ν, and f^{ν}, the proportion of stress population ν carries, are dimensionless quantities. Equation 2.14 can be re-arranged

$$\frac{c}{g}\left(w^{\nu} - w\right) = \left(\frac{f^{\nu}}{\phi^{\nu}} - 1\right) - \frac{d}{g}\frac{1}{\phi^{\nu}}\frac{\partial \phi^{\nu}}{\partial z} - \left[\frac{P_0}{\rho g} + (s - z)\right]\frac{1}{\phi^{\nu}}\frac{\partial f^{\nu}}{\partial \phi^{\nu}}\frac{\partial \phi^{\nu}}{\partial z}. \tag{2.21}$$

where, s is the height of granular surface in the z-direction. This seems a sensible choice of typical height scale H, leading to

$$\frac{cs}{Tg}\left(\hat{w}^{\nu} - \hat{w}\right) = \left(\frac{f^{\nu}}{\phi^{\nu}} - 1\right) - \frac{d}{gs}\frac{1}{\phi^{\nu}}\frac{\partial \phi^{\nu}}{\partial \hat{z}} - \left[\frac{P_0}{\rho g s} + (1 - \hat{z})\right]\frac{1}{\phi^{\nu}}\frac{\partial f^{\nu}}{\partial \phi^{\nu}}\frac{\partial \phi^{\nu}}{\partial \hat{z}}. \tag{2.22}$$

where $P_0/\rho g s$ is the normalised 'surface pressure' due to the applied normal confining stress relative to the geostatic load associated with the granular layer. Substituting this non-dimensionalised segregation speed equation 2.22 into the mas balance equation 2.20, the non-dimensionalised seg-

regation equation becomes

$$\frac{\partial \phi^\nu}{\partial \hat{t}} + \frac{\partial}{\partial \hat{x}}\left(\phi^\nu \hat{u}\right) + \frac{Tg}{cs}\frac{\partial}{\partial \hat{z}}\left(f^\nu - \phi^\nu\right) - \frac{Tg}{cs}\frac{\partial}{\partial \hat{z}}\left[\frac{d}{gs}\left(\frac{\partial \phi^\nu}{\partial \hat{z}}\right)\right] - \frac{Tg}{cs}\frac{\partial}{\partial \hat{z}}\left\{\left[\frac{P_0}{\rho g s} + (1-\hat{z})\right]\frac{1}{\phi^\nu}\frac{\partial f^\nu}{\partial \phi^\nu}\frac{\partial \phi^\nu}{\partial \hat{z}}\right\} = 0. \quad (2.23)$$

Taking the pairwise stress partition function f^ν from equation 2.15, the final form of non-dimensionalised segregation equation is

$$\frac{\partial \phi^\nu}{\partial t} + \frac{\partial}{\partial x}(\phi^\nu u) + \frac{\partial}{\partial z}\left\{\frac{1}{\mathcal{C}}\sum_{\forall \mu} B_{\nu\mu}\phi^\nu\phi^\mu - \frac{\mathcal{D}}{\mathcal{C}}\frac{\partial \phi^\nu}{\partial z}\right\} - \frac{1}{\mathcal{C}}\frac{\partial}{\partial z}\left\{\left[\mathcal{P}_0 + (1-z)\right]\left(1 + \sum_{\forall \mu} B_{\nu\mu}\phi^\mu\right)\frac{\partial \phi^\nu}{\partial z}\right\} = 0 \quad (2.24)$$

$$\text{where} \quad \mathcal{C} = \frac{sc}{Tg}, \quad \mathcal{D} = \frac{d}{gs}, \quad \mathcal{P}_0 = \frac{P_0}{\rho g s}, \quad (2.25)$$

the non-dimensional groups controlling inter-particle drag, vertical diffusion and confining pressure respectively.

Compared to Gray and Thornton's segregation equation, although more parameters appear in our equation 2.24, due to the introduction of the confining pressure, the conservative convection-diffusion structure remains the same. As long as the equation can be written as series of flux function derivatives in x, z and t, it should be theoretically solvable by the Kurganov-Tadmor scheme (Kurganov and Tadmor, 2000). We note that our control space of $\mathcal{C}, \mathcal{D}$ and $\mathcal{P}_0$ translates to the parameterisations used by Gray, Thorton, Ancey (Gray and Ancey, 2011) via $S_{\nu\mu} = B_{\nu\mu}Tg/Hc$ and $D_r = Td/Hc$.

2.5 Numerical solver

Having reached a non-dimensionalized segregation equation 2.24 for each size population, we can employ the semi-discrete Kurganov-Tadmor method (Kurganov and Tadmor, 2000) to obtain numerical approximations of all the spatial derivative terms. The Kurganov-Tadmor method is a powerful central scheme that offer great resolution independent of the eigenstructure, especially suitable for solving convection-diffusion equations (Kurganov and Tadmor, 2000). We will use a two-dimensional bi-disperse set-up to demonstrate the workflow of the numerical method.

For a two-dimensional bi-disperse problem, only one of the two population needs to be solved thanks to conservation of mass and therefore solid volume fraction. Therefore, the bi-disperse segregation equation takes form:

$$\begin{aligned} &\frac{\partial \phi}{\partial t} + \frac{\partial}{\partial x}(\phi u) + \frac{\partial}{\partial z}\left\{\frac{1}{\mathcal{C}} B\phi(1-\phi) - \frac{\mathcal{D}}{\mathcal{C}}\frac{\partial \phi}{\partial z}\right\} \\ &-\frac{1}{\mathcal{C}}\frac{\partial}{\partial z}\left\{\left[\mathcal{P}_0 + (1-z)\right](1 + B(1-\phi))\frac{\partial \phi}{\partial z}\right\} = 0 \end{aligned} \tag{2.26}$$

$$\text{where} \quad \mathcal{C} = \frac{sc}{Tg}, \quad \mathcal{D} = \frac{d}{gs}, \quad \mathcal{P}_0 = \frac{P_0}{\rho g s}.$$

Here, ϕ is the solid volume fraction of small-sized population and B denotes the segregation rate caused by large-sized grains. The two-dimensional space is discretised into a $m \times n$ array with step-sizes Δx and Δz in x and z direction respectively, and solid volume fraction $\phi_{i,j}$ is stored on each node of the array. We are solving the segregation equation 2.26 coupled with an initial condition such as $\phi_{i,j} = 0.5 \ \forall i, j$ at $t = 0$. At the start of each time-step, Kurganov-Tadmor scheme starts by taking local gradients

of $\phi_{i,j}$ in spatial dimensions:

$$(\phi_x)_{i,j} = minmod\left(\frac{\phi_{i+1,j}-\phi_{i,j}}{\Delta x}, \frac{\phi_{i,j}-\phi_{i-1,j}}{\Delta x}\right)$$
$$(\phi_z)_{i,j} = minmod\left(\frac{\phi_{i,j+1}-\phi_{i,j}}{\Delta z}, \frac{\phi_{i,j}-\phi_{i,j-1}}{\Delta z}\right)$$
$$\text{where} \quad minmod(a,b) = \frac{1}{2}\left(sign(a)+sign(b)\right) \times \min(|a|,|b|). \tag{2.27}$$

The *minmod* operator takes both the forward and backward piecewise

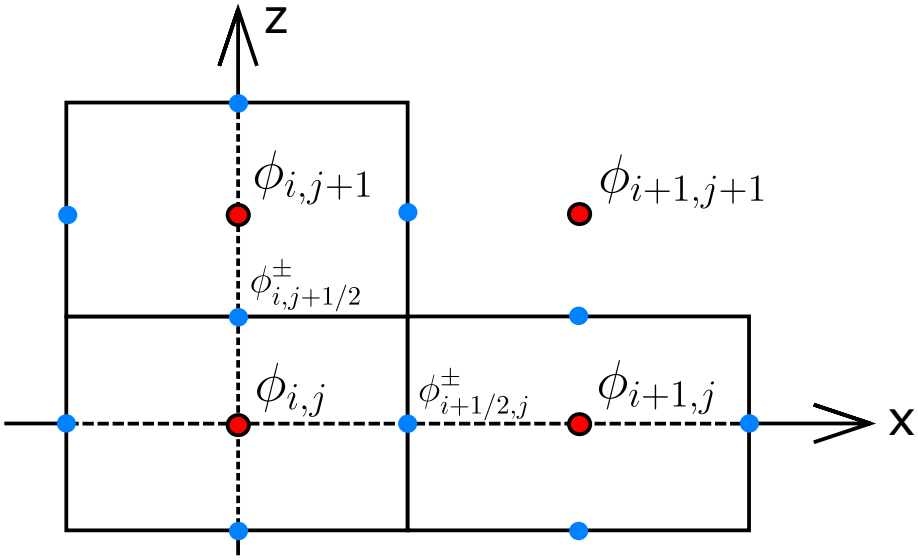


Figure 2.2: Illustrations of the node values and grid-point values in Kurganov-Tadmor scheme. Initial condition and calculated solutions of solid volume fraction ϕ are stored on nodes (shown as red points), the midpoints (shown as blue dots) in-between adjacent nodes are referred to as grid-points because they form a grid enveloping all the nodes. Intermediate estimates $\phi^{\pm}_{i+1/2,j}$ and $\phi^{\pm}_{i,j+1/2}$ are calculated for each grid-point based on piece-wise local gradients of the node values. Based on intermediate values, local speeds of propagation can be numerically calculated, leading to a numerical estimate of the flux values, effectively turning the non-dimensionalised segregation equation into a time-dependent 1^{st}-order ODE. This is then numerically solved by a 2^{nd}-order Runge-Kutta method for each timestep.

gradients into account and does a good job at avoiding shocks, ensuring the non-oscillatory property (Kurganov and Tadmor, 2000). As shown in Figure 2.2, we can take halfway points between each adjacent node value of $\phi_{i,j}$. These halfway points form a grid surrounding each node

value and allow midpoint values to be calculated and stored for numerical approximations of the flux terms. These approximated intermediate values at grid-points are defined as

$$\begin{aligned}\phi^{\pm}_{i+\frac{1}{2},j} &= \phi_{i+\frac{1}{2}\pm\frac{1}{2},j} \mp \frac{\Delta x}{2}(\phi_x)_{i+\frac{1}{2}\pm\frac{1}{2},j} \\ \phi^{\pm}_{i,j+\frac{1}{2}} &= \phi_{i,j+\frac{1}{2}\pm\frac{1}{2}} \mp \frac{\Delta z}{2}(\phi_x)_{i,j+\frac{1}{2}\pm\frac{1}{2}}.\end{aligned} \tag{2.28}$$

The local maximal speed $a^x_{i+1/2,j}$ and $a^z_{i,j+1/2}$ are defined at grid-points:

$$\begin{aligned}a^x_{i+\frac{1}{2},j} &= \max_{\phi^-_{i+1/2,j},\phi^+_{i+1/2,j}} \rho\left(\frac{\partial f(\phi)}{\partial \phi}\right) \\ a^z_{i,j+\frac{1}{2}} &= \max_{\phi^-_{i,j+1/2},\phi^+_{i,j+1/2}} \rho\left(\frac{\partial g(\phi)}{\partial \phi}\right) \\ \text{where} \quad f(\phi) &= \phi u, \quad g(\phi) = \frac{1}{\mathcal{C}} B\phi(1-\phi).\end{aligned} \tag{2.29}$$

Here $f(\phi)$ and $g(\phi)$ are the flux functions in x and z directions from segregation equation 2.24, and $\rho(\phi) = \max_i(|\lambda_i(\phi)|)$ where λ_i denote eigenvalues of ϕ. Conveniently, for a bi-disperse case,

$$\begin{aligned}a^x_{i+\frac{1}{2},j} &= \max_{\phi^-_{i+1/2,j},\phi^+_{i+1/2,j}} \left|\frac{\partial f(\phi)}{\partial \phi}\right| \\ a^z_{i,j+\frac{1}{2}} &= \max_{\phi^-_{i,j+1/2},\phi^+_{i,j+1/2}} \left|\frac{\partial g(\phi)}{\partial \phi}\right| \\ \text{where} \quad f(\phi) &= \phi u, \quad g(\phi) = \frac{1}{\mathcal{C}} B\phi(1-\phi).\end{aligned} \tag{2.30}$$

The numerical convection flux approximations are therefore defined

$$\begin{aligned}H^x_{i+\frac{1}{2},j} &= \frac{f(\phi^+_{i+\frac{1}{2},j}) + f(\phi^-_{i+\frac{1}{2},j})}{2} - \frac{a^x_{i+\frac{1}{2},j}}{2}\left[\phi^+_{i+\frac{1}{2},j} - \phi^-_{i+\frac{1}{2},j}\right] \\ H^z_{i,j+\frac{1}{2}} &= \frac{g(\phi^+_{i,j+\frac{1}{2}}) + g(\phi^-_{i,j+\frac{1}{2}})}{2} - \frac{a^x_{i,j+\frac{1}{2}}}{2}\left[\phi^+_{i,j+\frac{1}{2}} - \phi^-_{i,j+\frac{1}{2}}\right] \\ \text{where} \quad f(\phi) &= \phi u, \quad g(\phi) = \frac{1}{\mathcal{C}} B\phi(1-\phi).\end{aligned} \tag{2.31}$$

Since numerical fluxes are calculated and stored on grid-points, it is essential to impose spatial boundary conditions for boundary grid-points. Considering the ultimate goal of modelling cyclic loading within a shear box, we propose periodic inflow and outflow boundary conditions in x direction and zero-flux boundary condition in z direction. We hope this can provide a reasonable ground for simulating granular behaviour near the central columns. One of the reasons behind this choice is that, as we get closer towards the horizontal sides of shear box, re-arrangement of grains might be too volatile for a continuum model to provide predictions.

The remaining terms in the segregation equation 2.24 all have 2^{nd}-order spatial derivatives and are also numerically approximated as an entire 'diffusion term'. Due to the emphasis on vertical diffusion as well as the horizontal periodic boundary condition, the bi-disperse segregation equation 2.26 only contain spatial derivatives in z direction inside the diffusion flux terms. Therefore, the numerical diffusion flux approximations at grid-points are calculated using forward gradients of piece-wise node values of ϕ as numerical derivatives:

$$P^z_{i,j+\frac{1}{2}} = \frac{1}{2}\left[Q\left(\phi_{i,j}, \frac{\phi_{i,j+1} - \phi i,j}{\Delta z}\right) + Q\left(\phi_{i,j+1}, \frac{\phi_{i,j+1} - \phi i,j}{\Delta z}\right)\right]$$
$$\text{where} \quad Q(\phi, \frac{\partial \phi}{\partial z}) = \frac{\mathcal{D}}{\mathcal{C}}\frac{\partial \phi}{\partial z} + \frac{1}{\mathcal{C}}\left[\mathcal{P}_0 + (1-z)\right](1 + B(1-\phi))\frac{\partial \phi}{\partial z}. \quad (2.32)$$

After numerically approximating each flux term in spatial dimensions, the system of PDEs (in bi-disperse case only one equation) takes form of a time-dependent ODE

$$\frac{d}{dt}\phi_{i,j} = -\frac{H^x_{i+\frac{1}{2},j} - H^x_{i-\frac{1}{2},j}}{\Delta x} - \frac{H^z_{i,j+\frac{1}{2}} - H^z_{i,j-\frac{1}{2}}}{\Delta z} + \frac{P^z_{i,j+\frac{1}{2}} - P^z_{i,j-\frac{1}{2}}}{\Delta z}. \quad (2.33)$$

All the terms on the right hand side are numerically calculated for each

time-step. We use 2$^{\text{nd}}$-order Runge-Kutta method (Heun's method) as the time-stepper to solve the ODE for $\phi_{i,j}$ and use it as the initial condition for the following time-step:

$$\begin{aligned}\hat{\phi}_{t_i+1} &= \phi_{t_i} + tR\left(t_i, \phi_{t_i}\right) \\ \phi_{t_i+1} &= \phi_{t_i} + \frac{\Delta t}{2}\left[R\left(t_i, \phi_{t_i}\right) + R\left(t_{i+1}, \hat{\phi}_{t_i+1}\right)\right].\end{aligned} \tag{2.34}$$

Here, ϕ_{t_i} denotes the ϕ value at i-th time-step in time, $R(t_i, \phi_{t_i})$ is the right hand side function from equation 2.33. The intermediate value $\hat{\phi}_{t_i+1}$ is calculated first and then serves to increase the accuracy of the final prediction for next time-step.

2.6 Summary

In this chapter we have constructed a time-dependent problem for the purpose of modelling granular behaviour inside a shear box. A shear box is a classic, idealised set-up that has many of the features that will allow us to explore geotechnical problems involving the interactions between soil grains and cyclically loaded structures within them. Following the steps of the steady-state kinetic-sieving chute flow model (Gray and Thornton, 2005), we adopt its mixture theory framework and use solid volume fraction ϕ^{ν} to represent distribution of any particular granular population categorized in its size. After imposing assumptions of geo-static stress profile and interaction drag β^{ν} (Thornton and Gray, 2008) between granular populations, an expression of any population's relative velocity to the bulk 2.14 can derived.

One key departure from the mixture theory occurs when stress is partitioned among different populations in kinetic-sieving model. After cov-

ering the basic requirements for the stress partition functions, we adopt the pair-wise set-up (Gray and Thornton, 2005) as a starting point due to its mathematical simplicity and potential convenience in experimental parameter measurement. At this point, a basic system of PDEs have been established. We then used scales such as typical height of the granular mixture and shear box length for non-dimensionalisation and end up with a system of PDEs, equation 2.24. This is mathematically similar to the original chute-flow model and can also be solved using the Kurganov-Tadmor convection-diffusion numerical scheme.

In the following chapter we will develop this system further to understand how the cyclic loads which develop the shear in the geotechnical case can be modelled.

Chapter 3

Cyclic Loading

This chapter describes the modification of the kinetic sieving model to capture granular behaviour in a cyclic loading set-up. In particular we consider the choices of time-evolving velocity profiles and their feedback to the granular behaviour.

Contents

3.1 Driving through cyclic shear

In order to model the cyclic, confined conditions of geo-technical loads, such as those on the soil around a wind turbine foundation, we are investigating the geometry of a shear box (Figure 2.1). Here, the grains are mobilized by an imposed shear on the top lid, mimicking the action of an oscillating structure surface on neighbouring soil grains in its vicinity.

So far, we have put together a time-dependent continuum problem that models granular behaviour within a shear box where the top plate is shearing the granules back and forth, creating void spaces locally and driving the different-sized granules to segregate into packets of high-concentration regions. The kinetic-sieving model (Gray and Thornton, 2005) was originally designed to model steady-state chute flows and achieved excellent agreement with experimental data (van der Vaart et al., 2015). In previous sections, we modified this configuration to include time dependence, a normal confining pressure and we explored new flux term structures to the kinetic sieving model. These changes adapted what was originally designed as a steady-state chute flow model to a cyclic loading shear box scenario.

In this chapter, we consider the nature of the driving shear term at the surface of the cyclic cell and how this boundary condition affects segregation.

3.1.1 Cyclic implementation

Reflecting the horizontal symmetry of the setup, we adopt periodic boundary conditions. When reversal of top plate shear direction takes place, we prescribe the solid volume fraction, ϕ, of the original downstream boundary as a constant inflow boundary condition, and the former inflow boundary is

given the ϕ values from its closest neighbour column under quasi-steadiness assumption.

Physically, this implies a 'central' part of the granular mass is modelled, because we are assuming a quasi-steady horizontal inflow and outflow condition for the modelled region. Therefore, instead of rocks and boulders trapped inside a metal box, the granular behaviour we are trying to simulate here bare a closer resemblance towards a small section of sand particles underneath a long metal pipe exerting periodic horizontal shear.

The periodic boundary conditions are chosen in part due to concern over the behaviour of a granule close to boundaries and the reflection of those behaviours within a mixture theory framework: We are assuming the volume fraction to be similar across the modelled region. But if we consider a small confined model region, the granular material would soon be sheared towards one side, making the bulk material denser than the other side, effectively breaking our volume fraction assumption. Furthermore, we expect granular motion to be more complicated near confined boundaries. In reality, the imbalance of volume fraction as described before could potentially create void space so large that the mixture can no-longer be modelled as a continuum. Hence, periodic boundary conditions in the horizontal, x, direction have been adopted.

Although the simulation results are two-dimensional, it makes sense to only extract the central column from each time-step. There are two reasons for this simplification. First, we are mostly interested in normal transport and diffusion flux terms in z direction, leaving flux terms in x direction less dominant, hence ϕ has relatively less significant horizontal variations. Second, changes in horizontal flux term magnitude are largely determined by size of ϕu, but the velocity profile can vary on a case-by-case basis, and we

are currently only imposing profile without a constitutive law, once again making it difficult to give a general prediction for ϕ changes in x direction. In later sections, we will demonstrate that choice of velocity profile can still influence simulation results when coupled with additional assumptions. For now, we only sample the central column of ϕ from each time-step, combining them into a matrix containing solid volume fraction evolution data. This effectively reduces the model to a single spatial dimension z.

3.2 Validation against published results

The oscillatory shear experiment (Scott and Bridgwater, 1975) conducted by van der Vaart *et al.* (2015) provides a set of reproducible results to draw comparison with thanks to its non-intrusive measurement method of all granular particles and laterally-uniform configurations. Additionally, the authors also employed the continuum-based kinetic sieving model to generate theoretical predictions of how the solid volume fraction of small particles ϕ evolves in time with the motivation being to justify the introduction of asymmetric segregation flux function in $z-$direction. Hence, we can validate our time-dependent solver and choice of lateral boundary conditions through comparison with these data, before exploring the effects of the confining pressure and imposed, time-dependent velocity profile.

With similar structure to equation 2.24, the advection-diffusion equation to be solved for this side-wall driven oscillatory shear flow takes the form

$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\phi \mathbf{u}) - \frac{\partial}{\partial z}\left[qF(\phi)\right] = \frac{\partial}{\partial z}\left(D\frac{\partial \phi}{\partial z}\right). \tag{3.1}$$

where $\mathbf{u}$ denotes the bulk velocity profile, q is the mean segregation velocity within in shear box, D denotes diffusivity and $F(\phi)$ is the segregation

flux function. Derived from the simplest form of stress partition function equation 2.15, the resultant segregation flux function takes the form

$$F(\phi) = \phi(1-\phi). \tag{3.2}$$

Due to the lateral uniformity of oscillatory shear box, the bulk velocity profile has only horizontal components with magnitude dependent on depth z and time t, i.e. $\mathbf{u} = (u(z,t), 0, 0)$. After non-dimensionalisation, equation 3.1 effectively reduces to a one-dimensional convection-diffusion equation

$$\frac{\partial \phi}{\partial t} + -\frac{\partial}{\partial z}\left(S_r F(\phi)\right) = \frac{\partial}{\partial z}\left(D_r \frac{\partial \phi}{\partial z}\right). \tag{3.3}$$

It was noted that the trajectories of large and small particles suggest that small particles reach the bottom boundary faster than the large particles reach the top boundary. Hence, a novel asymmetric flux function devised by Gajjar and Gray (2014) was introduced

$$F(\phi) = A_\kappa \phi(1-\phi)(1-\kappa\phi). \tag{3.4}$$

Here, the asymmetry parameter κ can be used to configure the flux function from quadratic to cubic form, and the amplitude parameter A_κ serves to guarantee unchanged flux amplitude. After substituting the asymmetric flux function, the numerical solution provides a closer prediction towards experimental measurement of ϕ evolution, as shown in Figure 3.1. It can be seen that this one-dimensional time-dependent problem has very similar mathematical structure to the two-dimensional time-dependent problem covered in section 2, when combined with periodic boundary conditions. Such mathematical resemblance presents an opportunity to validate the time-dependent continuum implementation of this work against published data. Using the same velocity profile, segregation rate and diffusion rate

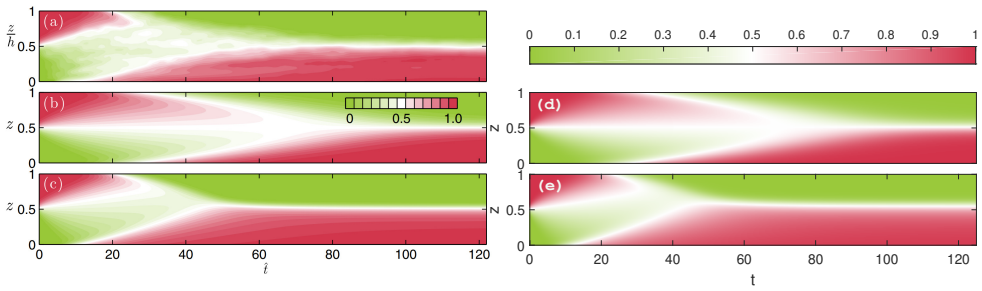


Figure 3.1: Figures produced by van der Vaart et al. (2015): (a) Volume fraction, ϕ, data extracted using refractive index-matched scanning under oscillatory shear. (b) Theoretical prediction of small particle solid volume fraction ϕ using bi-disperse kinetic sieving model using the symmetric flux function, with $S_r = 0.016$ and Péclet number $S_r/D_r = 20.9$. (c) Theoretical prediction of ϕ using the asymmetric cubic flux function proposed by Gajjar and Gray (2014), with $S_r = 0.030$ and Péclet number $S_r/D_r = 29.6$. $(d)(e)$ Our time evolution graph numerical solution ϕ to the non-dimensionalised segregation-diffusion problem obtained using the symmetric flux and asymmetric flux and equivalent periodic boundary condition to match the oscillatory shear box problem. The parameters S_r, D_r, κ and A_κ are chosen to match with the results produced by van der Vaart et al. (2015).

parameters as featured in original work (van der Vaart et al., 2015), and recording central column of the 2D solution under periodic boundary condition, a time evolution heat graph can be generated for both symmetric and asymmetric flux functions, shown in Figure 3.1. Qualitative and quantitative agreement can be seen between the published experimental data and model implementation of Figures 3.1(a), (b) and (c) and the equations derived here - subject to periodic boundary conditions - Figures 3.1(d) and (e). This agreement justifies the functionality of our continuum model for granular segregation, and sets the foundation for further modifications on time-varying velocity profile and shear rate dependence in later sections.

3.3 Time-evolving velocity profile

3.3.1 Desired characteristics

Within a cyclic shearing environment with periodic back-and-forth motion exerted from the top interface, it is natural to expect that how velocity profile changes in time would have a strong contribution or impact onto granular behaviour. We need to provide a time-evolving velocity profile that reflects the features of the periodic, surface-driven shear cell. A goal should be to use a constitutive law in ways such as a $\mu - I$ or kinetic theory approach (Montanero et al., 1999; Jenkins and Berzi, 2012) to directly relate the shear rate of the granular mixture with the imposed stress. However, there are many open choices and questions in such an approach which would also add complexity to the solution method. Here, for simplicity and as an initial step, we will impose on our system a time-evolving velocity profile function $u(z, t)$ that fits with observations. This is an approach that could be revisited in further work.

Therefore, this section starts by exploring what criteria need to be met for such a velocity profile, and what features we should anticipate the profile to have based on existing knowledge and assumptions.

Inside a cyclic loading environment, it makes sense to assume that velocity profiles undergo active and complex changes in time. Unlike steady-state granular flow, extraction of velocity profile under non-steady motion proves challenging. Measurements on boundary surfaces are most convenient, but usually introduce additional concerns such as interface friction and slipping effects (Forterre and Pouliquen, 2008). On the other hand, measuring velocity profile internally poses more technical difficulties and often requires remarkable efforts to make the process less intrusive to the material (Pu-

dasaini et al., 2005). Discrete element models (Ketterhagen et al., 2007; Marks et al., 2012b) have also been used to explore velocity profiles within granular mixtures, but the results are often sensitive to parameter and setting choices (Brandao et al., 2020b).

Since this thesis revolves around modifying the continuum model and directly relevant cyclic shearing experiments have been conducted yet, we want our non-steady velocity profile to be as general and flexible as possible. However, we do have a basic understanding of the desired features of such velocity profiles. It needs to be capable of qualitatively fitting known granular profiles in similar shear flow scenarios, but can also be easily tweaked for substituting specific experimental readings, such as surface velocity measurements. Additionally, previous cyclic shearing experiments indicate that the shape of velocity profile is hysteretic, changing upon reversal of the shear; so there needs to be a way of manipulating shape of velocity profile in time.

3.3.2 First attempt: Discrete sequence

Data from chute flow models (Johnson et al., 2012), annular shear cell experiments (Golick and Daniels, 2009) and various DEM simulations (Qiao et al., 2021) show that a number of factors affect the strain response to shear. These factors include surface roughness of the shearing surface and granular beads, curvature influences in annular shear cells, packing fraction of the granular material, shear strain rate and even crush-ability of the material.

We can attempt to capture some of these effects through our choice of evolving profile, i.e. prescribing a sequence of velocity profiles that not

only adopt the key shapes observed in shear layer velocity profiles, but are also based on empirical expectations of how such velocity profiles might behave in a cyclic-loading shear box environment.

For simplicity of implementation, while maintaining the appropriate geometries of the cyclic shear cell, we propose a series of hyperbolic velocity profiles. Ideally, conducting a poly-disperse granular shearing experiment and measuring how velocity profile evolves under cyclic shear would be preferable. However, the Covid-19 pandemic made planning and conducting experimental studies challenging. Therefore, our choice of hyperbolic profiles is mostly inspired by published cyclic shear cell experimental studies (Golick and Daniels, 2009) as shown in Figure 3.2. However, it should made clear that the closest experimental study (May et al., 2010) featured a bottom-driven annular shear cell instead of a top-driven one. Therefore, we are employing the hyperbolic profiles as a starting point, and have made sure that experimentally-measure velocity profiles can indeed be implemented into the model conveniently in future. For the remaining part of this thesis, we will continue exploring with hyperbolic velocity profiles.

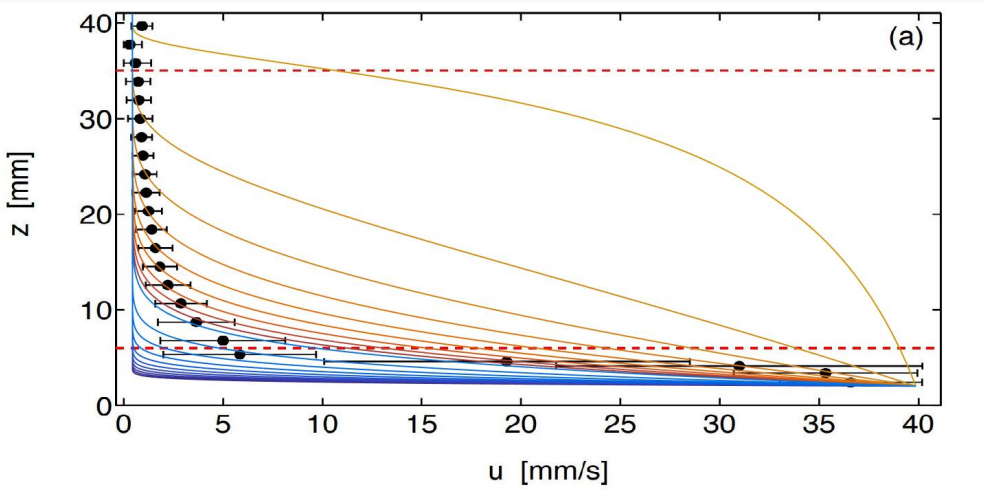


Figure 3.2: Side-by-side comparison between the velocity profile measured by May et al.(2010) and the family of hyperbolic velocity profiles used in this thesis, this time inverted in z.

The hyperbolic form we adopt for the velocity $u(z)$ at an elevation z from the base of the shear box

$$u(z) = \frac{1}{u_{norm}} \left[\tanh\left(\frac{\beta}{\pi z}\right) - 1 \right], \tag{3.5}$$

where $u_{norm} = \tanh\frac{\beta}{\pi} - 1$ serves to normalize velocity profile magnitude based on surface speed and β is the shape-adjusting parameter: an increase in β results in a faster decay of horizontal speed in depth, i.e. darker-colored curves shown in Figure 3.3(a).

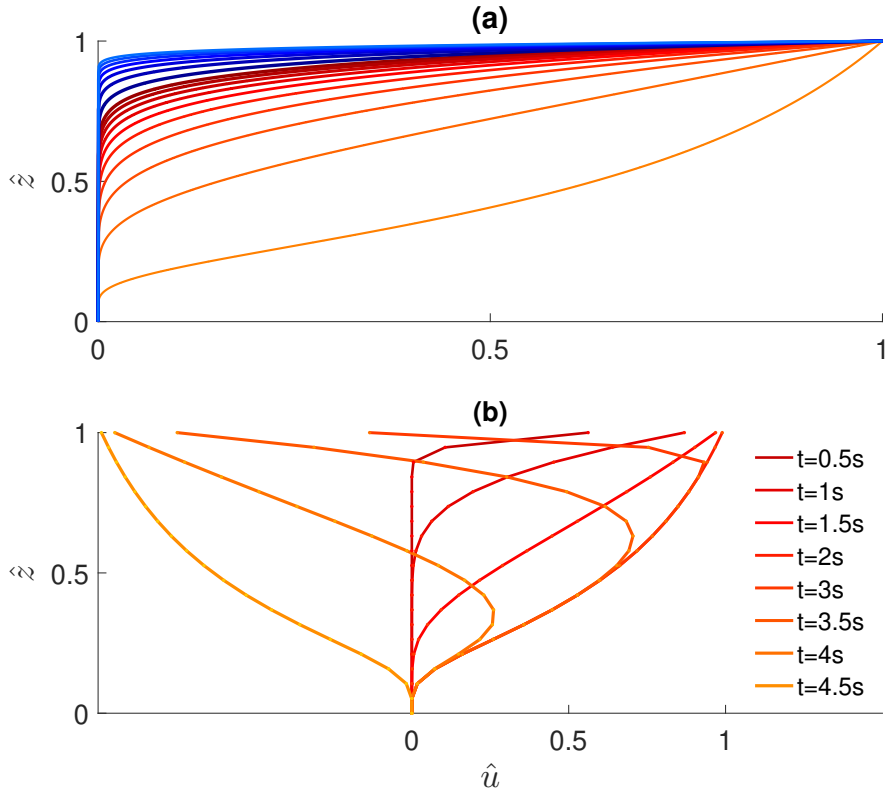


Figure 3.3: (a) Illustration of how $u(z)$ iterates to mimic initialisation process as the layer becomes mobilised by the applied top shear. Time increases with blue through red to orange colour; (b) Illustration of how $u(z)$ iterates over one complete shear cell cycle.

Inside a shear box where the top plate is shearing the granular material back and forth periodically, it makes sense to expect velocity profile to have maximum magnitude towards top surface and decay towards bottom. The

velocity is thus expected to be at maximum when $z = s$, where grains are directly in contact with and driven by top plate, tending to zero towards the base of the cell to facilitate the non-slip boundary condition. The hyperbolic profile can be adjusted via the shape parameter β to mimic the gradual increase of horizontal motion in depth as shearing motion continues especially upon initialization. As shown in Figure 3.3(a), the warm-colored group of functions can simulate how velocity profile would evolve in time as the shearing starts (β decreases from 10 to 0.5 as the curve color change from dark red to light orange). However, during the earliest stages of mobilization, we anticipate the velocity profile to take a much more top-heavy shape as the motion is still being transferred to bottom. Unfortunately, setting β beyond a certain threshold would result in non-smooth profiles. Hence, the hyperbolic function is stretched upwards in z-direction (instead of changing β because the curve becomes non-smooth as β becomes too big) to form the dark-colored curves. Combining the two groups of functions together, an reasonable iteration sequence of velocity profile upon initialization is complete.

When the top plate changes shearing direction, we expect grains near top surface to decelerate and consequently change direction of velocity along with top plate, while such reversal of horizontal motion also take place in lower regions with a slight 'hysteresis' due to their distance from the top. In order to simulate this effect, we take difference increments of velocity profiles in-between time-steps for both directions and use these truncated values to model evolution of velocity during reversal process as shown in Figure 3.3(b). As the top velocity changes direction, grains below the surface is prescribed to change their movement direction according to the surface profile, but in a slightly delayed fashion. Another reason for using sequences of truncated values is mathematical simplicity, as the interme-

diate curves are not possible to generate by tweaking parameters of the hyperbolic profile.

In this section, we have assembled a sequence of velocity profiles attempting to capture the gradual propagation of velocity down depth and hysteresis effect upon reversal. However, being a discrete sequence poses some limits: the sequence have to be generated and sometimes adjusted before running the simulation; the group of stretched profiles (dark-colored ones in Figure 3.3(a)) are also potentially unreliable because their shapes don't necessarily align with main group, sometimes requiring additional calibration. In hindsight, another overlooked element is the attention to surface velocity magnitude. In fact, the surface velocity magnitudes start with a fixed value upon initialization, which is not accurate: it should rise from 0 until the maximum is reached. Additionally, parameters used to adjust profile behaviour are not linked to non-dimensionalizing scales or crucial experimental values such as shear cycle period and surface velocity magnitude, making it difficult to compare or calibrate with experimental data. Simulation results using this version of velocity profile can be found in section 3.3.4.

3.3.3 Second attempt: Continuous function

Indeed, generating a sequence of velocity profiles beforehand for the simulation leaves room for improvement. This compromise is made in order to see if reversal behaviour could be captured using the method. Naturally, we now advance to improve the drawbacks and limitations by proposing a new set-up of hyperbolic profiles, this time having a firmer anchor in reality.

$$u(z) = \frac{1}{u_{norm}} \left[\tanh\left(\frac{\beta}{\pi z}\right) - 1 \right] \tag{3.6}$$

where

$$u_{norm} = \frac{\left(\tanh\frac{\beta}{\pi} - 1\right)}{u_s},$$

β is the shape-adjusting parameter and u_s is the surface velocity magnitude. As shown in Figure 3.4(a), β still controls how the velocity retains its magnitude as it propagates in depth. Figure 3.4(b) demonstrates how u_s forces the velocity profile to take a certain value at surface $z = s$. This feature can simplify the progress of tweaking the profile against experimental measurements. The biggest change we made is how β and u_s change in time. It has been discussed in previous section that, it makes sense to tie these parameters and their ways of iteration with the non-dimensionalisation scales and measurement values easy to retrieve from experiments. Therefore, we propose a scenario where β and u_s are defined by periodic trigonometric functions.

$$\beta = \sin\left(\frac{\pi t}{T}\right), \quad u_s = 5\cos\left(\frac{2\pi t}{T}\right) + 5.5 \tag{3.7}$$

where T is the shearing period. As shown in Figure 3.4(c) and Figure 3.5, the evolution curves of β and u_s are chosen to fit the conditions of initialization and reversal: When shearing starts from still, u_s should increase from 0 and β decreases from 10.5 to 0.5 to imply the gradual mobilization and acceleration of granules in depth. Upon reaching $T/2$, surface velocity touches its local maximum 1 and begin decreasing whereas the deceleration gradually propagate in depth, reflected by increase of β. The first reversal process finally concludes at $3T/2$, when u_s reaches local minimum of -1 and shape parameter β declines to 0.5 once again. Compared with the previous attempt, this version of velocity profile evolution has the following advantages: First, most of its parameters (u_s and T) can be easily measured and retrieved from field experiments. Second, it provides great

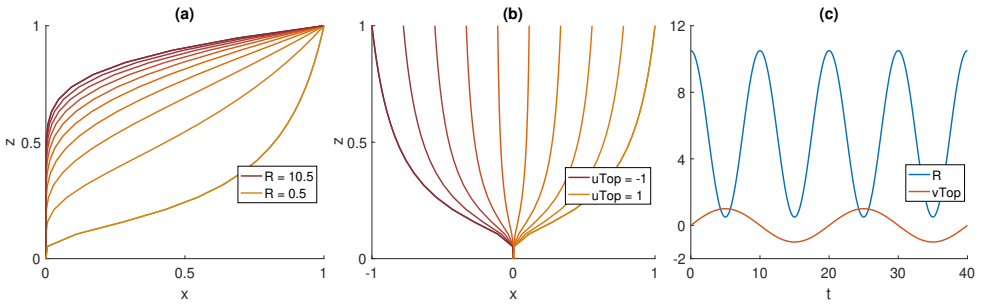


Figure 3.4: Illustration of how the control parameters manipulate velocity profile characteristics and evolve themselves. (a) Simulated velocity profile shape change via changing shape factor β; (b) Surface velocity condition enforced by u_s; (c) Design of how β and u_s periodically fluctuate.

freedom in choosing what kind of periodic behaviour the controlling parameters take, making it easy to calibrate. Third, using smooth functions instead of discrete arrays offers improved flexibility when it comes to adjusting the profile for different simulations. However, it should be noted that both versions of velocity profiles serve as initial solutions towards incorporating more shear-box-related features into the model. Having shear rate dependence take a more active role in constitutive law should be the goal moving forward.

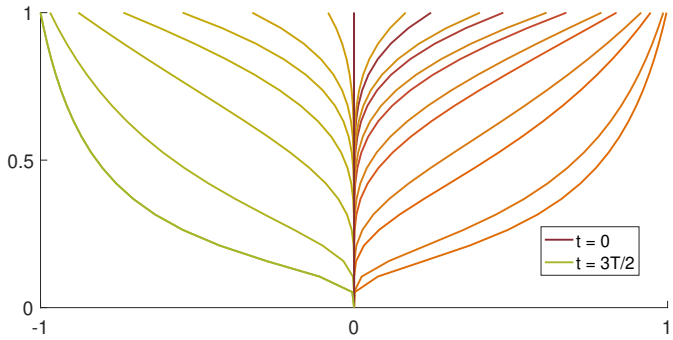


Figure 3.5: Illustration of how velocity profile evolves under the periodic set-up.

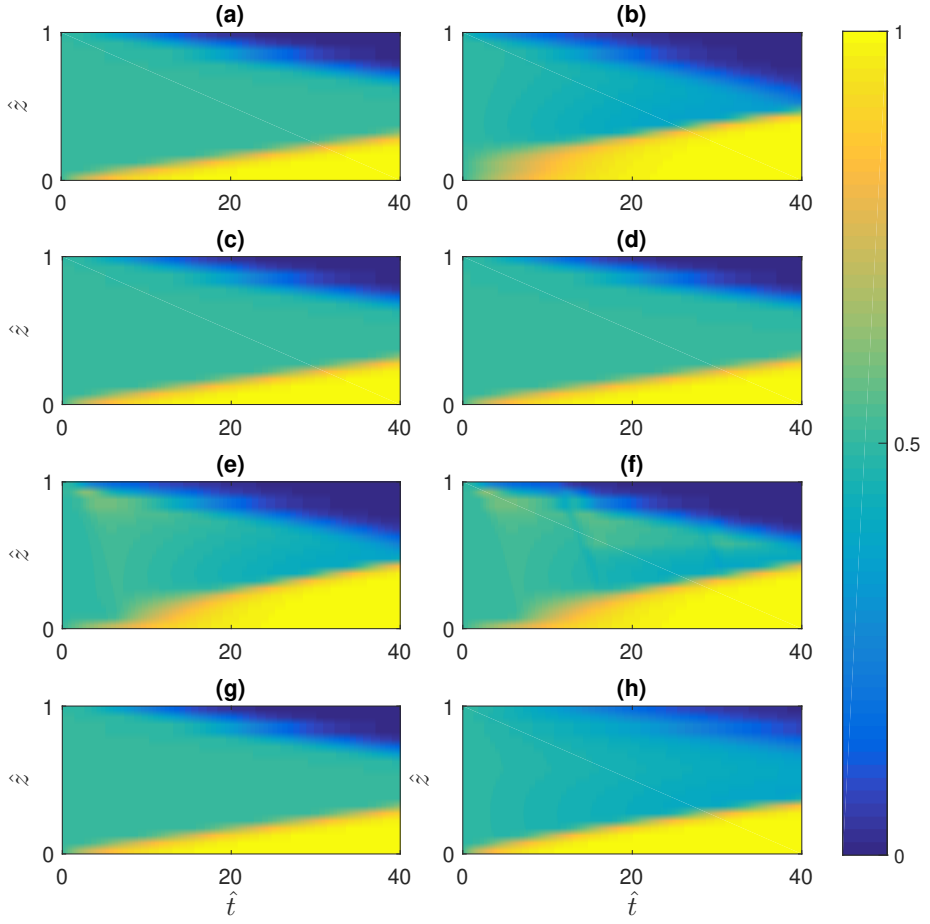


Figure 3.6: Heat maps presenting 2D bi-disperse simulation results for different configurations of discrete-sequenced velocity profiles and segregation rate dependence settings: (a) steady plug flow $\hat{u} = 0.5\ \forall \hat{z} \in [0, 1]$; (b) steady hyperbolic $u(z)$ as defined by equation 3.5 with $\beta = 1.0816$; (c) discrete hyperbolic $u(z)$ with only initialisation for $t \in [0, 5]$; (d) discrete hyperbolic $u(z)$ with initialisation and reversal events at $t = 10, 25$; (e) discrete hyperbolic $u(z)$ with only initialisation for $t \in [0, 5]$, now coupled with shear rate dependence enforced in $B = B_{base} + \sqrt{du/dz}\ B_0$ where $B_{base} = 0.5$ and $B_0 = 0.3$; (f) discrete hyperbolic $u(z)$ with with initialisation and reversal events at $t = 10, 25$, now coupled with shear rate dependence enforced in $B = B_{base} + \sqrt{du/dz}\ B_0$ where $B_{base} = 0.5$ and $B_0 = 0.3$; (g) continuous hyperbolic $u(z)$ as defined by equation 3.6; (h) continuous hyperbolic $u(z)$ as defined by equation 3.6, now coupled with shear rate dependence enforced in $B = B_{base} + \sqrt{du/dz}\ B_0$ where $B_{base} = 0.5$ and $B_0 = 0.3$.

3.3.4 Sensitivity of velocity profile choices

In the shear box scenario, one could argue that shear rate within the granular material should affect how actively granular material segregate in size. With this in mind, we experimented with adding shear rate dependency to the pairwise segregation rate $B_{\nu\mu}$ in a bi-disperse case using the formula

$$B_{updated} = \left(B_0 + \sqrt{\frac{\partial u}{\partial z}} \right) B_{base} \tag{3.8}$$

where B_0 is a background value and B_{base} is what was previously used as $B_{\nu\mu}$. The shear rate $\partial u/\partial z$ is obtained numerically from the velocity profile. As discussed in previous sections, different choices of velocity profile $u(z,t)$ does not produce dominant impact on ϕ evolution due to the size of horizontal flux and periodic boundary conditions. However, with the pairwise segregation rate forced to take a numerical shear rate dependence, we can really start to see feed-backs created by changes in velocity magnitude and direction. We start by experimenting with the sequenced version (as discussed in section 3.3.2) due to their sharper changes in velocity gradient.

As shown in Figure 3.6(a)-(h), feed-backs onto segregation timescale caused by various factors and dependence mechanisms are demonstrated. In each sub-figure, the horizontal axis corresponds to non-dimensionalised time and vertical axis corresponds to non-dimensionalised depth. For each simulation, only the central column of the 2D solution matrix is recorded at each time-step due to the observation of small horizontal differences mostly caused by periodic boundary conditions in x-direction. The collection of these 'column snapshots' form the heat maps shown here. Each simulation is run for $\hat{t} \in [0, 40]$ for a 2D bi-disperse simulation. Local distribution of the two particles is represented by ϕ, the solid volume fraction of small particles. In the sub-figures, noticeable differences in how quickly and how far

concentrated regions of high or low ϕ values form can be seen for different configurations.

Compared with the bi-disperse simulation result for plug flow, segregation appears to vary in depth once we use the hyperbolic velocity profile caused by the ϕu term in segregation equation 2.24. Additionally, subtle changes in segregation timescale caused by the initialisation and reversal events can be seen in Figure 3.6(c) and (d). A noticeably longer segregation timescale can be observed for cyclic shear (initialisation + reversal) than simple shear (initialisation + no reversal). The contrast can be amplified if we force segregation parameter to depend on numerical shear rate as defined in 3.8.

As shown in Figure 3.6(e) and (f), volume fraction ϕ visibly go through local fluctuations as reversal takes place and shear rate changes the most. This shows that more shear-rate-related behaviour could be incorporated in the continuum model, provided they are implemented under robust and realistic assumptions.

Figure 3.6(g) shows how ϕ evolves when the continuous function defined by equation 3.6 is introduced. As expected, when surface velocity is the same, differences in ϕu caused by different velocity profile evolution schemes are not significant. Lastly, Figure 3.6(h) demonstrates that smoother feedbacks to ϕ distribution compared to discrete profiles can be seen when shear-rate dependence is enforced, showing that sensitivity of simulated granular behaviour to velocity profiles can indeed be encouraged and customised to fit experimental data or specific modelling scenarios.

It should be made clear that the new patterns are only expected outcome made possible by our choice of modification and shear rate does not yet play a part anywhere further in the model. However, it does seem to produce periodic features related to cyclic loading. It remains for actual

experiments to guide whether such features are meaningful or not.

3.4 Bi-disperse results

Having described how the segregation equation is constructed and how solid volume fraction ϕ is numerically calculated using the Kurganov-Tadmor scheme (Kurganov and Tadmor, 2000) and Heun's method for each time-step, we are now able to simulate granular behaviour for elongated time. As shown in Figure 3.7 and Figure 3.8, effect of changing key parameters $\mathcal{B}$, $\mathcal{C}$, $\mathcal{D}$ and $\mathcal{P}_0$ are demonstrated by the differences of segregation behaviour.

Each subplot heat-map corresponds to the time evolution of a two-dimensional bi-disperse simulation. Due to the horizontal periodic boundary condition, we only sample the central column from the ϕ matrix at each time-step, making the horizontal axis of each heat-map t. Each simulation starts with the same initial condition: 50% large particles at the bottom and 50% small particles on top. For reference, a default template is chosen with parameters $\mathcal{B} = 0.9, \mathcal{C} = 20, \mathcal{D} = 0$ and $\mathcal{P}_0 = 0.1$, and each row focuses on changing one of the four parameters.

As the pair-wise segregation rate parameter $\mathcal{B}$ rises, the relative segregation velocity and normal convection flux relative to the bulk also increase, making the grains reach neutral and later reversed positions quicker in time. Coefficient of interactive drag c contributes to $\mathcal{C}$, and an increase in $\mathcal{C}$ suppresses the grains' ability to dilate locally and rearrange, preventing and delaying the emergence of reversed high-concentration layers. $\mathcal{D}$ is a generic diffusion parameter in z direction, and $\mathcal{P}_0$ is the non-dimensionalized confining pressure. One very interesting observation is that, these two parameters seem to both cause the granular mixture to reach neutral state sooner

but also harder to accumulate into reversed regions. Mathematically, this makes sense because we have shown that $\mathcal{P}_0$ also controls a diffusion flux term.

3.5 Summary

In this section, we have made additional modifications to the kinetic sieving model. Specifically, we explored two versions of prescribed velocity profile evolution pattern, each designed to capture granular patterns seen in experiments. We started by validating bi-disperse version of our model against published results, making sure we are building on a firm foundation. Because there is not yet a constitutive law to determine how the velocity should evolve in time for granular material, desired characteristics are discussed before two versions of velocity profile are proposed, with one being discrete and one being continuous. Feed-backs of the chosen velocity profiles are also discussed as shear rate dependence is applied. We finish this chapter by showcasing the simulation results of the bi-disperse simulation as well as a brief analysis of effects onto segregation behaviour by key parameters.

In following chapters, we continue building the continuum model by exploring poly-dispersity and simulated breakage.

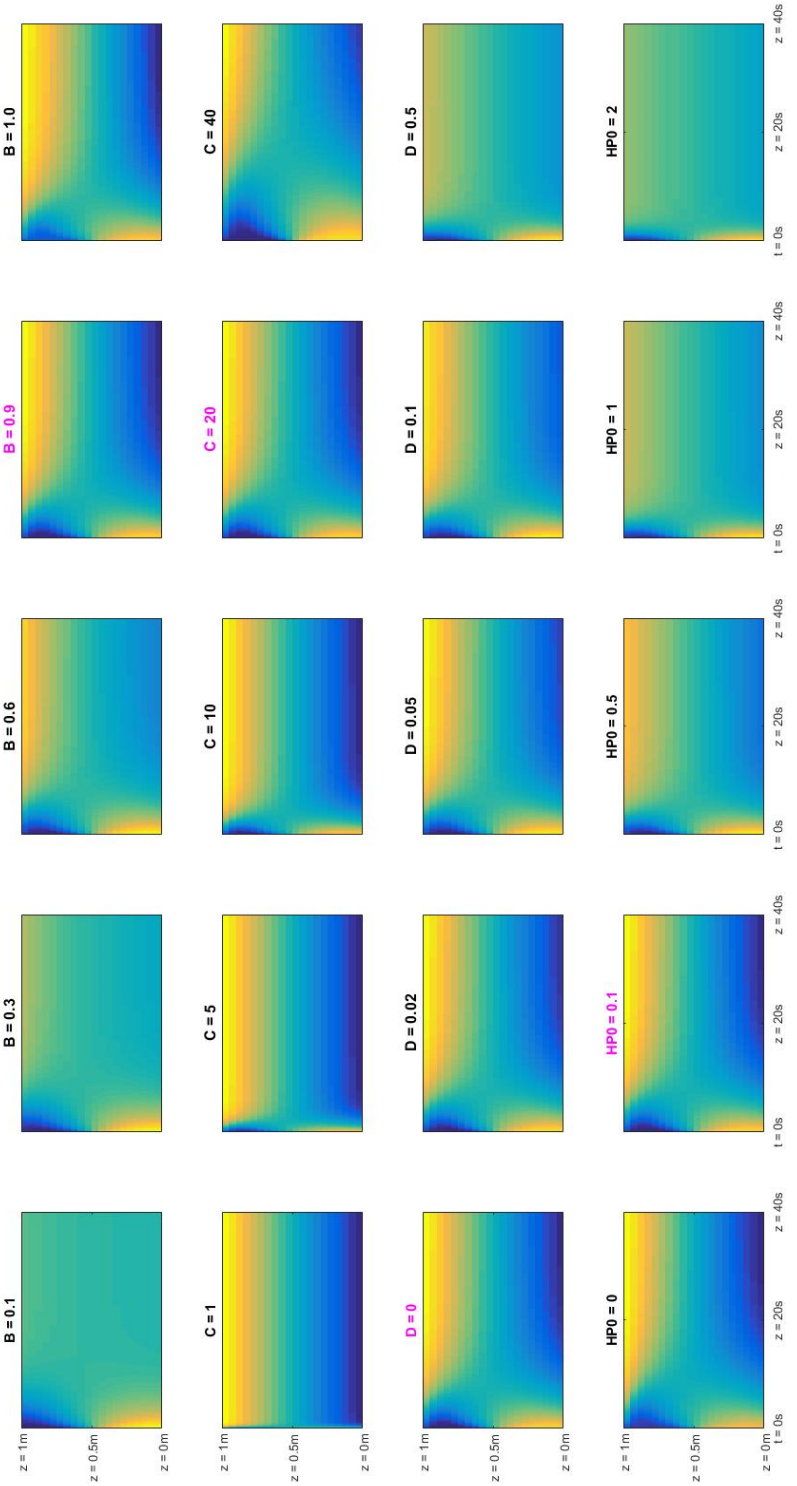


Figure 3.7: Results comparison for different input parameters, with the default template being $\mathcal{B} = 0.9, \mathcal{C} = 20, \mathcal{D} = 0$ and $\mathcal{P}_0 = 0.1$.

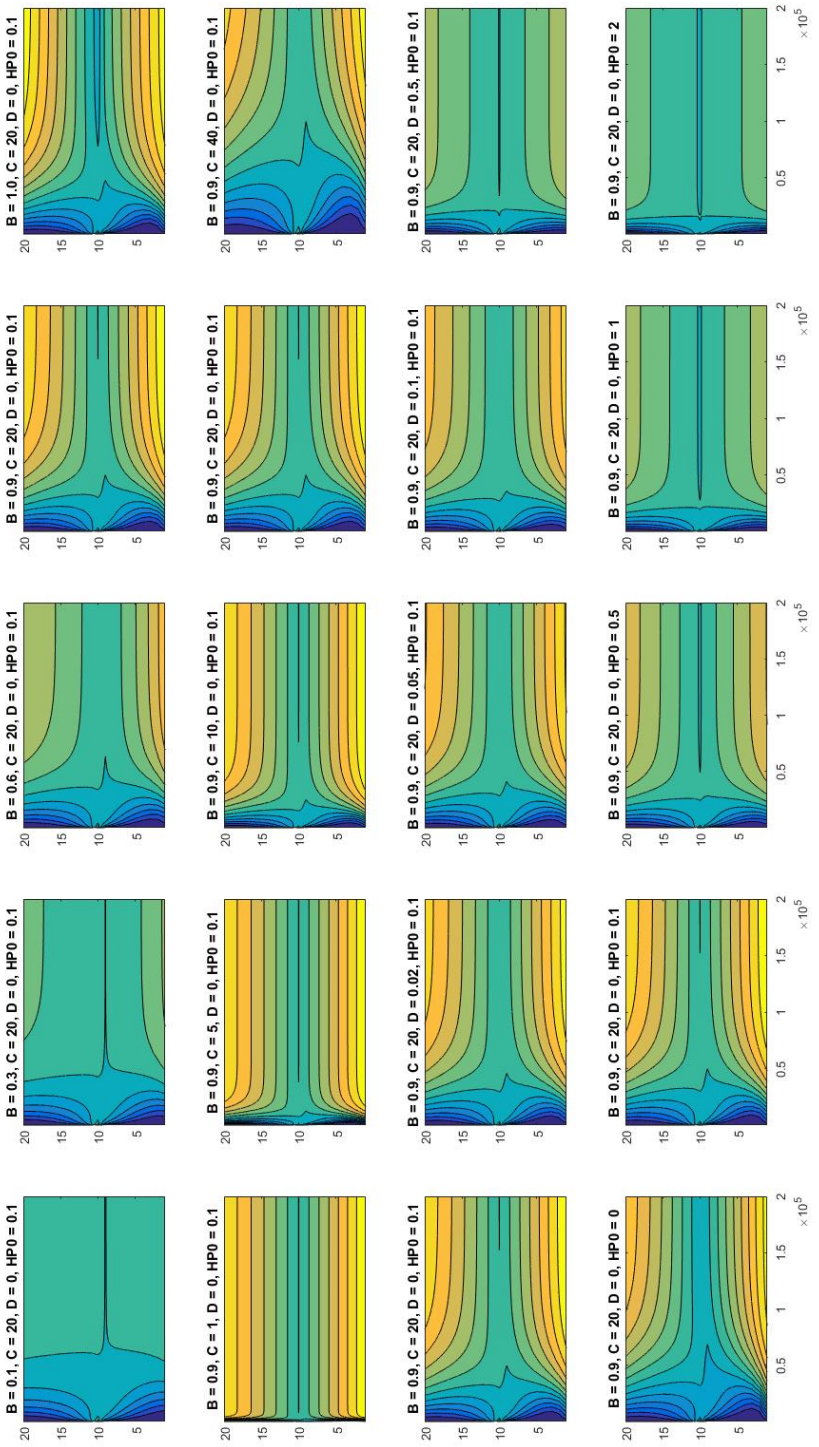


Figure 3.8: Segregation distance comparison for different input parameters, with the default template being $\mathcal{B} = 0.9, \mathcal{C} = 20, \mathcal{D} = 0$ and $\mathcal{P}_0 = 0.1$. Here, segregation distance denotes the difference from fully segregated expectation.

Chapter 4

Incorporating Poly-dispersity

This chapter walks through the process of expanding the kinetic sieving model from bi-disperse to poly-disperse. Various details about tackling model set-up during expansion is also discussed.

Contents

4.1 Importance of achieving poly-dispersity

One crucial element of geophysical granular mixtures is the great diversity of granular sizes. Specifically, it is almost impossible to find granular samples from sand, dry snow or dry soil consisting of only one or two size categories.

In previous chapters, we have made modifications to a continuum kinetic sieving model (Gray and Thornton, 2005) with the intention to prepare it for the goal of simulating the granular behaviour of a central region within a shear box, where the top plate is exerting cyclic shearing motion and normal pressure through contact. The list of modifications include the inclusion of normal pressure terms in segregation equation and two versions of generated velocity profile evolution patterns.

In this chapter, we extend the model to incorporate poly-dispersity, i.e. multiple populations categorized by granular sizes. We start by explaining how the scheme can in fact be translated into symbolic form for iterative expansion. After assembling a structure where the scheme can be appropriately expanded based on the number of size population and their information, we proceed to validate against existing bi-disperse and tri-disperse results. At last, we discuss various poly-disperse scenarios that might emulate features from different types of granular mixtures, before eventually showing and analyzing a few groups of simulation results modelling 10 different size populations.

4.2 Iterative expansion of species

Kinetic sieving model has been a well-known and powerful tool for modelling shallow quasi-steady chute flows (van der Vaart et al., 2015; Johnson et al., 2012). However, a great proportion of published works using this model are focused on bi-disperse (Thornton and Gray, 2008) and tri-disperse case (Gray and Ancey, 2011). Our ultimate goal remains to capture as many features of a geo-physical granular mixture under cyclic loading as possible. Therefore, it makes sense to go one step further and expand the population count from 2 and 3.

As defined in equation 2.15, rate of segregation activity is controlled by pairwise rates $B_{\nu\mu}$ between the pair of populations ν and μ. For a bi-disperse mixture, this set-up is straight-forward and convenient, but the relationships become rather complex as population count increases. In fact, discussion on impacts of how pairwise segregation rate (Gray and Ancey, 2011) has been made for the tri-disperse case. One important question to ask is: what dictates how big a value $B_{\nu\mu}$ takes for a certain pair of populations ν and μ? Based on segregation time scale measurements from experiments conducted by Golick and Daniels (2009) and May et al. (2010), Gray and Ancey (2011) proposed to model pairwise maximum segregation speed $q_{\nu\mu}$ ($B_{\nu\mu}$ multiplied by scale parameters) as a function of grain size ratio. The pairwise structure of prescribing segregation rate actually has a unique advantage: it can potentially simplify segregation relationships among multiple size populations into a superposition of pairwise relationships. Precisely due to the fact that the value of $B_{\nu\mu}$ need to be measured for each scenario, measuring for a group of pairwise relationships seems a much more convenient and feasible option. Going one step further, if we can find a suitable function or curve to describe how $B_{\nu\mu}$ evolves as size

ratio increases, then all the pairwise components of a poly-disperse segregation relationship can be mapped onto a an array of points on that curve, making visualizing and understanding the segregation relationships within the mixture a much more straight-forward task. Guided by the experimen-

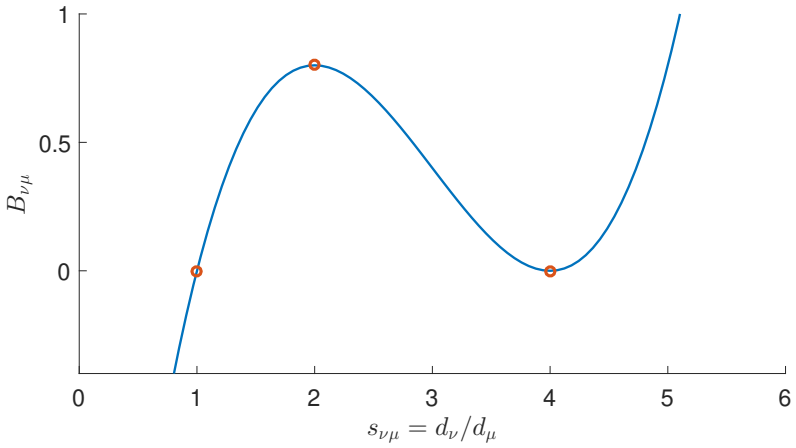


Figure 4.1: Illustration of an expected relationship between grain-size ratio $s_{\nu\mu}$ and non-dimensionalised segregation rate $B_{\nu\mu}$.The three highlighted points are at $s_{\nu\mu} = 1$, $s_{\nu\mu} = 2$ and $s_{\nu\mu} = 4$.

tal observation that segregation time scale t_s has a minimum at a specific size ratio (Golick and Daniels, 2009; May et al., 2010), a local maximum is suggested to exist for pairwise maximum segregation speed $q_{\nu\mu}$ (Gray and Ancey, 2011). This idea of having a local maximum for segregation rate makes sense to a certain degree, meaning that there exists an 'optimal size ratio' for granular segregation under a particular setup, coinciding with experimental observations of size ratio's influence on segregation behaviour from previous works (Scott and Bridgwater, 1975; Woodhouse et al., 2012; Gray, 2018).

As demonstrated in Figure 4.1, we adopt the assumption of a local maximum for segregation rate $B_{\nu\mu}$ and employed a 3rd power polynomial to capture the shape defined as below.

$$B_{\nu\mu} = \alpha \left((s_{\nu\mu} - 4)^3 + 3 (s_{\nu\mu} - 4)^2 \right) \tag{4.1}$$

where $\alpha = 0.2$ is an amplitude-controlling parameter and $s_{\nu\mu} = d_\mu/d_\nu$ is the size ratio. The second highlighted point at $s_{\nu\mu} = 2$ is where segregation rate reaches its local maximum by design. As mentioned in chapter 2, two crucial conditions have to be met when designing the pairwise segregation rate function:

$$\begin{aligned} B_{\nu\nu} &= 0 \quad \forall\nu \\ B_{\nu\mu} &= -B_{\mu\nu} \quad \forall\nu \neq \mu. \end{aligned} \tag{4.2}$$

The first condition implies that no segregation should take place within a mono-disperse mixture, i.e. only one size category is present (as shown at the first highlighted point at $s_{\nu\mu} = 1$ in Figure 4.1); The second condition corresponds to the idea that each pair of particle populations should have an equal and opposite rate and effect of size segregation. In context of designing the function, meeting the first condition means ensuring $B_{\nu\mu} = 0$ at $s_{\nu\mu} = 1$. As for the second condition, it can also be interpreted as:

$$\mathcal{B}(s_{\nu\mu}) = -\mathcal{B}(1/s_{\nu\mu}) \qquad \text{where } B_{\nu\mu} = \mathcal{B}(s_{\nu\mu}). \tag{4.3}$$

In practice, it makes sense to only use equation 4.1 for $s_{\nu\mu} \geq 1$, and refer to equation 4.3 to define $\mathcal{B}(s_{\nu\mu})$ for $0 \leq s_{\nu\mu} \leq 1$. This decision help avoid the complexity of designing a polynomial meeting all requirements across the entire domain.

The third highlighted point in Figure 4.1 demonstrates one additional assumption we propose: The segregation rate eventually falls back to 0 when the size ratio becomes too large, in this case, at $s_{\nu\mu} = 4$. In agreement with the local maximum segregation rate observation, we anticipate the segregation rate to decline as size ratio go beyond the optimal value. During this process of size ratio increase, the local gaps in-between particles are

expected to become larger compared to the smaller-sized particles, making them easier to fill in the fluctuating void spaces opened up and closed down during shearing motion. However, it becomes harder to lever larger-sized particles upwards because they are sustaining a large portion of the contact network, effectively locking them in place and preventing them from changing positions. Therefore, the ability for the two grains to segregation through kinetic sieving is expected to decline as size ratio grows beyond a certain optimal value.

The reason for us to go one step further and assume $B_{\nu\mu} = 0$ at $s_{\nu\mu} = 4$ is simple: such increased ability of small particles to fill in void spaces as well as the gradual inability of large particle to change position go against fundamental assumptions within the mixture theory framework. As described in Chapter 2, we assume solid volume fraction to be steady and near-uniform across the modelled region, this enables us to only consider solid volume fraction ϕ_ν for each population of grains. Our expectation of granular behaviour as $s_{\nu\mu}$ increases goes against such assumption, threatening the functionality of kinetic sieving mechanism and model. Therefore, we assume $B_{\nu\mu} = 0$ at $s_{\nu\mu} = 4$, indicating a point where the grains effectively interact similar to how dust and boulders interact. It might be really easy for dust to 'trickle down' and collect at the bottom, but it is much more difficult to have dust collectively lift the boulders upwards. In general, it becomes harder to model granular mixture in the form of a continuum as size ratio becomes too large (the average size of void spaces could be on the same level of some populations of small-sized particles). Therefore, we only use the last highlighted point (in this case at $s_{\nu\mu} = 4$) as a reference point 'in far distance' in size ratio axes and avoid actually approaching near it. It should also be noted that the function 4.1 is a manually-selected one, and the size ratio at which point $B_{\nu\mu}$ goes back to 0 can be altered to fit

different scenarios.

Conveniently, the Kurganov-Tadmor semi-discrete scheme (Kurganov and Tadmor, 2000) allows for expansion into groups of convection-diffusion equations to solve (it only implies that the formerly scalar problem is now in vector format), effectively permits the possibility of modelling multiple granular size categories simultaneously. However, the actual process of expanding the kinetic-sieving model is a complex and challenging one. To start with, a list of ϕ values need to be processed and stored at the same time, demanding a change of data structure and rewriting of how each function processes the data. Additionally, as there are more than 2 or 3 populations of particles at play, the complexity of function naturally increases, sometimes making it no-longer possible to adopt simplification measures that were previously in use.

It should be made clear that, our goal of expanding kinetic-sieving model to poly-dispersity is never to just have 4 or 5 types of particles being modelled at once. In fact, we aim to build an algorithm that can perform iterative expansion to involve any specific number n of granular populations. We believe this is the right step to take before incorporating breakage effect into the model although it does pose additional mathematical challenges. In order to ensure reliability, we minimise the amount of prescribed parameters and function settings. Instead, we have made a strong effort to make the expansion process as 'automatic' and flexible to any arbitrary population number as possible.

One good example of complexity of the iterative expansion process is how pairwise functions are properly superimposed when multiple new populations are introduced. As defined in equation 4.4, matrix $\mathbf{P}_n$ stores the pairing relationships among all population members when there are n pop-

ulations being modelled. For a bi-disperse problem, $\mathbf{P}_2$ is conveniently one-dimensional and therefore only require one partial equation to solve (as described in Chapter 2). Due to the pairwise design of stress partition functions, the pairing relationships play a key role in computing numerical values for various convection and diffusion terms in the group of segregation equations. For example, $\mathbf{P}_4$ can be used to guide how pairwise segregation effects should be superimposed to form the net motion relative to bulk. This might seem straightforward or unnecessary when there is only a handful of populations. However, size of the $\mathbf{P}_n$ matrix increases in both its dimensions as new populations are introduced. Therefore, it makes sense to establish an iterative way of constructing $\mathbf{P}_n$ based on $\mathbf{P}_{n-1}$ in an iterative manner as shown in equation 4.5 and equation 4.6.

$$\mathbf{P}_2 = \begin{bmatrix} 1 & 2 \end{bmatrix} \qquad \mathbf{P}_3 = \begin{bmatrix} 1 & 2 & 1 & 3 \\ 2 & 1 & 2 & 3 \end{bmatrix} \qquad \mathbf{P}_4 = \begin{bmatrix} 1 & 2 & 1 & 3 & 1 & 4 \\ 2 & 1 & 2 & 3 & 2 & 4 \\ 3 & 1 & 3 & 2 & 3 & 4 \end{bmatrix} \tag{4.4}$$

$$\mathbf{P}_{n-1} = \begin{bmatrix} 1 & 2 & 1 & 3 & \dots & 1 & n-1 \\ 2 & 1 & 2 & 3 & \dots & 2 & n-1 \\ \vdots & \vdots & & & \ddots & \vdots & \vdots \\ n-2 & 1 & n-2 & 2 & \dots & n-2 & n-1 \end{bmatrix} \tag{4.5}$$

$$\mathbf{P}_n = \begin{bmatrix} 1 & 2 & 1 & 3 & \dots & 1 & n-1 & 1 & n \\ 2 & 1 & 2 & 3 & \dots & 2 & n-1 & 2 & n \\ \vdots & \vdots & & & \ddots & \vdots & \vdots & \vdots & \vdots \\ n-2 & 1 & n-2 & 2 & \dots & n-2 & n-1 & n-2 & n \\ n-1 & 1 & n-1 & 2 & \dots & n-1 & n-2 & n-1 & n \end{bmatrix} \tag{4.6}$$

The pairing matrix $\mathbf{P}_{n-1}$ documents all pairwise relationships between non-identical pairs for $n-1$ populations. As shown in equation 4.5, elements

marked in blue and cyan were introduced to expand $\mathbf{P}_{n-2}$ into $\mathbf{P}_{n-1}$. Here, blue columns describe the relationships going from each other population to the $(n-1)$-th; cyan rows include relationships going from the $n-2$-th population to everyone else. The rule to automatically expand $\mathbf{P}_{n-1}$ into $\mathbf{P}_n$ is in two steps. First, information stored in the blue-colored last columns are transposed to form the orange-colored new row. Second, a new set of last columns colored in red is constructed according to relationships from each other population to the n-th.

$$\mathbf{P}_3 = \begin{bmatrix} 1 & 2 & 1 & 3 \\ 2 & 1 & 2 & 3 \end{bmatrix} \qquad \mathbf{G}_3 = \begin{bmatrix} 1 & 1 & 1 & 2 & 1 & 3 \\ 2 & 1 & 2 & 2 & 2 & 3 \\ 3 & 1 & 3 & 2 & 3 & 3 \end{bmatrix} \tag{4.7}$$

It should be noted that, only a selected subset of all pair-wise relationships are recorded by $\mathbf{P}_n$ matrices. The omitted relationships consist of two groups: self-identical pairs and the replaceable pairs thanks to mass conservation. A side-by-side comparison is shown in Figure 4.7, where $\mathbf{G}_3$ is a generic matrix recording all pair-wise relationships. Due to the non-segregation condition for mono-disperse mixture (i.e. $B_{\nu\nu} = 0$) and structure of Jacobian matrices used in computing flux function derivatives in equation 2.28, it makes sense to omit the diagonal pairs colored in red for higher computing efficiency.

Since volume fraction of the last population can always be calculated with information of the remaining populations based on mass conservation of all populations combined, we can omit the set of equation for one population. Therefore, the line of equation dedicated to the last population is no-longer necessary, and the row regarding to $n-1$-th particle (shown as the blue-colored pairs within $\mathbf{G}_3$ in Figure 4.7) is omitted.

The biggest strength of $\mathbf{P}_n$ matrices compared to generic $\mathbf{G}_n$ formulation is an more optimised and simplified data structure. Quantitatively, only a $(n-1) \times (n-1)$ matrix (instead of a $n \times n$ one) is required to model a n-disperse problem, potentially saving considerable amount of computing power and storage resource for simulations with very high population count.

One of the re-occurring themes of this chapter is the necessity to be able to establish and maintain a correct data structure as the population count increases. Iterative approaches of building the structure for n populations based on that for $n-1$ populations prove powerful and reliable. For example, pairing matrix $\mathbf{P}_n$ provides a clear framework to list or superimpose pairwise relationships. Given an array of n grain sizes, one can easily use $\mathbf{P}_n$ to calculate grain size ratios and hence pairwise segregation rates $B_{\nu\mu}$ based on equation 4.1. Similarly, numerical approximation of flux term values from segregation equation 2.24 include summation of pairwise segregation rate contributions. Iterative expansion can once again help construct appropriate equations based on information of all modelled populations using 4.1 coupled with symbolic calculation and differentiation. It can be said that symbolic calculation slows down computing speed significantly in coding practice, but we do believe it is the right sacrifice to make in order to ensure robustness of the model upon population expansion. Additionally, we want to make sure everything is working as intended during the iterative expansion process, therefore we greatly emphasised on performing all the steps in a way as straight-forward as we can. In future stages, there is a considerable amount of room for further optimizations.

The reason to employ symbolic calculation mostly comes from the need to generate partial derivatives while numerically approximating flux terms within Kurganov-Tadmor scheme for an arbitrary number of size populations. As described earlier, poly-dispersity implies that a vector $\mathbf{\Phi}$ instead

of a scalar ϕ acts as the unknown and hence every flux function takes vectorised format. As defined in equation 2.29, the partial derivative of a function vector $\mathbf{f}(\mathbf{\Phi})$ or $\mathbf{g}(\mathbf{\Phi})$ with respect to $\mathbf{\Phi}$ is therefore their Jacobian matrix $\partial\mathbf{f}/\partial\mathbf{\Phi}$.

So far, the pairing matrix introduced earlier is capable of allocating the correct array of pair-wise coefficients (namely $B_{\nu\mu}\ \forall\phi_\mu \neq \phi_\nu \in \mathbf{\Phi}$) for each element of a flux function $\mathbf{f}(\mathbf{\Phi})$ (namely $\forall\phi_\nu \in \mathbf{\Phi}$). However, constructing the Jacobian matrix involves partially differentiating each element of $\mathbf{f}(\mathbf{\Phi})$ with respect to each element of $\mathbf{\Phi}$. For a particular set-up with a fixed number of size populations (with bi-disperse and tri-disperse being the two simplest cases), it is not hard to manually and analytically prepare these derivatives beforehand and then prescribe the exact formulae of $\partial\mathbf{f}/\partial\mathbf{\Phi}$. However, adding a size population into the system would bring so many challenging changes to the data and computing structure, making this partially-prescribed approach no-longer suitable. Specifically, these challenging changes can take the form of additional terms in each element of $\mathbf{f}(\mathbf{\Phi})$, additional elements in $\mathbf{f}(\mathbf{\Phi})$ and consequently, entirely different formulae for the Jacobian matrix $\partial\mathbf{f}/\partial\mathbf{\Phi}$.

Having pointed out the limitations of partially-prescribed formulations for polydisperse and the need for a truly self-organized approach to assemble the correct formulae (with the biggest challenge being constructing $\partial\mathbf{f}/\partial\mathbf{\Phi}$), it becomes very clear that symbolic calculation has to be incorporated into the automated process. In practice, we use `SymPy` library for the symbolic calculation and differentiation due to its versatility of dealing with multiple symbolic items and compatibility with the rest of our defined functions written in `Python`. In the remaining part of this section, we will go through one poly-disperse example, explain how the Kurganov-Tadmor scheme is carried out with the help of symbolic preparation.

Consider a mixture of five size populations with their sizes defined as $\mathbf{s} = [1, 1.1, 1.2, 1.3, 1.4]$. According to the previously-defined procedure (equation 4.4 - 4.6), we have already constructed function `pairingAdd` to generate the appropriate pairing matrix (source code can be found in Appendix).

$$\mathbf{P}_5 = \begin{bmatrix} 1 & 2 & 1 & 3 & 1 & 4 & 1 & 5 \\ 2 & 1 & 2 & 3 & 2 & 4 & 2 & 5 \\ 3 & 1 & 3 & 2 & 3 & 4 & 3 & 5 \\ 4 & 1 & 4 & 2 & 4 & 3 & 4 & 5 \end{bmatrix} \tag{4.8}$$

Based on the pairing matrix $\mathbf{P}_5$ and the size array $\mathbf{s} = [1, 1.1, 1.2, 1.3, 1.4]$, function `updateSizeRatios` is created to calculated pairwise grain-size ratios according to $\mathbf{P}_5$. The output is a 4×4 matrix $\mathbf{R}_5$ containing these ratios as shown in equation 4.9. For example, the $(2, 3)$-th element of $\mathbf{R}_5$ is guided by the 3rd pairwise relationship stored on the 2nd row of $\mathbf{P}_5$, i.e. $[2, 4]$. This piece of information is then fed into `updateSizeRatios` and guides it to calculate size ratio of the 2nd over the 4th particle population, i.e. $1.1/1.3 = 0.846154$.

$$\mathbf{R}_5 = \begin{bmatrix} 0.909091 & 0.833333 & 0.769231 & 0.714286 \\ 1.1 & 0.916667 & 0.846154 & 0.785714 \\ 1.2 & 1.09091 & 0.923077 & 0.857143 \\ 1.3 & 1.18182 & 1.08333 & 0.928571 \end{bmatrix} \tag{4.9}$$

After enforcing the size-ratio dependence of $B_{\nu\mu}$ as defined in equation 4.1, an array of the pair-wise segregation rates can be computed via function `sizeRatioRelation` and `updateSegBs`. In this example, the resulting

matrix $\mathbf{B}_5$ consisting of all $B_{\nu\mu}$ takes the following form.

$$\mathbf{B}_5 = \begin{bmatrix} 0.1682 & 0.3136 & 0.4374 & 0.5408 \\ -0.1682 & 0.153869 & 0.288805 & 0.40571 \\ -0.3136 & -0.153869 & 0.141782 & 0.267593 \\ -0.4374 & -0.288805 & -0.141782 & 0.131452 \end{bmatrix} \tag{4.10}$$

As described in section (2.5), Kurganov-Tadmor scheme revolves around achieving numerical approximations for each convection and diffusion flux function. In this case, we need to supply the semi-discrete scheme with the correct procedure working towards the numerical flux for vectorized version of the non-dimensionalized segregation equation 2.24.

As explained earlier in the section, symbolic calculation via `SymPy` is proposed to keep track of polynomial structures of the flux functions and their derivatives with respect to solution vector $\partial\mathbf{f}/\partial\mathbf{\Phi}$. Similar to the pairing matrix $\mathbf{P}_n$, symbolic representation of these polynomials can effectively provide a correct mapping of mathematical structure of these crucial terms, guiding the numerical computation process while eliminating the need of prescribing the formulae manually for different configurations of poly-dispersed problems.

It is worth-noting that, while symbolic calculation does objectively draw more computing power and slows down the process, it does retain the critical advantage of flexibility. Furthermore, symbolic differentiation does not need to be carried out for each and every time-step. In fact, we propose generating and storing the polynomial structure for flux terms and their derivatives before the time loop, and referring to these symbolic polynomials for computing formulae for each time-step, minimizing the computing demand especially for simulation consisting of large number of populations.

Back in context of the example problem involving five size populations, the symbolic polynomials for convection flux in z-direction is calculated via `SymPy`. Combining pairwise segregation rate information from $\mathbf{B}_5$ (4.10) and pairwise relationship information from $\mathbf{P}_5$ (4.8), we are now able to assemble a matrix $\mathcal{P}_5$ storing symbolic polynomials representing the vectorized flux function $\mathbf{f}(\mathbf{\Phi})$ as shown in equation 4.11.

$$\mathcal{P}_5 = \frac{1}{\mathcal{C}} \begin{bmatrix} B_{12}\phi_1\phi_2 + B_{13}\phi_1\phi_3 + B_{14}\phi_1\phi_4 + B_{15}\phi_1\phi_5 \\ B_{21}\phi_2\phi_1 + B_{23}\phi_2\phi_3 + B_{24}\phi_2\phi_4 + B_{25}\phi_2\phi_5 \\ B_{31}\phi_3\phi_1 + B_{32}\phi_3\phi_2 + B_{34}\phi_3\phi_4 + B_{35}\phi_3\phi_5 \\ B_{41}\phi_4\phi_1 + B_{42}\phi_4\phi_2 + B_{43}\phi_4\phi_3 + B_{45}\phi_4\phi_5 \end{bmatrix} \tag{4.11}$$

It can be seen that, $\mathcal{P}_5$ consists of four rows, each dedicated to the net segregation influences experienced by one size population. Thanks to mass conservation, we can omit the last size population and express its volume fraction in terms of volume fraction of the remaining populations. In this case, we omit the row dedicated to ϕ_5 and utilize $\phi_5 = 1 - \phi_1 - \phi_2 - \phi_3 - \phi_4$ for the last pair of each row in $\mathcal{P}_5$. In fact, the pairing matrix P_5 itself was specifically designed with this intention in mind, so that the last 'binned' population would only occur in fixed position in each row. For example, the first row of $\mathcal{P}_5$ takes the form as shown below.

```
0.0841*phi1*phi2 + 0.1568*phi1*phi3 + 0.2187*phi1*phi4
                 + 0.2704*phi1*(-phi1-phi2-phi3-phi4+1)
```
(4.12)

where `phi1`, `phi2`, `phi3` and `phi4` are symbolic objects defined to represent the first four size populations. To further obtain Jacobian matrix of the flux function $\mathbf{f}(\mathbf{\Phi})$, we employ `symPy.diff` to differentiate the symbolic polynomial with respect to each symbolic object. For example, the first row shown in equation 4.12 can be differentiated with respect to `phi1`,

`phi2`, `phi3` and `phi4` and form the first row of the Jacobian matrix $\partial\mathbf{f}/\partial\mathbf{\Phi}$ as shown in equation 4.13.

$$\begin{bmatrix} \texttt{-0.5408*phi1-0.1863*phi2-0.1136*phi3-0.0517*phi4+0.2704} \\ \texttt{-0.1863*phi1} \\ \texttt{-0.1136*phi1} \\ \texttt{-0.0517*phi1} \end{bmatrix}^T \tag{4.13}$$

After cross-checking, it can be seen this expression reaches agreement with the correct formula 4.14. In fact, it is mathematically equivalent to the first row of Jacobian matrix. This example clearly demonstrates that `SymPy` can indeed function as intended for the purpose of generating and storing symbolic polynomial before the time loop starts, providing a formulae of substitution for constantly evolving elements of the solution vector $\mathbf{\Phi}$.

$$\begin{bmatrix} -2B_{15}\phi_1 + (B_{12} - B_{15})\phi_2 + (B_{13} - B_{15})\phi_3 + (B_{14} - B_{15})\phi_4 + B_{15} \\ (B_{12} - B_{15})\phi_1 \\ (B_{13} - B_{15})\phi_1 \\ (B_{14} - B_{15})\phi_1 \end{bmatrix}^T \tag{4.14}$$

Being capable of building a correct formula for Jacobian matrix independent of the current ϕ_ν values, we use the symbolic polynomials $\mathcal{P}_5$ and its derivative $\mathcal{J}_5$ to compute intermediate values defined in equations 2.29, 2.31 and 2.32 without pre-placed information about these polynomials, making it a fully-automated iterative process. It is worth-noting that, while calculating the normal local maximum speed a^z as defined in equations 2.29, `NumPy.linalg.eig` is employed to calculate eigenvalues of the Jacobian matrices.

Being able to compute all intermediate values, temporal numerical approx-

imations H^x and H^z (also in vectorized format for poly-dispersed case) of flux terms can be obtained, leading to a group of time-dependent ODEs dedicated to each population similar to equation 2.33. Since all the right-hand-sided terms are approximated numerically, a 2nd-order Runge-Kunta method (Heun's method) is used as a time-stepper to obtain an updated value of solution vector $\mathbf{\Phi}$ in the next time-step. This entire process can be repeated to generate evolution of all solid volume fractions for all size populations, allowing us to simulate granular behaviour subject to granular segregation coupled with cyclic loading.

4.3 Validation against bi-disperse and tri-disperse results

Having walked through how the iterative expansion process is able to formulate and numerically compute solid volume fraction vector ϕ via constructing the correct symbolic structure for pairing relationships, flux functions and their derivatives with respect to volume fraction of some size population ϕ^{ν}, it is important to first test and validate whether the iterative expansion method is doing everything correctly.

In this section, we document two validation cases, with the first one being against the classic bi-disperse kinetic-sieving model and the second one being against tri-disperse case. The first and most important objective is to make sure that iterative expansion process is adjusting the model and data structure in the correct way, producing reasonable simulation results aligning with classic models which require pre-placed formulae. Additionally, we hope to demonstrate that: although symbolic calculation takes a toll on computing speed and can potentially be optimized for a specific case, its

advantage of avoiding accumulating 'numerical noise' is indeed noticeable upon comparing simulation results.

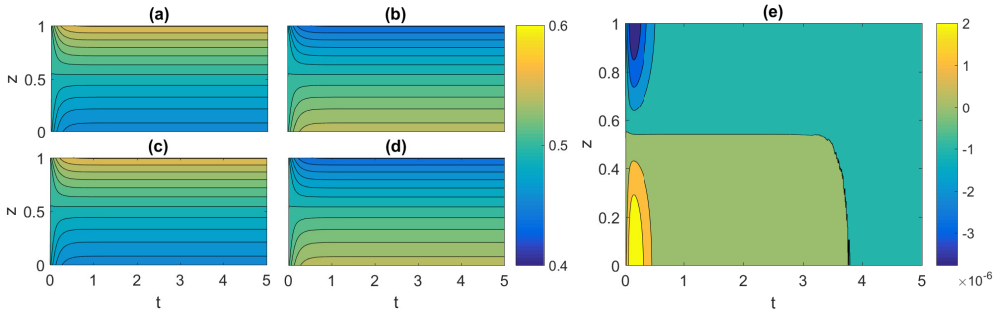


Figure 4.2: Side-by-side comparison of bi-disperse simulations from iterative expansion method and classic kinetic-sieving method. All parameters, with $\mathcal{P}_0 = 0.5$, $\mathcal{C} = 2$ and $\mathcal{D} = 0.2$, are kept the same. The same size ratio, pairwise segregation rate, cyclic-loading velocity profile, initial condition and pressure-dependent diffusion flux term are in place for both simulations. (a),(b): Large and small particle evolution profile from iterative expansion simulation; (c),(d): Large and small particle evolution profile from iterative expansion simulation; (e): Difference of large particle population solid volume fraction ϕ_L between the two simulations, the error is in magnitude of 10^{-6}, mostly taking place during segregation process, then declines afterwards.

For a bi-disperse set-up, the size ratio input vector is set as `[1.5,1.0]`, resulting in a size ratio of $s_{12} = 1.5$ (due to the order of size prescription) and hence a pair-wise segregation rate $B_{1,2} = 0.625$ defined by equation 4.1. The pressure-dependence diffusion flux as proposed in Chapter 2 Section 3 is kept in place with non-dimensionalized confining pressure $\mathcal{P}_0 = 0.5$. After prescribing all spatial and scaling parameters with the same value and using the same homogeneous initial condition $\phi_1 = 0.5 \ \forall z \in [0, 1]$, two simulations have been carried out with the only difference being between the usage of classic kinetic-sieving approach and that of iterative expansion approach. As shown in Figure 4.2, the results qualitatively and quantitatively agree to the magnitude of 10^{-6}, with most of the disagreements taking place when size segregation and population spatial migration are most active. After the populations have segregated to enriched regions and

convection flux has fallen back to a lower level, such numerical disagreement between the two retreats back to the order of 10^{-15}, suggesting a well-reached agreement for quasi-steady state between the two approaches.

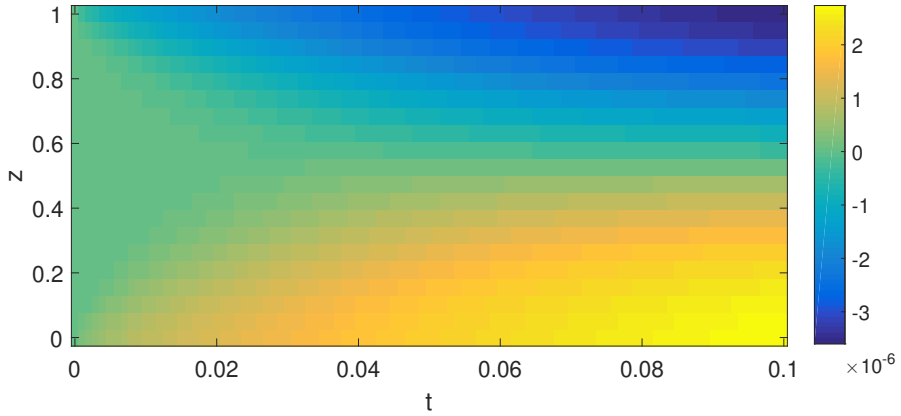


Figure 4.3: Heat-map of the accumulation of numerical disagreement between two bi-disperse simulations during the first 100 time-steps. Errors appear to emerge from the top and bottom boundaries and propagate towards the centre with time, eventually stabilizing with a magnitude of order 10^{-6}.

In addition to Figure 4.2(e), the heat-map shown in Figure 4.3 provides a closer inspection of how exactly the errors occur and accumulate for the first 100 time-steps out of a total of 5000. Numerical disagreements can clearly be observed to first emerge at top and bottom boundary layers, gradually increasing and accumulating into enriched central regions. The general trend of how these difference values evolve bears strong resemblance to the size segregation behaviour itself. However, no clear explanation has been found about the cause of these disagreements. Initially, it was suspected to be an 'hysteresis echo' caused by delayed enforcement of convection terms, but this is found not to be the case after data comparison: numerical changes of ϕ between adjacent time-steps are often beyond the magnitude of 10^{-6}. However, the general pattern does suggest the possibility of an unexpected or overlooked influence from or within the modelled convection in z-direction.

One possible explanation is the usage of symbolic formulae in iterative expansion approach. For example, when setting large particle to have solid volume fraction `phi = 0.8` for classic approach and equivalently `phis = [0.8]`, the latter approach manages to provide the most accurate result, whereas the former's output suffers from numerical discrepancy as demonstrated in equation 4.15. It is possible that these small numerical errors create accumulative noise in the solution for each time-step, further affecting the initial condition for future time-steps. As defined in equation 2.29, 2.31, magnitude of vertical numerical fluxes are influenced by local maximum speed a_z, whose process of computing is actually the biggest motivation behind developing and using the iterative expansion approach. In classic kinetic sieving models, it is convenient to prescribe the already-differentiated formulae to the solver alongside with the simplified case of $\rho(\mathbf{\Phi})$ (a clear example is shown in equation 2.30). In iterative expansion approach, symbolic calculation is performed to construct the Jacobian matrices and to find their eigenvalues. Similar to the case of flux function, the steps of symbolic calculation instead of numerical computation might contribute to the local numerical disagreements.

```
fluxFunZ(phi) = 0.04999999999999999
fluxFunZp(phis) = 0.05
```
(4.15)

Similar comparison of results are also drawn for a tri-disperse scenario. As shown in Figure 4.4, numerical disagreements also appear to emerge and eventually fade away in time with the peak magnitude once again being 10^{-6}. No final conclusion can yet be drawn about why the error start at top and bottom boundaries by the time of writing this version of the thesis. More efforts will be made to uncover more understanding of this phenomenon, especially for multi-disperse scenarios.

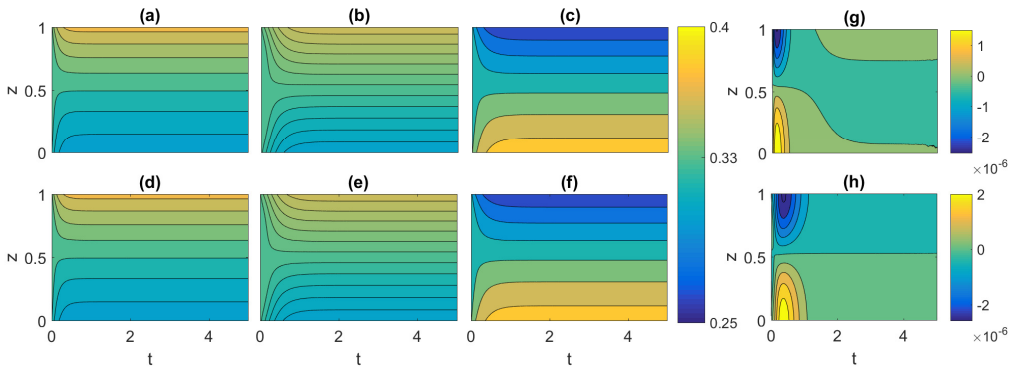


Figure 4.4: Side-by-side comparison of tri-disperse simulations from iterative expansion method and classic kinetic-sieving method. All parameters, with $\mathcal{P}_0 = 0.5$, $\mathcal{C} = 2$ and $\mathcal{D} = 0.2$, are kept the same. The same size ratio, pairwise segregation rate, cyclic-loading velocity profile, initial condition and pressure-dependent diffusion flux term are in place for both simulations. (a),(b),(c): Large, median and small particle evolution profile from iterative expansion simulation; (d),(e),(f): Large, median and small particle evolution profile from iterative expansion simulation; (g),(h): Difference of large and small particle population solid volume fraction ϕ_L and ϕ_S between the two simulations, the error is in magnitude of 10^{-6}, reaching their peaks during segregation process, then declines afterwards.

In this section, side-by-side comparisons against classic kinetic-sieving methods have been made for bi-disperse and tri-disperse cases. The objective is to validate the reliability and examine the exact performance of the more flexible iterative expansion approach. In conclusion, good quantitative agreements have been reached for both cases, showing clearly that the iterative expansion process can indeed produce the correct symbolic data storing as well as computing structures for bi-disperse and tri-disperse problems. Since no human configuration beyond initial conditions was required to switch from bi-disperse to tri-disperse, it is reasonable to believe that the iterative expansion approach can continue producing correct results for granular mixtures with an arbitrarily-larger population pool.

4.4 Test design scenarios based on size ratio distribution

In the previous section, validation against classic bi-disperse and tri-disperse kinetic-sieving models were carried out resulting in quantitative agreements up to the magnitude of 10^{-6}, suggesting that poly-dispersity problem could be safely constructed and numerically solved via the iterative expansion approach. These encouraging results open up vast room of possibility, especially in terms of accurately incorporating large number of granular size populations in the modelled mixture, bringing the continuum model one more step closer towards modelling geo-physical granular flows with entire spectrum's of complex particle size distributions. In fact, we will reveal in next chapter that poly-dispersity is also a very crucial prerequisite to simulating particle breakage effect.

To demonstrate capability of the iterative expansion poly-dispersity model as well as to explore how pair-wise size segregation behaves for large number of populations, we designed a series of 10-population simulations featuring different types of size distribution. Similar to the two examples covered in previous section, the poly-dispersity model reads the prescribed size array and generate the appropriate data structure for storing and calculating data using the iterative expansion approach. Due to the much larger population pool, high performance computers were employed to run these simulations, cutting the running time from over a week on average desktop to around three days.

Precisely as the population number grows larger, the significance of a well-thought-out test design become more and more highlighted. After all, too much time and computing power would go wasted if a 100-population

simulation turns out to be ill-organized. This section is dedicated to explain the motivation and thinking process behind these test designs.

A key defining choice of the classic kinetic-sieving model is the usage of pair-wise type stress partition function, shown in equation 2.15. By sacrificing the complexity of additional local information, it gains the advantage of mathematical simplicity and robustness: segregation effects now consist of pair-wise components dedicated to each pair of dissimilar size populations, making it possible to estimate as well as experimentally measure and calibrate the segregation rates $B_{\nu\mu}$. Furthermore, the pair-wise formulation allows a relationship between segregation rates and size ratios (such as the one defined by 4.1 and shown in Figure 4.1) to be established. This offers convenience both to potentially measuring the curve from experiment (measuring segregation time scale of a bi-disperse mixture should be much easier than that of a mixture containing 10 size populations) as well as designing a 10-population test: it is much easier to anchor and visualize the segregation rate distribution on a curve and determine whether it works as intended. In fact, such a direct link between segregation rate and size ratio will become even more valuable for simulations involving larger number of populations. For example, prescribing 3 pair-wise segregation rates for a tri-disperse simulation already needs to divided into multiple cases (Gray and Ancey, 2011), it can be predicted that manually prescribing segregation rates for an ambitious large-population-number test would be an impossible task without a rule of guidance, which is exactly what equation 2.15 provides.

Distribution test design	Particle size array
Small grain rich distribution	`[1, linspace(1.4,1.6,num=9)]`
Large grain rich distribution	`[1, linspace(2.6,2.9,num=9)]`
Peak spectrum distribution	`[1, linspace(1.6,2.4,num=9)]`
Broad spectrum distribution	`[1, linspace(1.1,3.4,num=9)]`
'Realistic' distribution	`[1, linspace(1.2,1.8,num=9)]`

Table 4.1: Size distribution details of each test design.

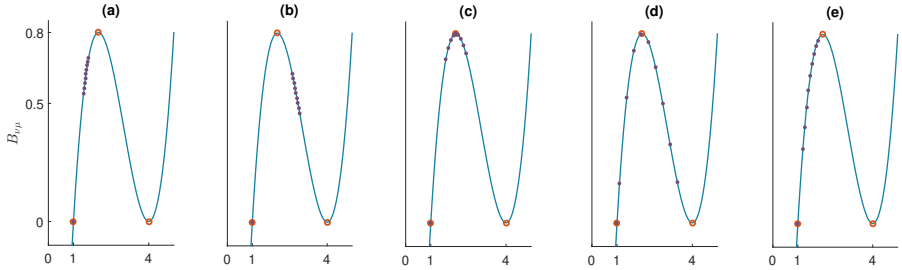


Figure 4.5: Illustration of the five test design scenarios for one poly-disperse mixture consisting of 10 size populations. The orange circled points mark the critical size ratios where non-segregation and peak-segregation rates are reached, the purple dots correspond to size ratios of larger grains against the reference grain category. (a) Small grain rich distribution; (b) Large grain rich distribution; (c) Peak spectrum distribution; (d) Broad spectrum distribution; (e) 'Realistic' distribution.

As demonstrated in Table 4.1 and Figure 4.5, 10-population simulations have been run for five different test designs which were designed in order to capture characteristics of different types of granular mixture. In order to provide reference, the smallest-sized population within each test design is set to have `size = 1.0`, and the remaining size populations are mapped onto equation 4.1 using their size ratios against the smallest population. This decision helps keep all size ratios bigger than or equal to 1, and at the same time provide a rough idea of how the spread of segregation rates look like, this is shown in details by Figure 4.6. All the dissimilar pair-wise size ratios are mapped onto the curve, demonstrating that the range of reference size ratios envelope that of remaining pair-wise ratios.

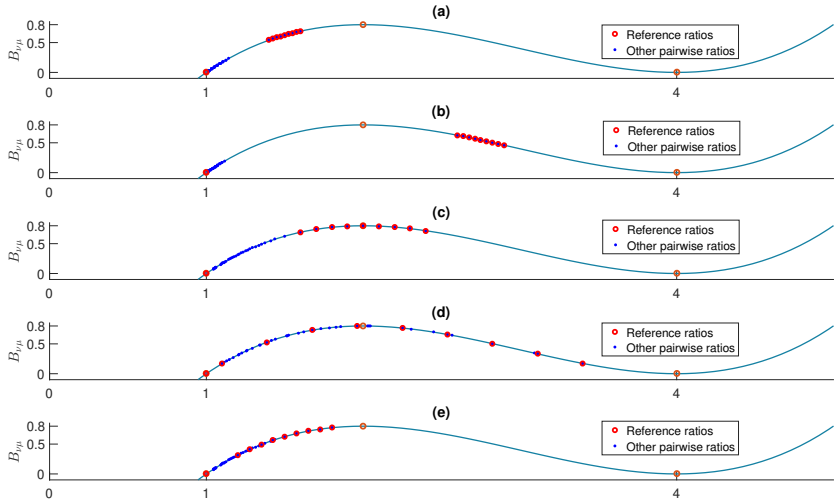


Figure 4.6: Distribution of reference size ratios as well as pairwise size ratios between each pair of dissimilar size populations. The particle size distribution is the 'broad spectrum' as defined in Figure 4.5.

The first three cases (small grain rich, large grain rich and peak spectrum) focus on how a packet of similar-sized populations would react to a smaller 'dust' population, differentiating in the gap of grain size as well as position of the packet on the segregation rate function. The broad spectrum case aims to model the opposite: the grains sizes are spread out very thinly, across both side of the optimal segregation ratio. We hope to observe what could happen if the granular mixture has a well-spaced spectrum of size distribution. The last case ('realistic') attempts to link further towards geo-physical granular mixture and to foreshadow the possibility of particle breakage. Within a cyclic-loading environment, breakage might occur in different fashions depending on grain material. It is possible to have breakage taking place mostly in the form of crushing and wearing off edges from larger particles. In this case, the grains break and form a much smaller collection of dust population. Therefore, the last case have a well-spaced-out packet of grain sizes as well as a much smaller population (in this case the reference population due to its smallest size).

4.5 Results and discussion

In previous section, we explained the rationale behind proposing five specific grain size distributions, mostly targeting diversity and anticipating to see variation of simulation results among different configurations. Utilizing these test designs, a series of 10-population simulations have been performed.

Test Case	**Diffusion Rate $\mathcal{D}$**	**Confining Pressure $\mathcal{P}_0$**
(a)	0.1	0.1
(b)	0.1	0.2
(c)	0.1	0.5
(d)	0.2	0.1
(e)	0.5	0.1

Table 4.2: Specification of test case parameters for each size distribution.

In total, 25 poly-disperse simulations were run using the high performance computers (HPC), with variation in three categories: particle size distribution, diffusion parameter $\mathcal{D}$ and non-dimensional confining pressure $\mathcal{P}_0$. Each simulation has the same spatial limits and discretisation and is set to run 2400 time-steps representing a non-dimensionalised simulation time-span $t \in [0, 2.4]$, starting from the same homogeneous initial condition where each population has solid volume fraction $\phi_\nu = 0.1\ \forall\nu$. We have demonstrated in Figures 3.7 and 3.8 that inter-particle drag coefficient $\mathcal{C}$ has a clear impact on time-scale of all flux terms for all size populations. However, the diffusion flux terms contributed by $\mathcal{D}$ and $\mathcal{P}_0$ seem to undergo more complex evolution in space and time. Also considering the fact that each batch of simulation costs around $3-5$ days to finish computing in parallel on HPC, the decision was made to plan 5 simulations for each size-distribution case, varying in $\mathcal{D}$ and $\mathcal{P}_0$. The exact setting of each case is

documented in Table 4.2. Test case (a) serves as a reference point with the smallest flux term amplitudes, while (b)(c) and (d)(e) focus on variations in diffusion rate $\mathcal{D}$ and confining pressure $\mathcal{P}_0$ respectively. The full list of poly-disperse figures can be seen in Appendix Chapter D.

Figures D.1 - D.25 document the exact simulation results corresponding to each test design and parameter configuration. After analysing the simulation results, two observations can be made. Firstly, size distribution seem to have a strong impact on shape and even in one case position of the high-concentration region (Most noticeable for dust populations). Secondly, both $\mathcal{D}$ and $\mathcal{P}_0$ have a clear and qualitatively similar impact on segregation time-scale.

Among the five types of size distributions, the first two (small grain rich and large grain rich) involve a close group of size populations far away from the reference population (in this case also set as the dust population). Therefore, it was expected to see the group of 9 non-reference populations to have rather limited degree of segregation among themselves compared to how they segregate against the dust population. Figures D.1 - D.10 certainly agrees with this prediction. For both of the packet-ed test designs, the dust population undergo a quick and clear segregation process and accumulate at the bottom. After roughly $0.5-0.7$ seconds, the concentration region of dust population seem to reach a stable value from this point.

In contrast, simulation conducted using peak spectrum distribution appear to have a much more fluctuating outline for dust population. Furthermore, due to the non-reference populations no-longer being on the same side of the optimal size ratio point, they seem to segregate much more rapidly compared with the two packet-ed distributions. Furthermore, since the smallest non-reference population now has a size closer to that of refer-

ence population, clear high concentration regions from more than one size populations can be observed to form at the bottom. However, we find the fluctuating outline of dust population regime most outstanding. One possible way of explaining this behaviour could be that, fluctuation of the pressure dependent diffusion flux term is causing a non-steady influence during the most active period of segregation process. This hypothesis is supported by the eventual stabilization of dust region regimes. However, we are not yet able to explain why only the dust population appear to posses such fluctuating pattern and the others do not.

As shown in Figures D.16 - D.20, changing the test design to broad spectrum distribution seems to drastically change the behaviour for smaller-sized populations. Clear concentration regions emerge for population (h) and (i), but the dust population seems to be unexpectedly migrating and accumulating near top boundary. We can not yet determine the exact reason behind such behaviour, but it is fair to assume that size distribution and its implicated pair-wise relationships must play a crucial role. In hindsight, Figure 4.5 seems to provide some insights especially considering the distribution of all pair-wise ratios. For packet-ed distributions, the non-reference pair-wise ratios are far apart from the reference ratios, making the reference relationships take a leading role in both segregation and pressure-dependent diffusion flux terms. In contrast, the broad distribution involves a much more inter-twined combination of reference and non-reference ratios, possibly making the pressure-dependent diffusion flux components overrule the pair-wise segregation ones, leading towards a reversely-segregated dust population. Although there is not a well-validated conclusive explanation yet, it makes sense to assume that broad spectrum distribution serves as a counter-example in terms of test design. In future poly-disperse simulations, the choice of avoiding overly spread-out reference ratios and conse-

quently non-reference ratios inter-twining with reference ratios should be seriously considered. For the 'realistic' distribution, similar fluctuation for dust population is once again evident in simulation results, also hinting the importance of imposing appropriate size ratio relationships.

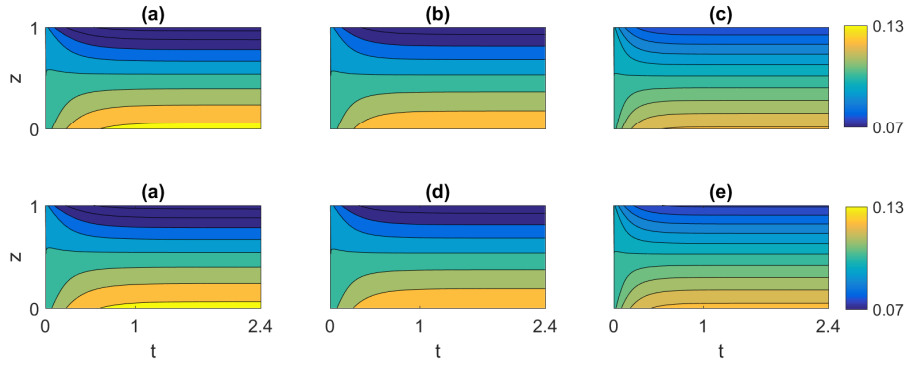


Figure 4.7: Comparison of simulated dust population results corresponding to different $\mathcal{D}$ and $\mathcal{P}_0$ settings under small grain rich distribution: (a) $\mathcal{D} = 0.1$, $\mathcal{P}_0 = 0.1$; (b) $\mathcal{D} = 0.1$, $\mathcal{P}_0 = 0.2$; (c) $\mathcal{D} = 0.1$, $\mathcal{P}_0 = 0.5$; (d) $\mathcal{D} = 0.2$, $\mathcal{P}_0 = 0.1$; (e) $\mathcal{D} = 0.5$, $\mathcal{P}_0 = 0.1$.

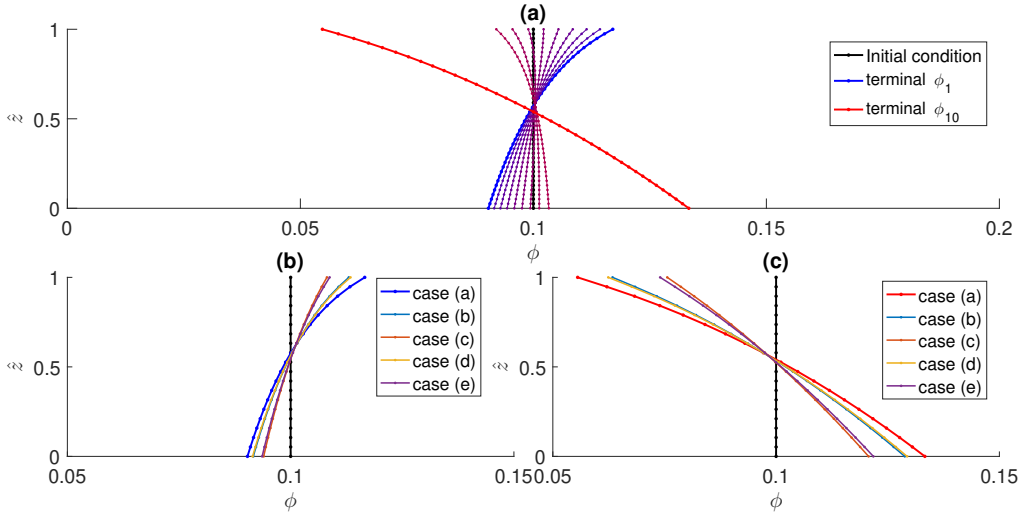


Figure 4.8: Illustrations of solid volume fraction ϕ against height $\hat{z}$: (a) Comparison of terminal ϕ data of all 10 populations at $\hat{t} = 4$ against their initial condition $\phi_0 = 0.1$ at $\hat{t} = 0$ under small grain rich distribution. Here ϕ_1 corresponds to the largest population and ϕ_10 corresponds to the smallest; (b) Comparison of terminal ϕ_1 for the five $\mathcal{D}$ and $\mathcal{P}_0$ settings under small grain rich distribution; (c) Comparison of terminal ϕ_10 for the five $\mathcal{D}$ and $\mathcal{P}_0$ settings under small grain rich distribution.

On the other hand, confining pressure appears to produce visible influence towards time-scale and extent of segregation outcome. As shown in Figure

4.7, even for the relatively stabler small grain rich distribution, increase of $\mathcal{P}_0$ leads to a very similar outcome as increase of $\mathcal{D}$. However, it should be noted that the effect of confining pressure is depth-dependent and therefore the pressure-dependent flux increases in magnitude towards bottom of the modelled region. More quantitative comparison can be seen in Figure 4.8, where the terminal volume fraction values are compared against the uniform initial condition ($\phi_0 = 0.1$ at $\hat{t} = 0$). As expected, the smallest-sized population undergoes the most significant changes in volume fraction distribution during the simulation. Additionally, similarity between feedbacks caused by increasing $\mathcal{D}$ and $\mathcal{P}_0$ is demonstrated in an even more quantitative manner. The depth-dependence of pressure-dependent flux can also be seen when being compared with effects of increasing $\mathcal{D}$ (i.e. curves corresponding to cases (b) and (c) are closer to the initial condition, suggesting longer segregation timescales at bottom).

Finally, we would like to report another simulation feature that we are not yet able to explain. From the definition of classic kinetic-sieving model to iterative expansion poly-disperse model, the simulation results seem to be sensitive to order of size prescribing order. Frankly, it is by no means a designed or desired feature, considerable amount of time has been devoted trying to uncover the root cause of this unexpected order-sensitivity, we can only say with confidence that the coding of pressure-dependent diffusion is possibly carrying over some discrepancies for large number of size populations. However, validations against classic bi-disperse and tri-disperse data have both reached quantitative agreements. Therefore, we are assuming this is a poly-disperse-specific problem. More efforts are being devoted into investigating this issue at the time of submission, it is possible that we have a better understanding of the root cause soon.

Chapter 5

Incorporating breakage

In this chapter, we start by addressing various technical challenges of combining breakage effect with a continuum model. A triangular breakage model is proposed to provide preliminary predictions of how breakage influence the grain size distribution. Afterwards, the idea of solid volume fraction exchange is introduced with some further explorations of using gain and loss functions in conservation equations. In the last section, polydisperse simulation results ran with simple breakage are presented and discussed.

Contents

5.1 Difficulty of simulating breakage effect within a continuum model

Although various continuum models have been developed to describe granular segregation in chute flows, most of the well-known breakage models come from a 'discrete element' point of view. While size segregation can be modelled as a macroscopic mass migration of certain populations of grains relative to the bulk mixture, particle breakage are generally seen as a rather microscopic phenomenon occurring to individual particles. In addition, proposed criteria of breakage often requires local information in the form of contact stress (McDowell et al., 2013) or force (Hanley et al., 2015) surrounding the particle of interest. Therefore, numerous studies on granular breakage employ discrete element modelling (DEM) methods (Golchert et al., 2004; Antonyuk et al., 2006; Wang and Yan, 2013; Zhang et al., 2020), varying in targeted breakage mechanisms, assumptions of breakage criteria and representation of crushing outcomes.

For example, McDowell et al. (2013) proposed to utilize octahedral shear stress to facilitate multiple contacts and Weibull distribution of particle strength to enforce size effects while Hanley et al. (2015) suggested that 'fragment particles' should be created upon breakage and comminution limit ought to be introduced. In addition, it is widely-accepted that DEM methods of crushing can be categorized in mainly two types: agglomerate models where uncrushable principle particles are joint together by breakable bonds and replacement models where parent particles would be replaced by daughter particles when breakage takes place (McDowell et al., 2013). Further reviews on various breakage mechanisms can be found in recent publications (McDowell et al., 2013; de Bono and McDowell, 2016; Yu, 2021).

When it comes to accommodating particle breakage, population balance modelling (Ramkrishna, 2000) has also seen considerable development and applications in various fields (Ramkrishna and Singh, 2014). Modelling via population balance equations has one similar feature as the mixture theory inspired model: changes in internal coordinate (i.e. aggregation (Jeldres et al., 2018) and breakage (Chakraborty and Ramkrishna, 2011)) can be phenomenologically described in terms of number density. However, it remains significant to supply sufficient criteria information on breakage events(Chakraborty and Ramkrishna, 2011). We acknowledge that population balance modelling is a successful and closely-relevant approach that has shown great potential in simulating computational fluid dynamics (Marchisio and Fox, 2005). However, we decided to focus on the kinetic sieving model in this thesis due to limited time of the project. In future, it would be interesting to see if population balance models can be introduced to model granular breakage occuring under cylic loading.

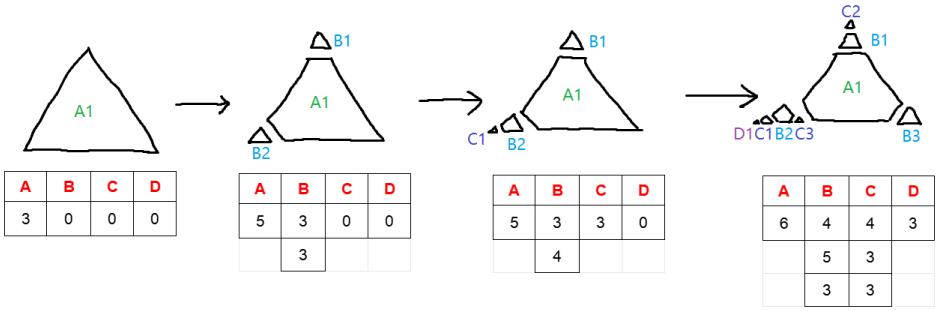
In general, it is realized that a shortage of particle breakage models compatible with continuum segregation models is present, and it remains a serious challenge to impose plausible breakage criteria for continuum models due to the lack of local stress/force information. Further investigation on DEM methods may be required for deeper understanding in particle breakage, because the majority of effort in this project has been committed on constructing a polydisperse time-dependent continuum segregation model. And the process of implementing such breakage mechanism within a continuum model will be covered in the following sections.

5.2 'Toy model' for visualising size distribution

Before embarking on the task of incorporating breakage ideas, it is worthwhile to gain some understanding of how the grain size distribution would evolve as breakage takes place. In all classic and polydisperse simulations covered so far, total solid volume fraction ϕ_ν remains conserved for each size population ν. However, as will be discussed in later sections, breakage can be visualized and interpreted as a re-distribution process of solid volume fraction, making them no longer conserved throughout simulated time-frame. Therefore, in order to understand what kind of change in grain size distribution we should anticipate, a 'toy model' for breakage was developed in early stages of this project.

In this simple model, we consider all objects to be in triangular shape, with each of the three vertices free to break away into smaller daughter triangles. As shown in Figure 5.1, creation of each daughter triangle adds one new edge to the mother triangle. When breakage has occurred on every vertex, the mother triangle will become a hexagon and is considered no-longer eligible for further breakage. Breakage progress of each object is documented via recording the number of edges. For example, when mother triangle A1 breaks two vertices and form daughter triangles B1 and B2, edge count of A1 increases from 3 to 5, and two new object entries of 3 edge count are created for new B-class triangles B1 and B2. When one vertex of B2 breaks into a smaller class of triangle, triangle C1 is also represented by a new object entry of 3 in C-class. When A1 eventually have all three vertices broken away, it reaches edge count 6 and is no-longer considered breakable. At each time-step, we loop over all non-hexagon triangles with edge count less than 6 and apply a probability p_{break} for event of breakage.

Figure 5.2 shows the cumulative density function of triangles assigned to different size classes after various time-steps of breakage constant breakage probability $p_{break} = 0.8$. The simulation starts with 10 triangles of size 1 (this can be viewed as edge length), and every daughter triangle takes 1/3 the size of its mother triangle.



A	B	C	D
3	0	0	0

A	B	C	D
5	3	0	0
	3		

A	B	C	D
5	3	3	0
	4		

A	B	C	D
6	4	4	3
	5	3	
	3	3	

Figure 5.1: Illustration of how the triangle breakage events are recorded via counting the available edges.

One of the key motives of using discrete triangle object to experiment breakage is to explore effects of possible crushing mechanisms. For example, a pseudo comminution point is implemented: The triangular particles are listed in descending order of size, and those below the 10% quantile is forbidden from breakage. In hindsight, this way of enforcing a comminution point at each time-step lacks a great deal of accuracy: the polygons are categorized into a discrete range of size populations, hence the 10% quantile threshold would only filter them into broad-stroke classes, and if a size class takes a big part of the total population, the entire group would be forbidden from breaking even if the quantiled range only cover a small percentage of this group. But since this simulation was done at an early stage of the project, the objective was just to see what would happen if features of geo-physical breakage were taken into consideration. Therefore, the algorithm was allowed to loop over all the size columns and check if the size of this column is smaller than 10% quantile threshold. If so, this class is treated as hexagons for this time-step only.

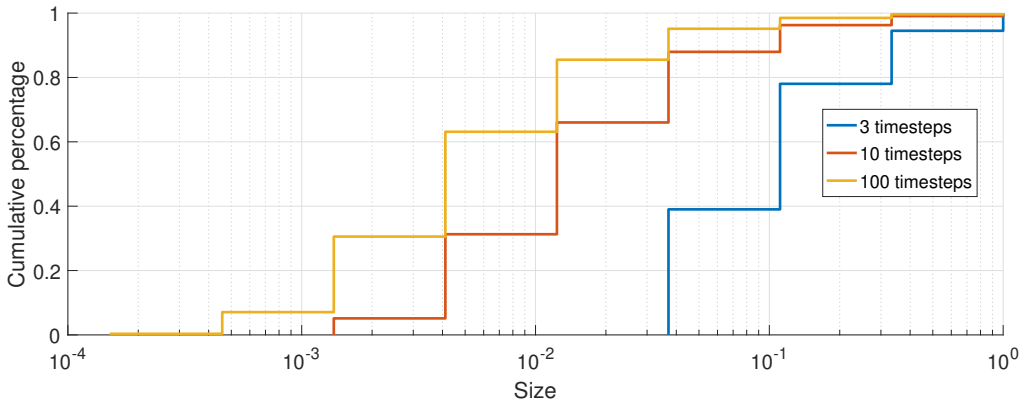


Figure 5.2: Cumulative distribution of size populations after 3, 10 and 100 time-steps of triangle breakage simulation with constant breakage probability $p_{break} = 0.8$.

It can be seen that, the cumulative distribution of size classes changes very rapidly during the first few time-steps, then gradually stabilizes at later time. Although smaller classes continue to be created as the iteration progresses, it is clear their magnitude grow drastically more insignificant compared to the 10 original triangles (now stable hexagons). This observation suggests the simplification of crushing via only having one designated population to represent fine dust compared to all regular-sized populations. Besides, the constant breakage probability does not necessarily apply to real geo-physical granular mixture. It is fair to assume that breaking off an edge from a regular-sized cobblestone is considerably easier than for a tiny grain of sand. Therefore, results of the 'toy model' motivated us to limit how far particles can break and to focus on having only one terminal size population dedicated to fine dust. This idea was later on incorporated in poly-disperse test designs (as shown in Table 4.4) in preparation for breakage implementation.

5.3 Simplest breakage idea: Solid volume fraction exchange

If we picture particle breakage within a polydisperse continuum framework, then there must be exchanges of solid volume among all particle types, hence solid volume fractions are redistributed whenever breakage takes place. The simplest way to achieve such effect within the polydisperse model is to force a certain amount of volume fraction redistribution in-between convection-diffusion time-steps.

As shown in Figure 5.3, breakage in the form of solid volume fraction redistribution is implemented within a quad-disperse simulation for example. During the $4s$ length of simulated time, three redistribution time-steps are inserted early on (at $t = 0.05s$, $t = 0.1s$ and $t = 0.2s$) to effectively break down 1%, 5% and 10% of the three largest-sized particles into the smallest-sized one. Even with just three inserted time-steps of redistribution, a considerable amount of feedback is visible in the solid volume fraction evolution. This once again demonstrates how much more exciting granular interactions and feed-backs can be captured once breakage is incorporated into the poly-disperse time-dependent continuum model.

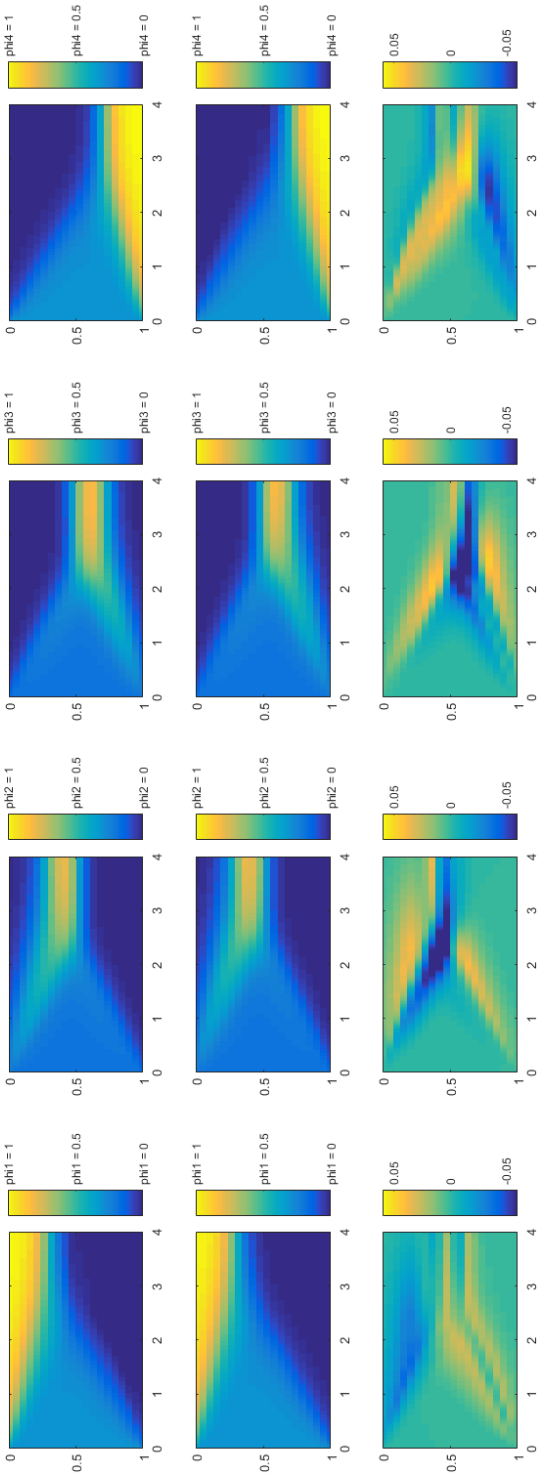


Figure 5.3: Segregation results comparison between with and without ϕ redistribution. The first row of sub-figures correspond to the ϕ evolution within a time-dependent quad-disperse simulation with particle sizes being $1.4, 1.0, 0.6$ and 0.4. The second row of sub-figures correspond to the same simulation but with the largest three species of particle being manually broken into large particles at $t = 0.05s$, $t = 0.1s$ and $t = 0.2s$. The third row of sub-figures are the difference in ϕ evolution due to the introduction of breakage.

5.4 Gain and loss functions

In order to see breakage effects having more detailed and realistic feedbacks to the granular flow, it makes sense to start by modifying the mass balance equation to describe this rearrangement of volume fractions across different phases. Recall that assuming uniform intrinsic density for all types of particles (i.e. $\rho^\nu = \phi^\nu \rho$ where $\rho = const$), the mass conservation equation satisfied by each phase is:

$$\frac{\partial \phi^\nu}{\partial t} + \nabla \cdot (\phi^\nu \mathbf{u}^\nu) = 0. \tag{5.1}$$

In earlier derivations of the kinetic-sieving segregation theory, information of the bulk velocity $\mathbf{u}$ and relative velocity $\mathbf{u}^\nu - \mathbf{u}$ for some phase ν are extensively explored and utilized, it makes sense to rearrange the general segregation equation 2.24 to involve the bulk and relative velocity profiles:

$$\frac{\partial \phi^\nu}{\partial t} + \nabla \cdot (\phi^\nu \mathbf{u}) + \nabla \cdot (\phi(\mathbf{u}^\nu - \mathbf{u})) = 0. \tag{5.2}$$

Applying the in-compressibility condition $\nabla \cdot \mathbf{u} = 0$ and using the chain rule, one can see that

$$\nabla \cdot (\phi^\nu \mathbf{u}) = (\nabla \phi^\nu) \cdot \mathbf{u}. \tag{5.3}$$

The shallow water assumption made in the original kinetic-sieving chute flow model (Gray and Thornton, 2005) suggests that the bulk velocity profile can be approximated as $\mathbf{u} = (u, 0, 0)$ in three-dimensional space, i.e. only moving in the down-slope direction. Therefore equation 5.3 can be further simplified to

$$\nabla \cdot (\phi^\nu \mathbf{u}) = (\nabla \phi^\nu) \cdot \mathbf{u} = u \frac{\partial \phi^\nu}{\partial x}. \tag{5.4}$$

The steady-state assumption is also typically implemented in the chute model, i.e. $\partial\phi^\nu/\partial t = 0$. As a result of these assumptions, equation 5.2 now becomes

$$u\frac{\partial\phi^\nu}{\partial x} + \frac{\partial}{\partial z}(\phi^\nu(w^\nu - w)) = 0. \tag{5.5}$$

A simple bi-disperse case of breakage could be: one type of large particle breaks into a smaller type of particle. Considering mass conservation during breakage and constant density for all particles, volume is conserved during breakage process. In other words, the amount of solid volume fraction lost by large particles and the amount of solid volume fraction gained by small particles should equate in magnitude. In other words, if ϕ_t^ν is denoted for terminal solid volume fraction of phase ν after breakage, then $\Delta\phi^l + \Delta\phi^s = 0$ holds for $\Delta\phi^l = \phi_t^l - \phi^l$ and $\Delta\phi^s = \phi_t^s - \phi^s$. For simplicity, the positive one $\Delta\phi^s$ is more preferable to be used, i.e. $\phi_t^l = \phi^l - \Delta\phi^s$ and $\phi_t^s = \phi^s + \Delta\phi^s$. Note that in this case, the relative velocity $w^\nu - w$ is prescribed as in equation 5.5 and the mass balance equation 5.1 eventually becomes

$$\begin{aligned} u\frac{\partial\phi^l}{\partial x} + \frac{\partial}{\partial z}(S_{ls}\phi^l\phi^s) &= \frac{\partial}{\partial z}(D_r\frac{\partial\phi^l}{\partial z}); \\ u\frac{\partial\phi^s}{\partial x} + \frac{\partial}{\partial z}(-S_{ls}\phi^l\phi^s) &= \frac{\partial}{\partial z}(D_r\frac{\partial\phi^s}{\partial z}). \end{aligned} \tag{5.6}$$

If the change in partial velocity profiles $\mathbf{u}^l$ and $\mathbf{u}^s$ due to breakage is neglected, then one can argue that the mixture still satisfies the same mass conservation relation after breakage

$$\begin{aligned} \frac{\partial\phi_t^l}{\partial t} + \nabla\cdot(\phi_t^l\mathbf{u}^l) &= 0; \\ \frac{\partial\phi_t^s}{\partial t} + \nabla\cdot(\phi_t^s\mathbf{u}^s) &= 0. \end{aligned} \tag{5.7}$$

Under the same assumptions made earlier, this group of equations can be expanded into

$$\begin{aligned}\frac{\partial \phi^l}{\partial t} + \nabla \cdot (\phi^l \mathbf{u}^l) + \Big(- \frac{\partial}{\partial t}\Delta\phi^s - \nabla \cdot (\Delta\phi^s \mathbf{u}^l)\Big) = 0 \\ \frac{\partial \phi^s}{\partial t} + \nabla \cdot (\phi^s \mathbf{u}^s) + \Big(\frac{\partial}{\partial t}\Delta\phi^s + \nabla \cdot (\Delta\phi^s \mathbf{u}^s)\Big) = 0.\end{aligned} \tag{5.8}$$

Considering $\Delta\phi^\nu$ to be a function of all phases, the ideal goal is to rearrange this system of equations into a general form:

$$\frac{\partial \phi^\nu}{\partial t} + \nabla \cdot (\phi^\nu \mathbf{u}^\nu) + Gain(\phi^\mu, \forall \mu \in S) - Loss(\phi^\mu, \forall \mu \in S) = 0, \tag{5.9}$$

where $Gain(\phi^\mu, \forall \mu \in S) = 0$ for $\nu = l$, $Loss(\phi^\mu, \forall \mu \in S)$ for $\nu = s$. With an appropriate formula of $\Delta\phi^s$, it may be possible to simulate breakage effects in the continuum framework. However, this approach poses serious challenges in mainly two ways.

First, particle breakage is mostly discussed in a discrete-element view where particles are modelled individually. This is not compatible with our continuum framework, which can lead to problems in crucial parts of the formulae. For example, one would expect the $Gain(\phi^\mu, \forall \mu \in S)$ and $Loss(\phi^\mu, \forall \mu \in S)$ terms to satisfy an overall 'conservation relation' so that the total mass or volume fraction of the system remains constant. However, it is difficult to construct such relations in detail within the continuum framework due to the complexity of the terms. Therefore some works remain to be done in order to explore further what can be done to ensure volume/mass conservation in breakage terms.

The second challenge comes from the complexity of the terms: even a simple formula of $\Delta\phi^s$ can lead to complex terms in the ODE, making it difficult to achieve analytical or numerical solutions. Here is an example

of such escalating complexity: In a steady-state problem, it is difficult to incorporate a separate time-dependency for $\Delta\phi^s$ alone, for the original 2D problem only has derivatives in x and z directions. As a first guess, $\Delta\phi^s = 0.01x^2$ is proposed: this would eliminate the time-derivatives, and thus only leaving the $\nabla\cdot(\Delta\phi^s\mathbf{u}^\nu)$ term to be simplified for $\nu = s, l$. Similar to the procedure in equations 5.3 - 5.6, this term can be expanded as:

$$\begin{aligned}\nabla\cdot(\Delta\phi^s\mathbf{u}^l) =& u\frac{\partial(\Delta\phi^s)}{\partial z} + \frac{\partial}{\partial z}[\Delta\phi^s(S_{ls}\phi^s - \frac{D_r}{\phi^l}\frac{\partial\phi^l}{\partial z})],\\ \nabla\cdot(\Delta\phi^s\mathbf{u}^s) =& u\frac{\partial(\Delta\phi^s)}{\partial z} + \frac{\partial}{\partial z}[\Delta\phi^s(-S_{ls}\phi^l - \frac{D_r}{\phi^s}\frac{\partial\phi^s}{\partial z})].\end{aligned} \tag{5.10}$$

Note that the $\frac{D_r}{\phi^l}$ and $\frac{D_r}{\phi^s}$ terms can no-longer be cancelled out like it was in derivation process of the segregation-remixing equation, which makes the steady-state solver unable to yield numerical results. Admittedly, a more suitable choice of $\Delta\phi^s$ could remedy this situation, but this example has already shown the difficulties in incorporating breakage in the steady-state problem in an already simplified bi-disperse problem. In reality, it is almost impossible to maintain bi-dispersity when breakage takes place and the granular particles should be allowed to break into at least several other size-classes. In addition, the dependency of ($\Delta\phi^s$ is not fully explored yet in this section. In order for the breakage model to achieve enough resemblance to reality, a lot of experimental or theoretical improvements remains to be done, thus leaving another area of huge potentials and challenges.

Another fundamental risk of such approach is that breakage of the granular material would suggest against the flow being in steady state. Therefore, the exploration of implementing breakage in steady-state is paused at this point, and we suggest a more general time-dependent problem for bi-disperse and poly-disperse granular mixture for future work.

5.5 Preliminary results and discussion

In Chapter 4, a way of constructing poly-disperse size segregation simulations using iterative expansion approach is proposed and validated against classic kinetic sieving models in bi -disperse and tri-disperse set-ups. Having briefly touched on the idea of representing breakage in the form of solid volume fraction re-distribution with clear in simulation results (as shown in Figure 5.3), we explored the possibility and feasibility of introducing loss and gain functions into the mass balance equations. The goal is to see whether we can simulate the inter-size-population re-distribution as introduced in previous section in the case of building from a steady-state problem. It becomes reasonably clear that difficulties such as an appropriate choice of loss/ gain functions as well as further complications of granular flow conditions make it challenging and difficult to actually involve breakage as a part of the numerical solving process within the time-dependent problem. Therefore, based on the earlier encouraging quad-disperse results (Figure 5.3), we propose to instead explore the possible effects of just solid volume fraction re-distribution in between time-steps for poly-disperse problems consisting of more than 4 populations, this time with the emphasis focused on capturing characteristics of 'crushing and wearing into dust' phenomenon.

One of the key differences between 'breaking into two parts' and 'wearing into dust' is the size ratio between mother and daughter particles. As demonstrated in section 5.2 (especially in Figure 5.2), it seems appropriate to have one or two dedicated populations that is much smaller in size than all other ones to represent the dust worn off from them. There are mainly two reasons behind this decision: Firstly, further breakage of the dust population can be less likely to happen while also having a smaller

impact on normal-sized particles. Secondly, it makes sense to assume the newly-created dust particles to be similar-sized for all normal particles under cyclic-loading because their size magnitudes are usually relatively far away in size axis. Hence, for a 10-population simulation, we need to have the majority of all size populations possess a much larger size than the dust population. To accommodate such diversity in size and size ratios, a slightly-adjusted version of segregation rate function is proposed as below.

$$B_{\nu\mu} = \alpha \left((s_{\nu\mu} - 20)^3 + 19 \, (s_{\nu\mu} - 20)^2 \right) \tag{5.11}$$

where $\alpha = 0.001$. As shown in Figure 5.4, sizes for majority of populations are between 10 and 19 whereas size of the dust population is defined to be 1. After comparing segregation rates corresponding to reference and remaining ratios, it is evident that segregation rates are prescribed to be remarkably higher against the dust population (as marked by reference ratios) compared to other combinations (as marked by other pairwise ratios), with its magnitude gradually declining as size ratio increases. Breakage is incorporated into the model via solid volume re-distribution in

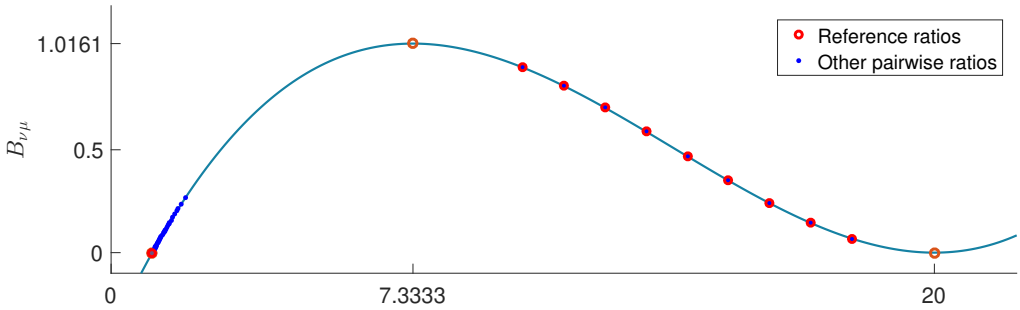


Figure 5.4: Illustration of size ratio distribution and segregation rate function for the 10-population simulation with size distribution set as `[1,10,11,12,13,14,15,16,17,18,19]`.

between time-steps with period $T_{break} = 0.04s$. The 10-population polydisperse mixture is simulated for $0.2s$ in total, resulting in breakage event at $t = 0.04s, t = 0.08s, t = 0.12s$ and $t = 0.16s$. At every breakage event, we

insert one additional procedure dedicated to re-distributing ϕ_ν before numerically solving for that time-step. The exact extent of the re-distribution is determined by a prescription vector `breakPlan` as defined in equation 5.12 containing the exact proportion of each non-reference population being transferred to reference duct population representing crushing into dust. The prescription vector `breakPlan` is highly customizable and compatible with potential modifications such as additional depth and size dependence. For this demonstration case, we keep a simple setting and prescribe a 5% breakage rate for each non-reference population.

```
breakPlan = [0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05,0.05]
```

(5.12)

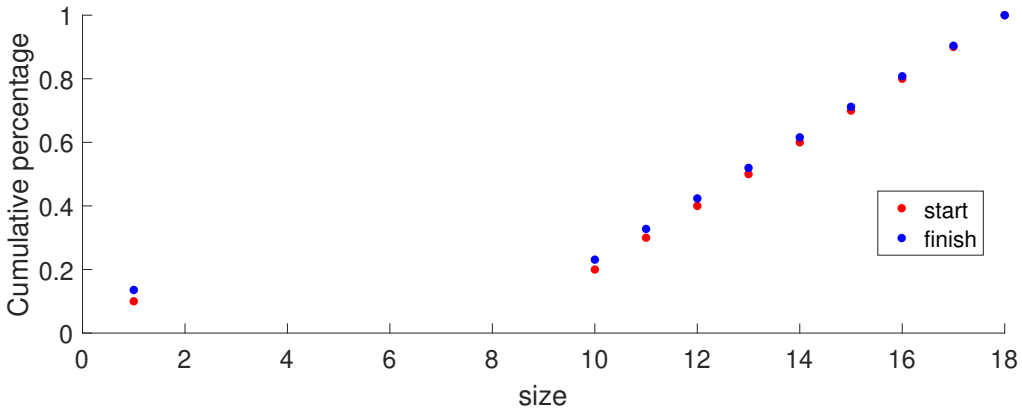


Figure 5.5: Terminal cumulative distribution of size populations of the 10-population polydisperse simulation with mass exchange mechanism functioning in between time-step.

As shown in simulation history results (Figure 5.6) and the cumulative distribution of size populations at start and finish of simulation (Figure 5.5), quantitative impacts are observed in simulation results, and the cumulative distribution of size populations shift towards a more 'dust-heavy' direction as expected. In Sub-figure 5.6(j), the dust population receive noticeable additional increases across the entire depth at breakage events, resulting a

realistic-looking accumulation of dust at the bottom of the simulated space. In fact, this high-concentration region of dust seems to create a feed-back to other populations, causing the smaller non-reference populations to accumulate at depths higher above bottom.

In conclusion, we have successfully incorporated simple solid-volume re-distribution functions into the poly-disperse simulations and are once again able to numerically solve the problem using Kurganov-Tadmor scheme. In simulation results, quantitative feed-backs caused by breakage events are evident, justifying solid-volume re-distribution as a starting approach towards incorporating complex breakage phenomenon with size segregation. We also discussed about a more complex and self-contained approach of involving breakage into the mass conservation equations themselves using gain and loss functions. Due to the limited time of this project, we are not able to pursue further in this direction, but a great amount of opportunities seem to be open for both approaches. It is foreseeable that, given more detailed dependence settings and designed breakage functions, having breakage effect actively modelled in a continuum poly-disperse size segregation model is no-longer one impossible task. Based on simulation results up to the point of thesis submission, a lot of breakage features appear to be captured correctly under even the simplest settings, encouraging us to be optimistic about the outlook and potential of these methods.

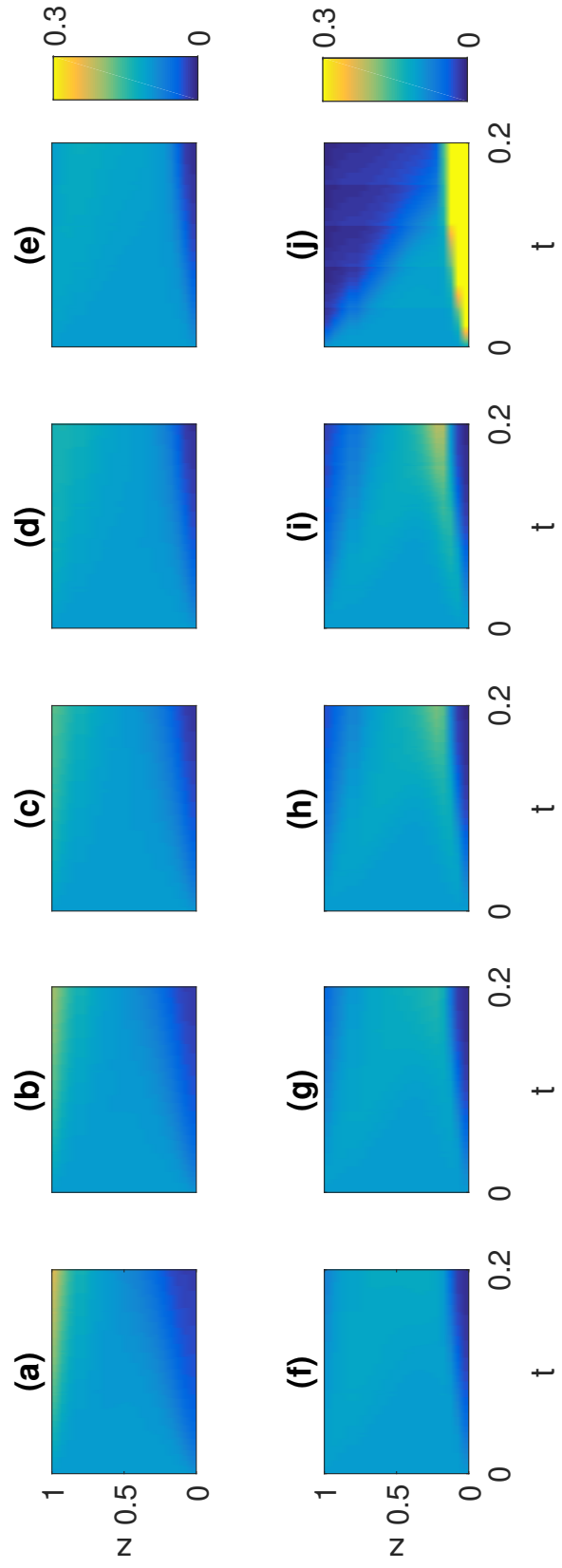


Figure 5.6: Simulation results of a polydisperse simulation coupled with mass exchange mechanism in between time-steps involving 10 size populations, sub-figures (a)-(j) correspond to the grain populations in descending order of size.

Chapter 6

Conclusions

6.1 From chute flow model to shear box model with breakage: highlights of the thesis

In this thesis, we walked through the process of building a highly-customised poly-disperse granular segregation continuum model based on classic kinetic sieving chute flow model for the purpose of capturing more features of granular behaviour inside a cyclic-shearing environment. With the ultimate goal being incorporating effective breakage into the model, we devised an iterative expansion mechanism allowing the model to self-expand from bi-disperse to arbitrarily-poly-disperse. In this conclusion chapter, we start by going through the highlights of each chapter in broad strokes before discussing in details the potential for future improvements and follow-up works.

In Chapter 1, we started by a brief walk-through of various granular segregation mechanisms and explained why it is important to gain further understandings of size segregation phenomenon in dry sense granular flows.

Specifically, we put a strong emphasis on simulating size segregation within poly-disperse mixtures under cyclic-loading scenario motivated by real-life problems such as underwater wind turbine foundations loading their surrounding soil particles and oil transport pipes cyclically shearing sand grains in deserts.

In Chapter 2, we went through the theoretical framework and mathematical procedures of the classic kinetic sieving model, originally proposed to model gravity-driven chute flows. After acknowledging the need for necessary departure to the classic model driven by different assumptions and set-ups, we implemented time dependence and normal pressure dependence into the model and conducted comparison with the classic model.

During non-dimensionalisation section, we introduced cyclic shearing period as the new time scale, leading to a different set of scaling parameters from the classic model. A brief overview of how the Kurganov-Tadmor semi-discrete central difference numerical scheme is employed to approximate groups of convection-diffusion PDEs as time-dependent ODEs for each time-step is also conducted. At this point, initial modifications to classic kinetic sieving model have been implemented with an appropriate numerical solver introduced.

In Chapter 3, we started by acknowledging the need for further modifications to the model in order to better capture cyclic shear features. After selecting two-dimensional simulation coupled with periodic boundary conditions for modelling central sections of the shear box and briefly describing how solutions are handled and stored, we performed validation of the model against results from a similar parallel shear study (van der Vaart et al., 2015). Agreements of the simulation results indicate that our model agrees with the classic kinetic-sieving model for non-chute-flow scenarios, which

further encourages implementations of modifications based on specific modelling scenarios. Therefore, we recognised time-evolving velocity profile as a desired characteristics in modelling cyclic shearing motion driven by top plate.

After going through the list of desired features we anticipate for the profile, two attempts were made to incorporate time-dependent velocity profile. First, we proposed a discrete sequence of hyperbolic profiles inspired by experimental measurements of a bottom-driven annular shear box study (Golick and Daniels, 2009; May et al., 2010). The sequenced profiles indeed enabled us to simulate initialisation and reversal events within cyclic shearing environments. However, it has many limitations such as inconvenience of configuring surface velocity magnitude and potential disconnect between different groups of profiles. Additionally, the parameters used to adjust profile behaviour are not directly linked to non-dimensionalising scales or typical experimental measurements.

Recognising the room for improvement, we proposed using periodic functions to control the surface velocity magnitude and shape parameter. In this way, we arrived at a more physically anchored control over changing velocity profiles. We also experimented with enforcing numerical shear rate dependence for segregation rate based on the notion that shear rate within the granular material should affect how actively granular material segregate in size. Indeed, we were able to achieve clear feed-backs to bi-disperse two-dimensional simulation results, with or without the enforced shear rate dependence. The sensitivity of velocity profiles suggest the need for more purpose-driven and case-specific profiles in order to achieve the most optimal agreement with real-life scenarios. We concluded this chapter by showcasing the simulation results of the bi-disperse simulation as well as a brief analysis of effects onto segregation behaviour by key parameters.

In Chapter 4, we revealed that the modifications including pressure dependence and time-dependent velocity profiles were all introduced in preparation for achieving poly-dispersity, i.e. being able to model multiple granular populations simultaneously in the model. Traditionally, expanding a bi-disperse problem into a tri-disperse problem requires major overhaul of function definitions and data structure rework. We propose a noval iterative expansion approach to expand any n-population problem into a $(n+1)$-population one.

We began by visualising how pairwise segregation relationships could be generated and maintained within a poly-disperse model and the unique advantage of doing so. Motivated by the pairwise relationships, we designed pairing matrices $\mathbf{P}_n$ as a blueprint for storing segregation rate, pairwise segregation flux terms, pressure-dependent diffusion flux terms and various numerical values used by the Kurganov-Tadmor scheme. Due to the mass-conservation assumption and nature of self-segregation, we made changes to pairing matrices so that only a $(n-1) \times (n-1)$ matrix is required for a n-population problem, which would save considerable computing power for highly poly-disperse simulations.

Afterwards, we briefly showcased the general process of how critical data are generated and computed within the iterative expansion structure. With the help of symbolic calculation and relationship-storing matrices, the poly-disperse model only requires a short list of prescription parameters: size data for all populations and surface velocity magnitude. The choice of these two parameters is motivated by the ease of experimental measurement. Additionally, most of the modifications and settings are designed with flexibility and robustness in mind. For example, the function between segregation rate $B_{\nu\mu}$ and size ratio $s_{\nu\mu}$ can be freely adjusted to achieve better fitting towards a certain experiment or validation scenario, and the time-evolving

velocity profile can also be replaced with experimentally-measured data.

Naturally, it makes sense to validate whether the model is producing correct results under the new iterative expansion approach. We conducted qualitative and quantitative comparison of two-dimensional simulations for bi-disperse and tri-disperse problems. We compared the solutions computed using iterative expansion model with those computed using traditional bi-disperse and tri-disperse models. For both scenarios, solutions generated by two models agree to the order of 10^{-6} with the biggest disagreements happening during the most active stages of segregation and then converges to much lower values. We are not yet crystal clear about the source of these temporal disagreements, but the validation results are certainly encouraging enough for us to keep carrying out poly-disperse simulations using iterative expansion procedure.

In the last section of Chapter 4, test design scenarios for poly-disperse simulations are discussed in details. Specifically, we are interested in how size ratio distribution might potentially impact the outcome of the simulation. Consequently, we proposed five test distributions in hope to capture a diverse spectrum of granular size compositions, and presented 10-population simulations for these distributions. Additionally, we designed a group of five test cases to compare the effect of diffusion rate $\mathcal{D}$ and confining pressure $\mathcal{P}_0$. After comparison among initial and terminal solid volume distributions for all populations, noticeable patterns revealing the desired characteristics of pressure-dependent flux terms can be observed.

In Chapter 5, we started by recognising the technical difficulty of incorporating breakage, an inherently discrete behaviour in a continuum model. A brief literature review on particle breakage DEMs and population balance method were presented. Ultimately, we expressed that further investiga-

tion on DEM and population balance methods may prove beneficial to our simulation, but we made the conscious decision to keep exploring possibilities within the continuum model set-up because of time constraint of the project and the closest relevance to our approach.

Before documenting the implementation details of breakage ideas, we used a triangular breakage 'toy model' to help visualise the concept of representing particle breakage via size distribution, and ultimately re-distribution of solid volume fractions. After briefly going through preliminary findings of the test, we concluded that it may be wise to limit how far particles can break and focus on having only one terminal size population to account for generated dust.

In the following chapter, we experimented with solid volume fraction re-distribution events taking place in-between simulation time-steps for a quad-disperse problem as a preliminary test for possible breakage-like behaviour. Noticeable feed-backs to the solution can be observed, even with only three inserted breakage events, demonstrating the feasibility of the simplest breakage mechanism.

Shortly afterwards, we also attempted a more self-contained approach to model breakage: this time in the form of gain and loss functions. After careful derivations, we concluded that it may be possible to develop one such functional mechanism of breakage and have it directly tied with the conservation equations. However, it probably demands a longer list of assumptions and may depend on specific settings of future experiments or model scenarios. Therefore, we acknowledge the addition layers of challenge of this approach, but we also believe this can be a very promising direction for future works.

Finally, we experimented the functional solid volume fraction re-distribution

mechanism on a 10-population poly-disperse simulation. After showing breakage can be modelled in the form of solid volume fraction re-distribution with clear feed-back created in simulation result, we are pleased to see that the poly-disperse model does indeed show promise in modelling complex geo-physical flows with a much higher level of detail in terms of size diversity and crushing phenomena.

6.2 Potential for future improvements

As described in the previous section, there is a vast room for customization when it comes to prescribed profiles and functions. More state-of-the-art findings and methodologies could also directly or indirectly provide insights for better ways of imposing stress-partition functions, segregation rate functions or velocity profiles.

Unfortunately, it became very difficult to conduct experimental works during the Covid-19 pandemic. As stated in Chapter 3, it would be hugely beneficial if a case-specific top-driven cyclic shearing experiment could be conducted. These results would provide valuable insights for velocity profile and list of assumptions.

The flexibility of modifications and prescription parameters can be especially advantageous when it comes to fitting the model configuration towards experimental results. Therefore, we believe it is a step in the right direction to conduct laboratory experiments with a non-trivial amount of size variations and possibly with crush-able particles. Data collected from these real-life experiments could provide huge guidance and feed-backs for adjusting and improving the poly-disperse model.

Additionally, further testing of the poly-disperse simulations is also crucial to obtain further and firmer understanding of the assumptions, behaviour and limitations of our model. As pointed out in Chapter 4 and 5, there are certainly times where we aren't able to entirely explain every unexpected behaviour of the solutions. Therefore, we believe there is still a considerable room for further assumption checking and programming optimisation.

Bibliography

Antonyuk, S., Khanal, M., Tomas, J., Heinrich, S., and Mörl, L. (2006). Impact breakage of spherical granules: experimental study and dem simulation. *Chemical Engineering and Processing: Process Intensification*, 45(10):838–856.

Bae, C.-J., Ramachandran, A., and Halloran, J. W. (2018). Quantifying particle segregation in sequential layers fabricated by additive manufacturing. *Journal of the European Ceramic Society*, 38(11):4082–4088.

Bartelt, P. and McArdell, B. W. (2009). Granulometric investigations of snow avalanches. *Journal of Glaciology*, 55(193):829–833.

Brandao, R. J., Lima, R. M., Santos, R. L., Duarte, C. R., and Barrozo, M. A. (2020a). Experimental study and dem analysis of granular segregation in a rotating drum. *Powder Technology*, 364:1–12.

Brandao, R. J., Lima, R. M., Santos, R. L., Duarte, C. R., and Barrozo, M. A. (2020b). Experimental study and dem analysis of granular segregation in a rotating drum. *Powder Technology*, 364:1–12.

Breu, A. P., Ensner, H.-M., Kruelle, C. A., and Rehberg, I. (2003). Reversing the brazil-nut effect: competition between percolation and condensation. *Physical review letters*, 90(1):014302.

Brewster, R., Grest, G. S., Landry, J. W., and Levine, A. J. (2005). Plug

flow and the breakdown of bagnold scaling in cohesive granular flows. *Physical Review E*, 72(6):061301.

Chakraborty, J. and Ramkrishna, D. (2011). Population balance modeling of environment dependent breakage: role of granular viscosity, density and compaction. model formulation and similarity analysis. *Industrial & engineering chemistry research*, 50(23):13116–13128.

Combarros, M., Feise, H., Zetzener, H., and Kwade, A. (2014). Segregation of particulate solids: Experiments and dem simulations. *Particuology*, 12:25–32.

de Bono, J. and McDowell, G. (2016). Particle breakage criteria in discrete-element modelling. *Géotechnique*, 66(12):1014–1027.

Forterre, Y. and Pouliquen, O. (2008). Flows of dense granular media. *Annual Review of Fluid Mechanics*, 40:1–24.

Gajjar, P. and Gray, J. (2014). Asymmetric flux models for particle-size segregation in granular avalanches. *Journal of Fluid Mechanics*, 757:297–329.

Golchert, D., Moreno, R., Ghadiri, M., and Litster, J. (2004). Effect of granule morphology on breakage behaviour during compression. *Powder Technology*, 143:84–96.

Golick, L. A. and Daniels, K. E. (2009). Mixing and segregation rates in sheared granular materials. *Physical Review E*, 80(4):042301.

Gray, J. and Ancey, C. (2011). Multi-component particle-size segregation in shallow granular avalanches. *Journal of Fluid Mechanics*, 678:535–588.

Gray, J. and Thornton, A. (2005). A theory for particle size segregation in shallow granular free-surface flows. *Proceedings of the Royal Society*

A: Mathematical, Physical and Engineering Sciences, 461(2057):1447–1473.

Gray, J. M. N. T. (2018). Particle segregation in dense granular flows. *Annual Review of Fluid Mechanics*, 50:407–433.

Hanley, K. J., O'Sullivan, C., and Huang, X. (2015). Particle-scale mechanics of sand crushing in compression and shearing using dem. *Soils and Foundations*, 55(5):1100–1112.

Hong, D. C., Quinn, P. V., and Luding, S. (2001). Reverse brazil nut problem: competition between percolation and condensation. *Physical Review Letters*, 86(15):3423.

Issler, D., Jenkins, J. T., and McElwaine, J. N. (2018). Comments on avalanche flow models based on the concept of random kinetic energy. *Journal of Glaciology*, 64(243):148–164.

Jeldres, R. I., Fawell, P. D., and Florio, B. J. (2018). Population balance modelling to describe the particle aggregation process: A review. *Powder technology*, 326:190–207.

Jenkins, J. T. and Berzi, D. (2012). Kinetic theory applied to inclined flows. *Granular Matter*, 14:79–84.

Johnson, C., Kokelaar, B., Iverson, R. M., Logan, M., LaHusen, R., and Gray, J. (2012). Grain-size segregation and levee formation in geophysical mass flows. *Journal of Geophysical Research: Earth Surface*, 117(F1).

Jomelli, V. and Bertran, P. (2001). Wet snow avalanche deposits in the french alps: structure and sedimentology. *Geografiska Annaler: Series A, Physical Geography*, 83(1-2):15–28.

Ketterhagen, W. R., Curtis, J. S., Wassgren, C. R., Kong, A., Narayan, P. J., and Hancock, B. C. (2007). Granular segregation in discharging cylindrical hoppers: a discrete element and experimental study. *Chemical Engineering Science*, 62(22):6423–6439.

Kurganov, A. and Tadmor, E. (2000). New high-resolution semi-discrete central schemes for hamilton–jacobi equations. *Journal of Computational Physics*, 160(2):720–742.

Leadbeater, T. W., Parker, D. J., and Gargiuli, J. (2012). Positron imaging systems for studying particulate, granular and multiphase flows. *Particuology*, 10(2):146–153.

Lo, C., Bolton, M., and Cheng, Y. (2010). Velocity fields of granular flows down a rough incline: a dem investigation. *Granular Matter*, 12(5):477.

Marchisio, D. L. and Fox, R. O. (2005). Solution of population balance equations using the direct quadrature method of moments. *Journal of Aerosol Science*, 36(1):43–73.

Marks, B., Rognon, P., and Einav, I. (2012a). Grainsize dynamics of polydisperse granular segregation down inclined planes. *Journal of Fluid Mechanics*, 690:499–511.

Marks, B., Rognon, P., and Einav, I. (2012b). Grainsize dynamics of polydisperse granular segregation down inclined planes. *Journal of Fluid Mechanics*, 690:499–511.

May, L. B., Golick, L. A., Phillips, K. C., Shearer, M., and Daniels, K. E. (2010). Shear-driven size segregation of granular materials: Modeling and experiment. *Physical Review E*, 81(5):051301.

McDowell, G. R., De Bono, J. P., Yue, P., and Yu, H.-S. (2013). Mi-

cro mechanics of isotropic normal compression. *Géotechnique Letters*, 3(4):166–172.

Montanero, J., Garzó, V., Santos, A., and Brey, J. (1999). Kinetic theory of simple granular shear flows of smooth hard spheres. *Journal of Fluid Mechanics*, 389:391–411.

Mullin, T. (2000). Coarsening of self-organized clusters in binary mixtures of particles. *Physical Review Letters*, 84(20):4741.

Muzzio, F. J., Shinbrot, T., and Glasser, B. J. (2002). Powder technology in the pharmaceutical industry: the need to catch up fast.

Nikitas, G., Arany, L., Aingaran, S., Vimalan, J., and Bhattacharya, S. (2017). Predicting long term performance of offshore wind turbines using cyclic simple shear apparatus. *Soil Dynamics and Earthquake Engineering*, 92:678–683.

Ottino, J. and Khakhar, D. (2000). Mixing and segregation of granular materials. *Annual Review of Fluid Mechanics*, 32(1):55–91.

Pihler-Puzović, D. and Mullin, T. (2013). The timescales of granular segregation in horizontally shaken monolayers. *Proceedings of the Royal Society A: Mathematical, Physical and Engineering Sciences*, 469(2157):20130203.

Pudasaini, S. P., Hsiau, S.-S., Wang, Y., and Hutter, K. (2005). Velocity measurements in dry granular avalanches using particle image velocimetry technique and comparison with theoretical predictions. *Physics of Fluids*, 17(9).

Pudasaini, S. P. and Hutter, K. (2007). *Avalanche dynamics: dynamics of rapid flows of dense granular avalanches*. Springer Science & Business Media.

Qiao, J., Duan, C., Dong, K., Wang, W., Jiang, H., Zhu, H., and Zhao, Y. (2021). Dem study of segregation degree and velocity of binary granular mixtures subject to vibration. *Powder Technology*, 382:107–117.

Quinn, P. V., Hong, D. C., and Luding, S. (2002). Quinn, hong, and luding reply. *Physical Review Letters*, 89(18):189604.

Ramkrishna, D. (2000). *Population balances: Theory and applications to particulate systems in engineering*. Elsevier.

Ramkrishna, D. and Singh, M. R. (2014). Population balance modeling: current status and future prospects. *Annual review of chemical and biomolecular engineering*, 5:123–146.

Rosato, A., Strandburg, K. J., Prinz, F., and Swendsen, R. H. (1987). Why the brazil nuts are on top: Size segregation of particulate matter by shaking. *Physical Review Letters*, 58(10):1038.

Savage, S. and Lun, C. (1988). Particle size segregation in inclined chute flow of dry cohesionless granular solids. *Journal of Fluid Mechanics*, 189:311–335.

Scott, A. M. and Bridgwater, J. (1975). Interparticle percolation: a fundamental solids mixing mechanism. *Industrial & Engineering Chemistry Fundamentals*, 14(1):22–27.

Thevakumar, K., Indraratna, B., Ferreira, F. B., Carter, J., and Rujikiatkamjorn, C. (2021). The influence of cyclic loading on the response of soft subgrade soil in relation to heavy haul railways. *Transportation Geotechnics*, 29:100571.

Thornton, A. and Gray, J. (2008). Breaking size segregation waves and par-

ticle recirculation in granular avalanches. *Journal of Fluid Mechanics*, 596:261–284.

Thornton, A. R., Gray, J., and Hogg, A. (2006). A three-phase mixture theory for particle size segregation in shallow granular free-surface flows. *Journal of Fluid Mechanics*, 550:1–25.

Truesdell, C. and Truesdell, C. (1984). Thermodynamics of diffusion. *Rational thermodynamics*, pages 219–236.

Tunuguntla, D. R., Weinhart, T., and Thornton, A. R. (2017). Comparing and contrasting size-based particle segregation models: Applying coarse-graining to perfectly bidisperse systems. *Computational Particle Mechanics*, 4:387–405.

van der Vaart, K., Gajjar, P., Epely-Chauvin, G., Andreini, N., Gray, J., and Ancey, C. (2015). Underlying asymmetry within particle size segregation. *Physical Review Letters*, 114(23):238001.

Wang, J. and Yan, H. (2013). On the role of particle breakage in the shear failure behavior of granular soils by dem. *International Journal for Numerical and Analytical Methods in Geomechanics*, 37(8):832–854.

Woodhouse, M., Thornton, A. R., Johnson, C. G., Kokelaar, B., and Gray, J. (2012). Segregation-induced fingering instabilities in granular free-surface flows. *Journal of fluid Mechanics*, 709:543–580.

Yu, F. (2021). Particle breakage in granular soils: a review. *Particulate Science and Technology*, 39(1):91–100.

Zhang, T., Zhang, C., Zou, J., Wang, B., Song, F., and Yang, W. (2020). Dem exploration of the effect of particle shape on particle breakage in granular assemblies. *Computers and Geotechnics*, 122:103542.

Appendices

Appendix A

Python code for classic bi-disperse simulation

```
"""
Created on Sat Nov 21 10:50:03 2020

@author: Jiaxin Zhang
"""
# Import useful modules
import numpy as np
import matplotlib.pyplot as plt

# Define global variables
# B = -0.0003
# C = 0.2
# D = 0.00007
# HP0 = 10000
B = 0.625
C = 2
D = 0.2
HP0 = 0.5

T = 1
# H = 1
# # U  = 0.05
# c = 100
# SLS = 0.0016
# Pe = 20.9

# # I = 1/(np.sqrt(H*H + U*U*T*T))
# # J = (np.sqrt(H*H + U*U*T*T))*(9.81*H + U*U*T*T)/H

# B = -SLS*(H*c)/(T*9.81)
# d = -B/Pe*9.81

# s = 1.5
```

```
# rho = 2500
# P0 = 0
# HP0 = P0/(rho*9.81*H)

nX = 10
nZ = 20
nT = 5000

xStart = 0
xEnd = 5
zBot = 0
zTop = 1
tEnd = 5
# t0Rev = np.array([10, 20])
# tRev = T*0.5
# n0Rev = nT/tEnd*t0Rev + 1
# nRev = nT/tEnd*tRev

dx = (xEnd - xStart)/nX
dz = (zTop - zBot)/nZ
dt = tEnd/(nT-1)
x = np.linspace(xStart,xEnd,nX+1)
z = np.linspace(zBot,zTop,nZ+1)
time = np.linspace(0,tEnd,nT)

uTempCell = np.zeros([nZ,1])
uTempGrdpt = np.zeros([nZ+1,1])

"""
Some check parameters
"""
# aXCheck = np.zeros([nZ,nX+1])
# aZCheck = np.zeros([nZ+1,nX])

# ============================================================================
# Create the list of functions needed
# ============================================================================

# def sign(a):
#     """
#     Returns sign of the number.
#     Returns 0 if input is 0.
#     """
#     if a == 0.0:
#         output = 0
#     if a > 0.0:
#         output = 1
#     if a < 0.0:
#         output = -1
#     return output


def minmod(a,b):
    """
    Returns the flatter local slope out of the given two.
    If the two slopes have different signs, then it returns 0.
    """
    output = 0.5*(np.sign(a)+np.sign(b)) * np.min([np.abs(a),np.abs(b)])
```

```
    return output


def segB(dudz):
    """
    Returns the local pairwise segregation rate parameter
    """
    global B
    output = B
    # output = B
    return output


def fluxFunX(phi,iZ):
    """
    Flux function in x-direction, collects velocity profile via global.
    """
    global uTempCell
    output = phi * uTempCell[iZ]
    return output


def fluxFunXdPhi(iZ):
    """
    Derivative of flux function in x-direction.
    """
    global uTempCell
    output = uTempCell[iZ]
    return output


def fluxFunZ(phi,dudz):
    """
    Flux function in z-direction,
    with prescribed dependency on velocity gradient.
    """
    # global T
    # global H
    # global c
    # global B
    # global I
    # global s
    global C
    BTemp= segB(dudz)
    output = (1/C) * BTemp * phi * (1-phi)
    # output = (1/C)*phi + (1/C) * BTemp * phi * (1-phi)
    # output = (1/C) * BTemp * phi * (1-phi) + phi/C

    # bot = phi + s*(1-phi)
    # output = (1/C)*phi/bot
    return output


def fluxFunZdPhi(phi,dudz):
    """
    Derivative of flux function in z-direction.
    """
    # global T
    # global H
```

```
    # global c
    # global B
    # global I
    global s
    global C
    BTemp= segB(dudz)
    output = (1/C) * BTemp * (1-2*phi)
    # output = (1/C) + (1/C) * BTemp * (1-2*phi)
    # output = (1/C) * BTemp * (1-2*phi) + 1/C

    # bot = (phi+s*(1-phi))*(phi+s*(1-phi))
    # output = (1/C)*s/bot
    return output

# def fluxFunZ(phi):
#     """
#     Flux function in z-direction,
#     with prescribed dependency on velocity gradient.
#     """
#     global T
#     global H
#     global c
#     global B
#     output = (T*9.81*B)/(H*c) * phi * (1-phi)
#     return output

# def fluxFunZdPhi(phi):
#     """
#     Derivative of flux function in z-direction.
#     """
#     global T
#     global H
#     global c
#     global B
#     output = (T*9.81*B)/(H*c) * (1-2*phi)
#     return output


def fluxFunDiff(dphidz,iz,phi,dudz):
    """
    Diffusion flux function, more modifications to be applied
    """
    # global T
    # global d
    # global H
    # global c
    # global B
    # global rho
    # global J
    # global zTop
    # global HP
    global D
    global C
    global HP0
    global s
    BTemp= segB(dudz)
    # output = (D/C)*dphidz
    z = zBot + (0.5+iz)*dz
    output = (D/C)*dphidz + (1/C)*(HP0 + (1-z))*(1+BTemp*(1-phi))*dphidz
```

```
    # output = 0

    # bot = (phi+s*(1-phi))*(phi+s*(1-phi))
    # output = (D/C)*dphidz + (1/C)*(HP0 + (1-z))*(s/bot)*dphidz
    return output

# def fluxFunDiff(dphidz):
#     """
#     Diffusion flux function, more modifications to be applied
#     """
#     global T
#     global d
#     global H
#     global c
#     output = (T*d)/(H*c) * dphidz
#     return output

# def p(z):
#     global zTop
#     global rho
#     global P0
#     output = rho*9.81*(zTop-z) + P0
#     return output


# def u(z,alpha,beta):
#     """
#     Prescribed bulk velocity profile,
#     with its shape inspired by experimental results.
#     """
#     global uAmp
#     # global alpha
#     # global beta
#     L = 1 - 1/alpha
#     norm = np.tanh(beta/np.pi) - 1
#     if z >= L and z<= 1:
#         output = uAmp * (np.tanh(beta/(np.pi*alpha*(z-L)))-1)/norm
#         # output = uAmp * (np.tanh(beta/(np.pi*z))-1)/norm
#     else:
#         output = 0
#     return output

def u(z,R,velTop):
    """
    Prescribed bulk velocity profile,
    with its shape inspired by experimental results.
    """
    # global alpha
    # global beta
    norm = np.tanh(R/np.pi) - 1
    norm = norm/velTop
    output = np.sign(R)*(np.tanh(R/(np.pi*z))-1)/norm

    return output

# def getList(alphaList, betaList, z):
#     # global alpha
#     # global beta
#     """
```

```
#     Get incremental values
#     to be iterated and superposed for initialization process.
#     """
#     uNum = len(alphaList) + len(betaList)
#     uList = np.zeros([len(z),uNum])
#     "reverse the order of vector for top-down archiving"
#     # zRev = z[::-1]
#     # zRev = z

#     "loop over alpha list to record full evolution of velocity profile"
#     uI = 0
#     for alphaI in range(len(alphaList)):
#         alphaTemp = alphaList[alphaI]
#         betaTemp = betaList[0]
#         uTemp = np.zeros([len(z),1])
#         for i in range(len(z)):
#             alpha = alphaTemp
#             beta = betaTemp
#             # uTemp[i] = u(zRev[i],alpha,beta)
#             uTemp[i] = u(z[i],alpha,beta)
#             uList[i,uI] = uTemp[i]

#         uI = uI + 1

#     "loop over beta list to record full evolution of velocity profile"
#     uI = 0
#     for betaI in range(len(betaList)):
#         alphaTemp = 1
#         betaTemp = betaList[betaI]
#         uTemp = np.zeros([len(z),1])
#         for i in range(len(z)):
#             alpha = alphaTemp
#             beta = betaTemp
#             # uTemp[i] = u(zRev[i],alpha,beta)
#             uTemp[i] = u(z[i],alpha,beta)
#             uList[i,uI + len(alphaList)] = uTemp[i]

#         uI = uI + 1
#     return uList


# def getIncrements(alphaList, betaList, z, uList):
#     """
#     Take incremented values of velocity profiles based on given uList.
#     """
#     uNum = len(alphaList) + len(betaList)
#     duList = np.zeros([len(z),uNum])
#     duList[:,0] = uList[:,0]
#     for uI in range(1,uNum):
#         for i in range(len(z)):
#             duList[i,uI] = np.subtract(uList[i,uI], uList[i,uI-1])
#     return duList

# ================================================================================
# KT scheme function
# ================================================================================

def getRHS(phi):
    """
```

```
Semi-discrete solver that converts the PDE to a time-dependent ODE for
each time-step. Right-hand-side value of the ODE dphidt = RHS is outputed
along with other useful information.
"""
global nZ, nX, dx, dz
global direction, uTempCell, uTempGrdpt
global zBot

global phiC
phiC = phi
"""
Define the local slopes in x and z directions.
Stored at cells.
"""
phiX = np.zeros([nZ,nX])
phiZ = np.zeros([nZ,nX])

"Note: due to the zero initialization,"
"phiX = 0 at x = {xStart,xEnd} and"
"phiZ = 0 at z = {zBot,zTop} are automatically satisfied."
"Therefore it remains to set other local slope entries "
"using minmod limiter function."
for i in range(nZ):
    for j in range(1,nX-1):
        left = (phi[i,j] - phi[i,j-1])/dx
        right = (phi[i,j+1] - phi[i,j])/dx
        phiX[i,j] = minmod(left, right)
global downs, ups
downs = np.zeros([nZ,nX])
ups = np.zeros([nZ,nX])
for i in range(1,nZ-1):
    for j in range(nX):
        down = (phi[i,j] - phi[i-1,j])/dz
        up = (phi[i+1,j] - phi[i,j])/dz
        downs[i,j] = down
        ups[i,j] = up
        phiZ[i,j] = minmod(down, up)

"""
Define back and forward linear approximations of phi in x and z directions
based on local slopes.
Stored at Gridpoints.
"""
phiL = np.zeros([nZ,nX+1])
phiR = np.zeros([nZ,nX+1])
phiD = np.zeros([nZ+1,nX])
phiU = np.zeros([nZ+1,nX])

"Note: linear approximations at boundary endpoints are assumed to be 0."
"This is because one of their neighbours is non-existant. This BC is "
"automatically satisfied via zero initialization."
for i in range(nZ):
    for j in range(nX):
        "left(x-) linear approximation based on phiX"
        phiL[i,j+1] = phi[i,j] + phiX[i,j]*dx*0.5
        "right(x+) linear approximation based on phiX"
        phiR[i,j] = phi[i,j] - phiX[i,j]*dx*0.5
        "down(z-) linear approximation based on phiZ"
        phiD[i+1,j] = phi[i,j] + phiZ[i,j]*dz*0.5
```

```
        "up(z+) linear approximation based on phiZ"
        phiU[i,j] = phi[i,j] - phiZ[i,j]*dz*0.5

"""
Record the downstream velocity component values at cells and gridpoints
in order to calculate du/dz.
"""
uValCell = np.zeros([nZ,1])
uValGrdpt = np.zeros([nZ+1,1])

for i in range(nZ):
    uValCell[i] = uTempCell[i]

for i in range(nZ+1):
    uValGrdpt[i] = uTempGrdpt[i]

"""
Use minmod limiter to numerically calculate magnitude of local velocity
gradients du/dz at cells and gridpoints.
"""
dudzCell = np.zeros([nZ,1])
dudzGrdpt = np.zeros([nZ+1,1])

for i in range(1,nZ-1):
    L = (uValCell[i] - uValCell[i-1])/dz
    R = (uValCell[i+1] - uValCell[i])/dz
    dudzCell[i] = np.abs(minmod(L,R))

for i in range(1,nZ):
    L = ( uValGrdpt[i] -  uValGrdpt[i-1])/dz
    R = ( uValGrdpt[i+1] -  uValGrdpt[i])/dz
    dudzGrdpt[i] = np.abs(minmod(L,R))

"""
Define the maximum local speeds aX and aZ at cell endpoints in x and z
directions. Again additional row/col is required.
"""
aX = np.zeros([nZ,nX+1])
aZ = np.zeros([nZ+1,nX])
aU = np.zeros([nZ+1,nX])
aD = np.zeros([nZ+1,nX])

for i in range(nZ):
    for j in range(nX+1):
        aL = np.abs(fluxFunXdPhi(i))
        aR = np.abs(fluxFunXdPhi(i))
        aX[i,j] = np.max([aL, aR])

for i in range(nZ+1):
    for j in range(nX):
        aU[i,j] = np.abs(fluxFunZdPhi(phiU[i,j],dudzGrdpt[i]))
        aD[i,j] = np.abs(fluxFunZdPhi(phiD[i,j],dudzGrdpt[i]))
        aZ[i,j] = np.max([aU[i,j], aD[i,j]])

"""
Define the numerical flux terms Hx and Hz
in x and z directions stored at cell endpoints.
"""
HX = np.zeros([nZ,nX+1])
```

```
HZ = np.zeros([nZ+1,nX])

"First prescribe the inflow and outflow BCs in x, now periodic."
for i in range(nZ):
    if direction == 1:
        HX[i,0] = fluxFunX(phi[i,nX-1], i)
        HX[i,nX] = fluxFunX(phi[i,nX-1],i)
    else:
        HX[i,nX] = fluxFunX(phi[i,1],i)
        HX[i,0] = fluxFunX(phi[i,1],i)

    for j in range(1,nX):
        HX[i,j] = 0.5*(fluxFunX(phiR[i,j],i) + fluxFunX(phiL[i,j],i)) \
                  -0.5 * aX[i,j] * (phiR[i,j] - phiL[i,j])

"First prescribe top and bottom no-flux BC in z."
for j in range(nX):
    HZ[0,j] = 0
    HZ[nZ,j] = 0

    for i in range(1,nZ):
        # HZ[i,j] = 0.5*(fluxFunZ(phiU[i,j],dudzCell[i]) + \
        #                fluxFunZ(phiD[i,j],dudzCell[i])) \
        #           -0.5* aZ[i,j] * (phiU[i,j] - phiD[i,j])
        HZ[i,j] = 0.5*(fluxFunZ(phiU[i,j], dudzCell[i]) + \
                       fluxFunZ(phiD[i,j], dudzCell[i])) \
                  -0.5* aZ[i,j] * (phiU[i,j] - phiD[i,j])

"""
Define the numerical diffusion flux terms.
Stored at gridpoints.
"""
PZ = np.zeros([nZ+1,nX])
for j in range(nX):
    for i in range(1,nZ):
        # zUp = zTop - (i-0.5)*dz
        # zDown = zTop - (i+0.5)*dz
        # z = zTop - (i+0.5)*dz

        dphidz = (phi[i,j] - phi[i-1,j])/dz
        PZ[i,j] = 0.5*(fluxFunDiff(dphidz,i,phi[i,j],dudzCell[i]) \
                       + fluxFunDiff(dphidz,i,phi[i,j],dudzCell[i]))

RHS = np.zeros([nZ,nX])
for i in range(nZ):
    for j in range(nX):
        RHS[i,j] = -(HX[i,j+1] - HX[i,j])/dx \
                   -(HZ[i+1,j] - HZ[i,j])/dz \
                   +(PZ[i+1,j] - PZ[i,j])/dz
"""
Checks
"""
# global XCheck, ZCheck
# XCheck = phiX
# ZCheck = phiZ
# global LCheck, RCheck, DCheck, UCheck
# LCheck = phiL
# RCheck = phiR
# DCheck = phiD
```

```
    # UCheck = phiU
    # global aXCheck, aZCheck
    # aXCheck = aX
    # aZCheck = aZ
    # global HXC,HZC
    # HXC = HX
    # HZC = HZ
    # global PZC
    # PZC = PZ
    # global RHSC
    # RHSC = RHS

    return RHS


def rk_timestep(phi):
    """
    Runge-Kutta timestepper to solve the ODE for each timestep.
    """
    global nX, nZ, dt
    eta = 0.5
    phiTemp = np.zeros([nZ,nX])
    phiNew = np.zeros([nZ,nX])

    RHS = getRHS(phi)
    for i in range(nZ):
        for j in range(nX):
            phiTemp[i,j]  = phi[i,j] + dt*RHS[i,j]

    RHSTemp = getRHS(phiTemp)
    for i in range(nZ):
        for j in range(nX):
            phiNew[i,j] = eta*phi[i,j] + \
                          (1-eta) * (phiTemp[i,j] + dt*RHSTemp[i,j])

    return phiNew
    # return phiTemp


# ================================================================================
# Main function
# ================================================================================

"Initial volume fraction condition"
# phi = np.concatenate([ \
#     0.2*np.ones([int(nZ*0.5),nX]),0.8*np.ones([int(nZ*0.5),nX])])
# phi = np.ones([nZ,nX])*0.6
phi = np.ones([nZ,nX]) * (1/2)

"Generate velocity evolution data to iterate in time"
zCell = np.linspace(zBot,zTop-dz,nZ)+0.5*dz
# fuListCell = getList(alphaList, betaList, zCell)
# fduListCell = getIncrements(alphaList, betaList, zCell, uListCell)
# # uListCell = np.flipud(getList(alphaList, betaList, zCell))
# # duListCell = np.flipud(getIncrements(alphaList,betaList,zCell,uListCell))

# zMod = z
# zMod[0] = z[0]+0.01*dz
# uListGrdpt = getList(alphaList, betaList, zMod)
```

```
# duListGrdpt = getIncrements(alphaList, betaList, z, uListGrdpt)
# # uListGrdpt = np.flipud(getList(alphaList, betaList, zMod))
# # duListGrdpt = np.flipud(getIncrements(alphaList, betaList, z, uListGrdpt))

uListCellHist = np.zeros([nZ,nT])
duListCellHist = np.zeros([nZ,nT])

"Initialize matrix to store volume fraction and other data"
results = np.zeros([nZ,nT+1])

# Seg1 = np.zeros([nZ,nT])
# Seg2 = np.zeros([nZ,nT])
# Seg3 = np.zeros([nZ,nT])
# SegVel = np.zeros([nZ,nT])
# SegFlux = np.zeros([nZ,nT])
# DiffFlux = np.zeros([nZ,nT])

"Record IC in results"
for i in range(nZ):
    results[i,0] = 0.5*(phi[i,int(nX/2)] + phi[i,int(nX/2)+1])

"Index used to document evolution progress of velocity profile in time"
# indexI = 0
# indexR = len(alphaList) + len(betaList)

"Loop over time"
for t in range(nT):
    # "check whether the timestep is the start of the initialization period"
    # if indexI+1 <= len(alphaList)+len(betaList):
    #     for i in range(nZ):
    #         uTempCell[i] = uTempCell[i] + direction * duListCell[i,indexI]
    #     for i in range(nZ+1):
    #         uTempGrdpt[i] = uTempGrdpt[i] + direction * duListGrdpt[i,indexI]
    #         "update the counter"
    #     indexI = indexI + 1

    # "check whether the timestep is the start of the reversal period"
    # if t in n0Rev:
    #     "reset the reversal progress counter and record change of direction"
    #     indexR = 0
    #     direction = -1*direction
    #     for i in range(nZ):
    #         uTempCell[i] = uTempCell[i] + 2*direction*duListCell[i,indexR]
    #     for i in range(nZ+1):
    #         uTempGrdpt[i] = uTempGrdpt[i] + 2*direction*duListGrdpt[i,indexR]

    #     "update the counter"
    #     indexR = indexR + 1

    # "check whether the timestep is within the reversal period"
    # if t not in n0Rev and indexR+1 <= len(alphaList)+len(betaList):
    #     "add the prescribed increment assigned to this step"
    #     for i in range(nZ):
    #         uTempCell[i] = uTempCell[i] + 2*direction*duListCell[i,indexR]
    #     for i in range(nZ+1):
    #         uTempGrdpt[i] = uTempGrdpt[i] + 2*direction*duListGrdpt[i,indexR]

    #     "update the counter"
    #     indexR = indexR + 1
```

```
"Second version of velocity profile evolution"
RTemp = 5*np.cos(2*np.pi*t/T)+5.5
velTopTemp = np.sin(np.pi*t/T)
direction = np.sign(velTopTemp)
for i in range(nZ):
    zTemp = zTop - (i+0.5)*dz
    uTempCell[i] = u(zTemp,RTemp,velTopTemp)
for i in range(nZ+1):
    zTemp = zTop - i*dz
    uTempGrdpt[i] = u(zTemp,RTemp,velTopTemp)

"record velocity evolution data"
for i in range(nZ):
    uListCellHist[i,t] = uTempCell[i]

"record shear rate data"
for i in range(1,nZ-1):
    dudzTemp = minmod((uTempCell[i] - uTempCell[i-1])/dz,
                      (uTempCell[i+1] - uTempCell[i])/dz)
    duListCellHist[i,t] = dudzTemp

# "record velocity evolution data"
# for i in range(nZ):
#     uListCellHist[i,t] = uTempCell[i]

# "record shear rate data"
# for i in range(1,nZ-1):
#     dudzTemp = minmod((uTempCell[i] - uTempCell[i-1])/dz,
# (uTempCell[i+1] - uTempCell[i])/dz)
#     duListCellHist[i,t] = dudzTemp

# uTempCell = np.ones([nZ,1])
# uTempGrdpt = np.ones([nZ+1,1])

"calculate phi for next timestep using KT solver"
phiNew = rk_timestep(phi)
phi = phiNew

"store the central column for evolution graph"
if np.mod(nX,2) == 0:
    for i in range(nZ):
        results[i,t+1] = 0.5*(phi[i,int(nX*0.5)] + phi[i,int(nX*0.5)+1])

elif np.mod(nX,2) == 1:
    for i in range(nZ):
        results[i,t+1] = phi[i,(nX+1)*0.5]

# "Calculate and store the shear rate data"
# dudzCellTemp = np.zeros([nZ,1])
# for i in range(1,nZ-1):
#     L = (uTempCell[i] - uTempCell[i-1])/dz
#     R = (uTempCell[i+1] - uTempCell[i])/dz
#     dudzCellTemp[i] = np.abs(minmod(L,R))

# "calculate and store the segregation velocity normalized by sqrt(H/g)"
# for i in range(nZ):
#     zTemp = zTop - (i+0.5)*dz
#     phiTemp = results[i,t]
#     if i == 0 or i == nZ-1:
```

```
    #           dphidzTemp = 0
    #       else:
    #           dphidzTemp = minmod((results[i,t] - results[i-1,t])/dz, \
    #                               (results[i+1,t] - results[i,t])/dz)
    #       BTemp = segB(dudzCellTemp[i])
    #       Seg1Temp = 9.81/(C*T)*BTemp*(1-phiTemp)
    #       Diff1Temp = -(D/C)*(1/phiTemp)*dphidzTemp
    #       Diff2Temp = -1/(C*T*9.81)*(HP0 + (1-zTemp))* \
    #           (1+BTemp*(1-phiTemp))/phiTemp*dphidzTemp
    #       # Seg1 = 9.81/(C*T)*B*(1-phiTemp)*phiTemp
    #       # Diff1 = -(D/C)*dphidzTemp
    #       # Diff2 = -1/(C*T*9.81)*(HP0+(1-zTemp))*(1+B*(1-phiTemp))*dphidzTemp

    #       Seg1[i,t] = Seg1Temp
    #       Seg2[i,t] = Diff1Temp
    #       Seg3[i,t] = Diff2Temp
    #       SegVel[i,t] = Seg1Temp + Diff1Temp + Diff2Temp


    # for i in range(nZ):
    #       phiTemp = results[i,t]
    #       SegFlux[i,t] = fluxFunZ(phiTemp, dudzCellTemp[i])

    # "calculate and store the magnitude of diffusion flux"

    # for i in range(nZ):
    #       zTemp = zTop - (i+0.5)*dz
    #       phiTemp = results[i,t]
    #       if i == 0 or i == nZ-1:
    #           dphidzTemp = 0
    #       else:
    #           dphidzTemp = minmod((results[i,t] - results[i-1,t])/dz, \
    #                               (results[i+1,t] - results[i,t])/dz)
    #       DiffFlux[i,t] = fluxFunDiff(dphidzTemp,zTemp,phiTemp,dudzCellTemp[i])

    "counter"
    print("timestep number ",t ,"completed")


"Plot the evolution graph"
# fig, axes =  plt.subplots(1,1,figsize = (5,5))
# plt.imshow(np.flipud(results), interpolation='none', \
#             cmap='viridis',aspect=nT/nZ)
# plt.imshow(np.flipud(np.ones([nZ,nT+1])-results), \
#             interpolation='none', cmap='viridis',aspect=nT/nZ)
# plt.show()

"Save the evolution data"
np.savetxt("results0_10_P0_0.5_spread_real1a_bi.csv", results, delimiter=",")
# np.savetxt("SegVel_B_0.9_C_20_D_0_HP0_0.1_flipped_v.csv", SegVel, delimiter=",")
# np.savetxt("SegFlux_B_0.9_C_20_D_0_HP0_0.1_flipped_v.csv", SegFlux, delimiter
    =",")
# np.savetxt("DiffFlux_B_0.9_C_20_D_0_HP0_0.1_flipped_v.csv", DiffFlux, delimiter
    =",")
```

Appendix B

Python code for classic tri-disperse simulation

```
"""
Created on Sat Nov 21 10:50:03 2020

@author: Jiaxin Zhang
"""
# Import useful modules
import numpy as np
from numpy import linalg as LA
import matplotlib.pyplot as plt

# Define global variables
# B = -0.0003
# C = 0.2
# D = 0.00007
# HP0 = 10000
BLS = 0.625
BLM = 0.378125
BMS = 0.3136
# sizeRatioLS = 10
# sizeRatioMS = 1
# sizeRatioLM = sizeRatioLS / sizeRatioMS
# sizeRatioDim = 1


C = 2
D = 0.2
HP0 = 0.5

T = 1
# H = 1
# # U  = 0.05
# c = 100
# SLS = 0.0016
```

```
# Pe = 20.9

# # I = 1/(np.sqrt(H*H + U*U*T*T))
# # J = (np.sqrt(H*H + U*U*T*T))*(9.81*H + U*U*T*T)/H

# B = -SLS*(H*c)/(T*9.81)
# d = -B/Pe*9.81

s = 1.5

# rho = 2500
# P0 = 0
# HP0 = P0/(rho*9.81*H)

nX = 10
nZ = 20
nT = 5000

xStart = 0
xEnd = 5
zBot = 0
zTop = 1
tEnd = 5
# t0Rev = np.array([10,20])
# tRev = T*0.5
# n0Rev = nT/tEnd*t0Rev + 1
# nRev = nT/tEnd*tRev

# uAmp = 1
# alpha = np.linspace(3, 1.04, int(nRev*0.8))
# # alpha = np.linspace(3, 1.04, 50)
# alphaList = np.multiply(alpha,alpha)
# betaList = np.linspace(8, 1, int(nRev*0.5))
# # betaList = np.linspace(10, 0.5, 10)

x = np.linspace(xStart,xEnd,nX+1)
z = np.linspace(zBot,zTop,nZ+1)
time = np.linspace(0,tEnd,nT+1)

dx = (xEnd - xStart)/nX
dz = (zTop - zBot)/nZ
dt = tEnd/(nT-1)


# direction = 1
uTempCell = np.zeros([nZ,1])
uTempGrdpt = np.zeros([nZ+1,1])


# ============================================================================
# Create the list of functions needed
# ============================================================================
def sign(a):
    """
    Returns sign of the number.
    Returns 0 if input is 0.
    """
    if a == 0.0:
        output = 0
```

```
    if a > 0.0:
        output = 1
    if a < 0.0:
        output = -1
    return output

def minmod(a,b):
    """
    Returns the flatter local slope out of the given two.
    If the two slopes have different signs, then it returns 0.
    """
    output = 0.5*(np.sign(a)+np.sign(b)) * np.min([np.abs(a),np.abs(b)])
    return output

def segB(dudz):
    """
    Returns the local pairwise segregation rate parameter
    """
    global BLS, BLM, BMS

    BTemp = np.zeros([3,1])
    # BTemp[0] = np.sqrt(dudz)*BLS
    # BTemp[1] = np.sqrt(dudz)*BLM
    # BTemp[2] = np.sqrt(dudz)*BMS
    BTemp[0] = BLS
    BTemp[1] = BLM
    BTemp[2] = BMS
    # output = (0.5 + np.sqrt(dudz))*B
    output = BTemp
    return output

def fluxFunX(phiL,phiS,iZ):
    """
    Flux function in x-direction, collects velocity profile via global.
    """
    global uTempCell

    output = np.zeros([2,1])
    output[0] = phiL * uTempCell[iZ]
    output[1] = phiS * uTempCell[iZ]
    return output

def fluxFunXdPhi(iZ):
    """
    Derivative of flux function in x-direction.
    """
    global uTempCell
    output = uTempCell[iZ]
    return output

# def fluxFunZ(phi,dudz):
#     """
#     Flux function in z-direction, with prescribed dependency on velocity
    gradient.
#     """
#     # global T
#     # global H
#     # global c
#     # global B
```

```
#     # global I
#     global s
#     global C
#     BTemp= segB(dudz)
#     output = (1/C) * BTemp * phi * (1-phi)

#     # bot = phi + s*(1-phi)
#     # output = (1/C)*phi/bot
#     return output

def fluxFunZ(phiL,phiS,dudz):
    """
    Flux function in z-direction, with prescribed dependency on velocity gradient.
    """
    output = np.zeros([2,1])
    "pair-wise function"
    global C
    BTemp= segB(dudz)
    output[0] = (1/C)*BTemp[0]*phiL*phiS + (1/C)*BTemp[1]*phiL*(1-phiL-phiS)
    output[1] = -(1/C)*BTemp[0]*phiL*phiS - (1/C)*BTemp[2]*phiS*(1-phiL-phiS)
    "quotient function"
    # global sizeRatioLS, sizeRatioLM, sizeRatioMS, sizeRatioDim
    # botL = phiL + (1/sizeRatioLS)**sizeRatioDim*phiS + (1/sizeRatioLM)**
        sizeRatioDim*(1-phiL-phiS)
    # botS = phiS + sizeRatioLS**sizeRatioDim*phiL + sizeRatioMS**sizeRatioDim*(1-
        phiL-phiS)
    # output[0] = (1/C)*phiL/botL
    # output[1] = (1/C)*phiS/botS

    return output

def fluxFunZJacobian(phiL,phiS,dudz):

    output = np.zeros([2,2])
    "pair-wise function"
    global C
    BTemp = segB(dudz)
    output[0,0] = (1/C)*BTemp[1] - 2*(1/C)*BTemp[1]*phiL + (1/C)*(BTemp[0]-BTemp
        [1])*phiS
    output[0,1] = (1/C)*(BTemp[0]-BTemp[1])*phiL
    output[1,0] = (1/C)*(BTemp[2]-BTemp[0])*phiS
    output[1,1] = -(1/C)*BTemp[2] + 2*(1/C)*BTemp[2]*phiS + (1/C)*(BTemp[2]-BTemp
        [0])*phiL

    "quotient function"
    # global sizeRatioLS, sizeRatioLM, sizeRatioMS, sizeRatioDim
    # botL = phiL + (1/sizeRatioLS)**sizeRatioDim*phiS + (1/sizeRatioLM)**
        sizeRatioDim*(1-phiL-phiS)
    # botS = phiS + sizeRatioLS**sizeRatioDim*phiL + sizeRatioMS**sizeRatioDim*(1-
        phiL-phiS)
    # up1 = ((1/sizeRatioLS)**sizeRatioDim - (1/sizeRatioLM)**sizeRatioDim)*phiS +
         (1/sizeRatioLM)**sizeRatioDim
    # up2 = ((1/sizeRatioLM)**sizeRatioDim - (1/sizeRatioLS)**sizeRatioDim)*phiL
    # up3 = (sizeRatioMS**sizeRatioDim - sizeRatioLS**sizeRatioDim)*phiS
    # up4 = (sizeRatioLS**sizeRatioDim - sizeRatioMS**sizeRatioDim)*phiL +
        sizeRatioMS**sizeRatioDim
    # output[0,0] = (1/C)*up1/(botL**2)
    # output[0,1] = (1/C)*up2/(botL**2)
    # output[1,0] = (1/C)*up3/(botS**2)
```

```
    # output[1,1] = (1/C)*up4/(botS**2)

    return output

def getRho(jacobian):
    w,v = LA.eig(jacobian)
    output = np.amax(np.abs(w))
    return output

# def fluxFunZdPhi(phi,dudz):
#     """
#     Derivative of flux function in z-direction.
#     """
#     # global T
#     # global H
#     # global c
#     # global B
#     # global I
#     global s
#     global C
#     # BTemp= segB(dudz)
#     # output = (1/C) * BTemp * (1-2*phi)

#     bot = (phi+s*(1-phi))*(phi+s*(1-phi))
#     output = (1/C)*s/bot
#     return output

# def fluxFunZ(phi):
#     """
#     Flux function in z-direction, with prescribed dependency on velocity
    gradient.
#     """
#     global T
#     global H
#     global c
#     global B
#     output = (T*9.81*B)/(H*c) * phi * (1-phi)
#     return output

# def fluxFunZdPhi(phi):
#     """
#     Derivative of flux function in z-direction.
#     """
#     global T
#     global H
#     global c
#     global B
#     output = (T*9.81*B)/(H*c) * (1-2*phi)
#     return output

# def fluxFunDiff(dphidz,z,phi,dudz):
#     """
#     Diffusion flux function, more modifications to be applied
#     """
#     # global T
#     # global d
#     # global H
#     # global c
#     # global B
```

```
#     # global rho
#     # global J
#     # global zTop
#     # global HP
#     global D
#     global C
#     global HP0
#     global s
#     # BTemp= segB(dudz)
#     # output = (D/C)*dphidz + (1/C)*(HP0 + (1-z))*(1+BTemp*(1-phi))*dphidz
#     # output = 0

#     bot = (phi+s*(1-phi))*(phi+s*(1-phi))
#     output = (D/C)*dphidz + (1/C)*(HP0 + (1-z))*(s/bot)*dphidz
#     return output

def fluxFunDiff(dphiLdz,dphiSdz,iz,phiL,phiS,dudz):
    """
    Diffusion flux function, more modifications to be applied
    """
    output = np.zeros([2,1])
    "pair-wise function"
    global D
    global C
    global HP0
    global zBot
    BTemp= segB(dudz)
    z = zBot + (iz+0.5)*dz
    output[0] = (D/C)*dphiLdz + (1/C)*(HP0 + (1-z))*(1 + BTemp[0]*phiS + BTemp
        [1]*(1-phiS-phiL))*dphiLdz
    output[1] = (D/C)*dphiSdz + (1/C)*(HP0 + (1-z))*(1 - BTemp[0]*phiL - BTemp
        [2]*(1-phiS-phiL))*dphiSdz

    "quotient function"
    # global sizeRatioLS, sizeRatioLM, sizeRatioMS, sizeRatioDim
    # botL = phiL + (1/sizeRatioLS)**sizeRatioDim*phiS + (1/sizeRatioLM)**
        sizeRatioDim*(1-phiL-phiS)
    # botS = phiS + sizeRatioLS**sizeRatioDim*phiL + sizeRatioMS**sizeRatioDim*(1-
        phiL-phiS)
    # up5 = ((1/sizeRatioLS)**sizeRatioDim - (1/sizeRatioLM)**sizeRatioDim)*phiS +
         (1/sizeRatioLM)**sizeRatioDim
    # up6 = (sizeRatioLS**sizeRatioDim - sizeRatioMS**sizeRatioDim)*phiL +
        sizeRatioMS**sizeRatioDim
    # output[0] = (D/C)*dphiLdz + (1/C)*(HP0 + (1-z))*(up5/(botL**2))*dphiLdz
    # output[1] = (D/C)*dphiSdz + (1/C)*(HP0 + (1-z))*(up6/(botS**2))*dphiSdz
    return output

# def fluxFunDiff(dphidz):
#     """
#     Diffusion flux function, more modifications to be applied
#     """
#     global T
#     global d
#     global H
#     global c
#     output = (T*d)/(H*c) * dphidz
#     return output

# def p(z):
```

```
#     global zTop
#     global rho
#     global P0
#     output = rho*9.81*(zTop-z) + P0
#     return output

# def u(z,alpha,beta):
#     """
#     Prescribed bulk velocity profile, with its shape inspired by experimental
    results.
#     """
#     global uAmp
#     # global alpha
#     # global beta
#     L = 1 - 1/alpha
#     norm = np.tanh(beta/np.pi) - 1
#     if z >= L and z<= 1:
#         output = uAmp * (np.tanh(beta/(np.pi*alpha*(z-L)))-1)/norm
#     else:
#         output = 0
#     return output

def u(z,R,velTop):
    """
    Prescribed bulk velocity profile, with its shape inspired by experimental
        results.
    """
    # global alpha
    # global beta
    norm = np.tanh(R/np.pi) - 1
    norm = norm/velTop
    output = np.sign(R)*(np.tanh(R/(np.pi*z))-1)/norm

    return output

# def getList(alphaList, betaList, z):
#     # global alpha
#     # global beta
#     """
#     Get incremental values to be iterated and superposed for initialization
    process.
#     """
#     uNum = len(alphaList) + len(betaList)
#     uList = np.zeros([len(z),uNum])
#     "reverse the order of vector for top-down archiving"
#     # zRev = z[::-1]
#     # zRev = z

#     "loop over alpha list to record full evolution of velocity profile"
#     uI = 0
#     for alphaI in range(len(alphaList)):
#         alphaTemp = alphaList[alphaI]
#         betaTemp = betaList[0]
#         uTemp = np.zeros([len(z),1])
#         for i in range(len(z)):
#             alpha = alphaTemp
#             beta = betaTemp
#             # uTemp[i] = u(zRev[i],alpha,beta)
#             uTemp[i] = u(z[i],alpha,beta)
```

```
#                 uList[i,uI] = uTemp[i]

#             uI = uI + 1

#         "loop over beta list to record full evolution of velocity profile"
#         uI = 0
#         for betaI in range(len(betaList)):
#             alphaTemp = 1
#             betaTemp = betaList[betaI]
#             uTemp = np.zeros([len(z),1])
#             for i in range(len(z)):
#                 alpha = alphaTemp
#                 beta = betaTemp
#                 # uTemp[i] = u(zRev[i],alpha,beta)
#                 uTemp[i] = u(z[i],alpha,beta)
#                 uList[i,uI + len(alphaList)] = uTemp[i]

#             uI = uI + 1
#         return uList

# def getIncrements(alphaList, betaList, z, uList):
#         """
#         Take incremented values of velocity profiles based on given uList.
#         """
#         uNum = len(alphaList) + len(betaList)
#         duList = np.zeros([len(z),uNum])
#         duList[:,0] = uList[:,0]
#         for uI in range(1,uNum):
#             for i in range(len(z)):
#                 duList[i,uI] = np.subtract(uList[i,uI], uList[i,uI-1])
#         return duList

# ================================================================================
# KT scheme function
# ================================================================================
def getRHS(phiL,phiS):
    """
    Semi-discrete solver that converts the PDE to a time-dependent ODE for
    each time-step. Right-hand-side value of the ODE dphidt = RHS is outputed
    along with other useful information.
    """
    global nZ, nX, dx, dz
    global direction, uTempCell, uTempGrdpt
    global zTop

    """
    Define the local slopes in x and z directions.
    Stored at cells.
    """
    phiLX = np.zeros([nZ,nX])
    phiSX = np.zeros([nZ,nX])
    phiLZ = np.zeros([nZ,nX])
    phiSZ = np.zeros([nZ,nX])

    "Note: due to the zero initialization,"
    "phiX = 0 at x = {xStart,xEnd} and"
    "phiZ = 0 at z = {zBot,zTop} are automatically satisfied."
    "Therefore it remains to set other local slope entries "
    "using minmod limiter function."
```

```
for i in range(nZ):
    for j in range(1,nX-1):
        leftL = (phiL[i,j] - phiL[i,j-1])/dx
        rightL = (phiL[i,j+1] - phiL[i,j])/dx
        phiLX[i,j] = minmod(leftL, rightL)

        leftS = (phiS[i,j] - phiS[i,j-1])/dx
        rightS = (phiS[i,j+1] - phiS[i,j])/dx
        phiSX[i,j] = minmod(leftS, rightS)

for i in range(1,nZ-1):
    for j in range(nX):
        downL = (phiL[i,j] - phiL[i-1,j])/dz
        upL = (phiL[i+1,j] - phiL[i,j])/dz
        phiLZ[i,j] = minmod(downL, upL)

        downS = (phiS[i,j] - phiS[i-1,j])/dz
        upS = (phiS[i+1,j] - phiS[i,j])/dz
        phiSZ[i,j] = minmod(downS, upS)

"""
Define back and forward linear approximations of phi in x and z directions
based on local slopes.
Stored at Gridpoints.
"""
phiLL = np.zeros([nZ,nX+1])
phiLR = np.zeros([nZ,nX+1])
phiLD = np.zeros([nZ+1,nX])
phiLU = np.zeros([nZ+1,nX])

phiSL = np.zeros([nZ,nX+1])
phiSR = np.zeros([nZ,nX+1])
phiSD = np.zeros([nZ+1,nX])
phiSU = np.zeros([nZ+1,nX])

"Note: linear approximations at the boundary endpoints are assumed to be 0."
"This is because one of their neighbours is non-existant. This BC is "
"automatically satisfied via zero initialization."
for i in range(nZ):
    for j in range(nX):
        "left(x-) linear approximation based on phiX"
        phiLL[i,j+1] = phiL[i,j] + phiLX[i,j]*dx*0.5
        phiSL[i,j+1] = phiS[i,j] + phiSX[i,j]*dx*0.5
        "right(x+) linear approximation based on phiX"
        phiLR[i,j] = phiL[i,j] - phiLX[i,j]*dx*0.5
        phiSR[i,j] = phiS[i,j] - phiSX[i,j]*dx*0.5
        "down(z-) linear approximation based on phiZ"
        phiLD[i+1,j] = phiL[i,j] + phiLZ[i,j]*dz*0.5
        phiSD[i+1,j] = phiS[i,j] + phiSZ[i,j]*dz*0.5
        "up(z+) linear approximation based on phiZ"
        phiLU[i,j] = phiL[i,j] - phiLZ[i,j]*dz*0.5
        phiSU[i,j] = phiS[i,j] - phiSZ[i,j]*dz*0.5

"""
Record the downstream velocity component values at cells and gridpoints
in order to calculate du/dz.
"""
uValCell = np.zeros([nZ,1])
uValGrdpt = np.zeros([nZ+1,1])
```

```
for i in range(nZ):
    uValCell[i] = uTempCell[i]

for i in range(nZ+1):
    uValGrdpt[i] = uTempGrdpt[i]

"""
Use minmod limiter to numerically calculate magnitude of local velocity
gradients du/dz at cells and gridpoints.
"""
dudzCell = np.zeros([nZ,1])
dudzGrdpt = np.zeros([nZ+1,1])

for i in range(1,nZ-1):
    L = (uValCell[i] - uValCell[i-1])/dz
    R = (uValCell[i+1] - uValCell[i])/dz
    dudzCell[i] = np.abs(minmod(L,R))

for i in range(1,nZ):
    L = ( uValGrdpt[i] -  uValGrdpt[i-1])/dz
    R = ( uValGrdpt[i+1] -  uValGrdpt[i])/dz
    dudzGrdpt[i] = np.abs(minmod(L,R))

"""
Define the maximum local speeds aX and aZ at cell endpoints in x and z
directions. Again additional row/col is required.
"""
aX = np.zeros([nZ,nX+1])
aZ = np.zeros([nZ+1,nX])

for i in range(nZ):
    for j in range(nX+1):
        aL = np.abs(fluxFunXdPhi(i))
        aR = np.abs(fluxFunXdPhi(i))
        aX[i,j] = max(aL, aR)

for i in range(nZ+1):
    for j in range(nX):
        # aU = np.abs(fluxFunZdPhi(phiU[i,j],dudzGrdpt[i]))
        # aD = np.abs(fluxFunZdPhi(phiD[i,j],dudzGrdpt[i]))
        aU = getRho(fluxFunZJacobian(phiLU[i,j],phiSU[i,j],dudzGrdpt[i]))
        aD = getRho(fluxFunZJacobian(phiLD[i,j],phiSD[i,j],dudzGrdpt[i]))
        aZ[i,j] = max(aU, aD)

"""
Define the numerical flux terms Hx and Hz
in x and z directions stored at cell endpoints.
"""
HXL = np.zeros([nZ,nX+1])
HZL = np.zeros([nZ+1,nX])

HXS = np.zeros([nZ,nX+1])
HZS = np.zeros([nZ+1,nX])

"First prescribe the inflow and outflow BCs in x, now periodic."
for i in range(nZ):
    if direction == 1:
        inflow = fluxFunX(phiL[i,nX-1],phiS[i,nX-1], i)
```

```
            HXL[i,0] = inflow[0]
            HXS[i,0] = inflow[1]

            outflow = fluxFunX(phiL[i,nX-1],phiS[i,nX-1],i)
            HXL[i,nX] = outflow[0]
            HXS[i,nX] = outflow[1]
        else:
            inflow = fluxFunX(phiL[i,1],phiS[i,1], i)
            HXL[i,nX] = inflow[0]
            HXS[i,nX] = inflow[1]

            outflow = fluxFunX(phiL[i,1],phiS[i,1],i)
            HXL[i,0] = outflow[0]
            HXS[i,0] = outflow[1]

        for j in range(1,nX):
            phiDiff = np.zeros([2,1])
            phiDiff[0] = phiLR[i,j] - phiLL[i,j]
            phiDiff[1] = phiSR[i,j] - phiSL[i,j]

            H = 0.5*(fluxFunX(phiLR[i,j],phiSR[i,j],i) + fluxFunX(phiLL[i,j],phiSL
                [i,j],i)) \
                    -0.5*aX[i,j] * phiDiff

            HXL[i,j] = H[0]
            HXS[i,j] = H[1]

    "First prescribe top and bottom no-flux BC in z."
    for j in range(nX):
        HZL[0,j] = 0
        HZS[0,j] = 0
        HZL[nZ,j] = 0
        HZS[nZ,j] = 0

        for i in range(1,nZ):
            phiDiff = np.zeros([2,1])
            phiDiff[0] = phiLU[i,j] - phiLD[i,j]
            phiDiff[1] = phiSU[i,j] - phiSD[i,j]

            H = 0.5*(fluxFunZ(phiLU[i,j],phiSU[i,j], dudzCell[i]) + fluxFunZ(phiLD
                [i,j],phiSD[i,j], dudzCell[i])) \
                    -0.5* aZ[i,j] * phiDiff

            HZL[i,j] = H[0]
            HZS[i,j] = H[1]

    """
    Define the numerical diffusion flux terms.
    Stored at gridpoints.
    """
    PZL = np.zeros([nZ+1,nX])
    PZS = np.zeros([nZ+1,nX])

    for j in range(nX):
        for i in range(1,nZ):
            # zUp = zTop - (i-0.5)*dz
            # zDown = zTop - (i+0.5)*dz

            dphiLdz = (phiL[i,j] - phiL[i-1,j])/dz
```

```
            dphiSdz = (phiS[i,j] - phiS[i-1,j])/dz

            P = fluxFunDiff(dphiLdz,dphiSdz,i,phiL[i,j],phiS[i,j],dudzCell[i])
            PZL[i,j] = P[0]
            PZS[i,j] = P[1]

    RHSL = np.zeros([nZ,nX])
    RHSS = np.zeros([nZ,nX])

    for i in range(nZ):
        for j in range(nX):
            RHSL[i,j] = -(HXL[i,j+1] - HXL[i,j])/dx \
                        -(HZL[i+1,j] - HZL[i,j])/dz \
                        +(PZL[i+1,j] - PZL[i,j])/dz

            RHSS[i,j] = -(HXS[i,j+1] - HXS[i,j])/dx \
                        -(HZS[i+1,j] - HZS[i,j])/dz \
                        +(PZS[i+1,j] - PZS[i,j])/dz

    return RHSL,RHSS

def rk_timestep(phiL,phiS):
    """
    Runge-Kutta timestepper to solve the ODE for each timestep.
    """
    global nX, nZ, dt
    eta = 0.5
    phiTempL = np.zeros([nZ,nX])
    phiTempS = np.zeros([nZ,nX])

    phiNewL = np.zeros([nZ,nX])
    phiNewS = np.zeros([nZ,nX])


    RHSL,RHSS = getRHS(phiL,phiS)
    for i in range(nZ):
        for j in range(nX):
            phiTempL[i,j]  = phiL[i,j] + dt*RHSL[i,j]
            phiTempS[i,j]  = phiS[i,j] + dt*RHSS[i,j]

    RHSTempL,RHSTempS = getRHS(phiTempL,phiTempS)
    for i in range(nZ):
        for j in range(nX):
            phiNewL[i,j] = eta*phiL[i,j] + (1-eta) * (phiTempL[i,j] + dt*RHSTempL[
                i,j])
            phiNewS[i,j] = eta*phiS[i,j] + (1-eta) * (phiTempS[i,j] + dt*RHSTempS[
                i,j])
    return phiNewL, phiNewS


# ========================================================================
# Main function
# ========================================================================

"Initial volume fraction condition"
# phi = np.concatenate((np.ones([int(nZ*0.5),nX])*0.1, np.ones([int(nZ*0.5),nX]),)
    )
phiL = np.ones([nZ,nX])*(1/3)
phiS = np.ones([nZ,nX])*(1/3)
```

```
# phiL = np.concatenate([np.ones([int(nZ*0.3),nX]),np.zeros([int(nZ*0.7),nX])])
# phiS = np.concatenate([np.zeros([int(nZ*0.3),nX]),np.ones([int(nZ*0.5),nX]),np.
    zeros([int(nZ*0.2),nX])])
# phiM = np.ones([nZ,nX]) - phiL - phiS

"Generate velocity evolution data to iterate in time"
zCell = np.linspace(zBot,zTop-dz,nZ)+0.5*dz
# uListCell = getList(alphaList, betaList, zCell)
# duListCell = getIncrements(alphaList, betaList, zCell, uListCell)

uListCellHist = np.zeros([nZ,nT])
duListCellHist = np.zeros([nZ,nT])

# uListGrdpt = getList(alphaList, betaList, z)
# duListGrdpt = getIncrements(alphaList, betaList, z, uListGrdpt)

"Initialize matrix to store volume fraction and other data"
resultsL = np.zeros([nZ,nT+1])
# resultsM = np.zeros([nZ,nT+1])
resultsS = np.zeros([nZ,nT+1])

# Seg1L = np.zeros([nZ,nT])
# Seg2L = np.zeros([nZ,nT])
# Seg3L = np.zeros([nZ,nT])
# SegVelL = np.zeros([nZ,nT])

# Seg1M = np.zeros([nZ,nT])
# Seg2M = np.zeros([nZ,nT])
# Seg3M = np.zeros([nZ,nT])
# SegVelM = np.zeros([nZ,nT])

# Seg1S = np.zeros([nZ,nT])
# Seg2S = np.zeros([nZ,nT])
# Seg3S = np.zeros([nZ,nT])
# SegVelS = np.zeros([nZ,nT])

# SegFluxL = np.zeros([nZ,nT])
# # SegFluxM = np.zeros([nZ,nT])
# SegFluxS = np.zeros([nZ,nT])

# DiffFluxL = np.zeros([nZ,nT])
# # DiffFluxM = np.zeros([nZ,nT])
# DiffFluxS = np.zeros([nZ,nT])

"Record IC in results"
for i in range(nZ):
    resultsL[i,0] = 0.5*(phiL[i,int(nX/2)] + phiL[i,int(nX/2)+1])
    resultsS[i,0] = 0.5*(phiS[i,int(nX/2)] + phiS[i,int(nX/2)+1])

"Index used to document evolution progress of velocity profile in time"
# indexI = 0
# indexR = len(alphaList) + len(betaList)

"Loop over time"
for t in range(nT):
    # "check whether the timestep is the start of the initialization period"
    # if indexI+1 <= len(alphaList)+len(betaList):
    #     for i in range(nZ):
    #         uTempCell[i] = uTempCell[i] + direction * duListCell[i,indexI]
```

```
#       for i in range(nZ+1):
#           uTempGrdpt[i] = uTempGrdpt[i] + direction * duListGrdpt[i,indexI]
#           "update the counter"
#       indexI = indexI + 1

# "check whether the timestep is the start of the reversal period"
# if t in n0Rev:
#       "reset the reversal progress counter and record change of direction"
#       indexR = 0
#       direction = -1*direction
#       for i in range(nZ):
#           uTempCell[i] = uTempCell[i] + 2 * direction * duListCell[i,indexR]
#       for i in range(nZ+1):
#           uTempGrdpt[i] = uTempGrdpt[i] + 2 * direction * duListGrdpt[i,indexR
      ]

#       "update the counter"
#       indexR = indexR + 1

# "check whether the timestep is within the reversal period"
# if t not in n0Rev and indexR+1 <= len(alphaList)+len(betaList):
#       "add the prescribed increment assigned to this step"
#       for i in range(nZ):
#           uTempCell[i] = uTempCell[i] + 2 * direction * duListCell[i,indexR]
#       for i in range(nZ+1):
#           uTempGrdpt[i] = uTempGrdpt[i] + 2 * direction * duListGrdpt[i,indexR
      ]

#       "update the counter"
#       indexR = indexR + 1
"Second version of velocity profile evolution"
RTemp = 5*np.cos(2*np.pi*t/T)+5.5
velTopTemp = np.sin(np.pi*t/T)
direction = np.sign(velTopTemp)
for i in range(nZ):
    zTemp = zBot + (i+0.5)*dz
    uTempCell[i] = u(zTemp,RTemp,velTopTemp)
for i in range(nZ+1):
    zTemp = zBot + i*dz
    uTempGrdpt[i] = u(zTemp,RTemp,velTopTemp)

"record velocity evolution data"
for i in range(nZ):
    uListCellHist[i,t] = uTempCell[i]

"record velocity evolution data"
for i in range(nZ):
    uListCellHist[i,t] = uTempCell[i]

"record shear rate data"
for i in range(1,nZ-1):
    dudzTemp = minmod((uTempCell[i] - uTempCell[i-1])/dz, (uTempCell[i+1] -
        uTempCell[i])/dz)
    duListCellHist[i,t] = dudzTemp

# uTempCell = np.ones([nZ,1])
# uTempGrdpt = np.ones([nZ+1,1])

"calculate phi for next timestep using KT solver"
```

```
phiNewL,phiNewS = rk_timestep(phiL,phiS)
phiL = phiNewL
phiS = phiNewS

"store the central column for evolution graph"
if np.mod(nX,2) == 0:
    for i in range(nZ):
        resultsL[i,t+1] = 0.5*(phiL[i,int(nX*0.5)] + phiL[i,int(nX*0.5)+1])
        resultsS[i,t+1] = 0.5*(phiS[i,int(nX*0.5)] + phiS[i,int(nX*0.5)+1])
        # resultsM[i,t] = 1- resultsL[i,t] - resultsS[i,t]

elif np.mod(nX,2) == 1:
    for i in range(nZ):
        resultsL[i,t+1] = phiL[i,(nX+1)*0.5]
        resultsS[i,t+1] = phiS[i,(nX+1)*0.5]
        # resultsM[i,t] = 1- resultsL[i,t] - resultsS[i,t]

# "Calculate and store the shear rate data"
# dudzCellTemp = np.zeros([nZ,1])
# for i in range(1,nZ-1):
#     L = (uTempCell[i] - uTempCell[i-1])/dz
#     R = (uTempCell[i+1] - uTempCell[i])/dz
#     dudzCellTemp[i] = np.abs(minmod(L,R))

# "calculate and store the segregation velocity normalized by sqrt(H/g)"
# for i in range(nZ):
#     zTemp = zTop - (i+0.5)*dz
#     phiTempL = resultsL[i,t]
#     phiTempM = resultsM[i,t]
#     phiTempS = resultsS[i,t]
#     if i == 0 or i == nZ-1:
#         dphidzTempL = 0
#     else:
#         dphidzTempL = minmod((resultsL[i,t] - resultsL[i-1,t])/dz, (resultsL
    [i+1,t] - resultsL[i,t])/dz)

#     if i == 0 or i == nZ-1:
#         dphidzTempM = 0
#     else:
#         dphidzTempM = minmod((resultsM[i,t] - resultsM[i-1,t])/dz, (resultsM
    [i+1,t] - resultsM[i,t])/dz)

#     if i == 0 or i == nZ-1:
#         dphidzTempS = 0
#     else:
#         dphidzTempS = minmod((resultsS[i,t] - resultsS[i-1,t])/dz, (resultsS
    [i+1,t] - resultsS[i,t])/dz)

#     BTemp = segB(dudzCellTemp[i])
#     Seg1L[i,t] = 9.81/(C*T) * (BTemp[0]*phiTempS + BTemp[1]*phiTempM)
#     Seg1M[i,t] = 9.81/(C*T) * (BTemp[1]*phiTempL + BTemp[2]*phiTempS)
#     Seg1S[i,t] = 9.81/(C*T) * (BTemp[0]*phiTempL + BTemp[2]*phiTempM)

#     Seg2L[i,t] = -(D/C)*(1/phiTempL)*dphidzTempL
#     Seg2M[i,t] = -(D/C)*(1/phiTempM)*dphidzTempM
#     Seg2S[i,t] = -(D/C)*(1/phiTempS)*dphidzTempS

#     Seg3L[i,t] = -1/(C*T*9.81)*(HP0 + (1-zTemp))*(1 + BTemp[0]*phiTempS +
    BTemp[1]*phiTempM)/phiTempL*dphidzTempL
```

```
    #       Seg3M[i,t] = -1/(C*T*9.81)*(HP0 + (1-zTemp))*(1 + BTemp[1]*phiTempL +
        BTemp[2]*phiTempS)/phiTempM*dphidzTempM
    #       Seg3S[i,t] = -1/(C*T*9.81)*(HP0 + (1-zTemp))*(1 + BTemp[0]*phiTempL +
        BTemp[2]*phiTempM)/phiTempS*dphidzTempS

    #       SegVelL[i,t] = Seg1L[i,t] + Seg2L[i,t] + Seg3L[i,t]
    #       SegVelM[i,t] = Seg1M[i,t] + Seg2M[i,t] + Seg3M[i,t]
    #       SegVelL[i,t] = Seg1S[i,t] + Seg2S[i,t] + Seg3S[i,t]

    # "calculate and store the magnitude of segregation flux"
    # for i in range(nZ):
    #       phiTempL = resultsL[i,t]
    #       phiTempL = resultsS[i,t]
    #       SegFluxTemp = fluxFunZ(phiTempL, phiTempS, dudzCellTemp[i])
    #       SegFluxL[i,t] = SegFluxTemp[0]
    #       SegFluxS[i,t] = SegFluxTemp[1]


    # "calculate and store the magnitude of diffusion flux"

    # for i in range(nZ):
    #       zTemp = zTop - (i+0.5)*dz

    #       phiTempL = resultsL[i,t]
    #       phiTempM = resultsM[i,t]
    #       phiTempS = resultsS[i,t]

    #       if i == 0 or i == nZ-1:
    #           dphidzTempL = 0
    #       else:
    #           dphidzTempL = minmod((resultsL[i,t] - resultsL[i-1,t])/dz, (resultsL
        [i+1,t] - resultsL[i,t])/dz)

    #       if i == 0 or i == nZ-1:
    #           dphidzTempM = 0
    #       else:
    #           dphidzTempM = minmod((resultsM[i,t] - resultsM[i-1,t])/dz, (resultsM
        [i+1,t] - resultsM[i,t])/dz)

    #       if i == 0 or i == nZ-1:
    #           dphidzTempS = 0
    #       else:
    #           dphidzTempS = minmod((resultsS[i,t] - resultsS[i-1,t])/dz, (resultsS
        [i+1,t] - resultsS[i,t])/dz)


    #       DiffFluxTemp = fluxFunDiff(dphidzTempL,dphidzTempS,zTemp,phiTempL,
        phiTempS,dudzCellTemp[i])
    #       DiffFluxL[i,t] = DiffFluxTemp[0]
    #       DiffFluxS[i,t] = DiffFluxTemp[1]

    "counter"
    print("timestep number ",t ,"completed")

"create the figure and axes objects"
# fig, axes =  plt.subplots(1,1,figsize = (5,5))

"Plot the evolution graph"
fig, axes =  plt.subplots(1,1,figsize = (5,5))
```

```
plt.imshow(np.flipud(resultsL), interpolation='none', cmap='viridis',aspect=nT/nZ)
plt.clim(0.25,0.4)
plt.colorbar()

# fig, axes =  plt.subplots(1,1,figsize = (5,5))
# plt.imshow(np.flipud(resultsM), interpolation='none', cmap='viridis',aspect=nT/
    nZ)
# plt.clim(0,1)
# plt.colorbar()

fig, axes =  plt.subplots(1,1,figsize = (5,5))
plt.imshow(np.flipud(resultsS), interpolation='none', cmap='viridis',aspect=nT/nZ)
plt.clim(0.25,0.4)
plt.colorbar()

plt.show()

"Save the evolution data"
np.savetxt("results0_10_P0_0.5_spread_real2a1_tri.csv", resultsL, delimiter=",")
np.savetxt("results2_10_P0_0.5_spread_real2a1_tri.csv", resultsS, delimiter=",")
# np.savetxt("resultsS.csv", resultsS, delimiter=",")

# np.savetxt("SegVelL.csv", SegVelL, delimiter=",")
# np.savetxt("SegVelM.csv", SegVelM, delimiter=",")
# np.savetxt("SegVelS.csv", SegVelS, delimiter=",")

# np.savetxt("SegFluxL.csv", SegFluxL, delimiter=",")
# # np.savetxt("SegFluxM.csv", SegFluxM, delimiter=",")
# np.savetxt("SegFluxS.csv", SegFluxS, delimiter=",")

# np.savetxt("DiffFluxL.csv", DiffFluxL, delimiter=",")
# # np.savetxt("DiffFluxM.csv", DiffFluxM, delimiter=",")
# np.savetxt("DiffFluxS.csv", DiffFluxS, delimiter=",")
```

Appendix C

Python code for iterative expansion poly-disperse simulation

```
# -*- coding: utf-8 -*-
"""
Created on Thu Nov 10 10:31:51 2022

@author: Jiaxin Zhang
"""

import time as tm
import numpy as np
import sympy as sym
from numpy import linalg as LA
import matplotlib.pyplot as plt

"DEFINE THE PARAMETERS"
"-----------------------------------------------------------------"

"Population prescription parameters"
sizes = np.append([1],np.linspace(1.5,3,num=9))
sizes = np.linspace(1.5,1.0,num=2)
# temp = sizes[3]
# sizes[3] = sizes[2]
# sizes[2] = temp
symbols = ['phi1','phi2','phi3','phi4','phi5','phi6','phi7','phi8','phi9','phi10']
phaseNum = 2

"Key convection-diffusion parameters"
C = 2
D = 0.2
HP0 = 0.5
```

```
T = 1

"Spatial and temporal discretisation parameters"
nX = 10
nZ = 20
nT = 5000
xStart = 0
xEnd = 5
zBot = 0
zTop = 1
tEnd = 5
dx = (xEnd - xStart)/nX
dz = (zTop - zBot)/nZ
dt = tEnd/(nT)
x = np.linspace(xStart,xEnd,nX+1)
z = np.linspace(zBot,zTop,nZ+1)
time = np.linspace(0,tEnd,nT+1)

"Velocity profile evolution parameters (initiation + reversal)"
# direction = 1
# t0Rev = np.array([100])
# tRev = T*0.5
# n0Rev = nT/tEnd*t0Rev + 1
# nRev = nT/tEnd*tRev
# uAmp = 1
# alpha = np.linspace(3, 1.04, int(nRev*0.8))
# alphaList = np.multiply(alpha,alpha)
# betaList = np.linspace(8, 1, int(nRev*0.5))
uTempCell = np.zeros([nZ,1])
uTempGrdpt = np.zeros([nZ+1,1])

"Breakage-related parameters"
tBreak = np.array([100])
# tBreak = np.linspace(0.005,3.995,799)
# tBreak = np.linspace(0.05,3.95,79)
nBreak = nT/tEnd*tBreak + 1
nBreak = nBreak.astype(int)
# breakPlan = np.array([0.1,0.05,0.01])
breakPlan = np.array([0.01,0.01,0.01])

"ITERATION-RELATED FUNCTIONS"
"----------------------------------------------------------------"

def pairingAdd(matrix,phaseNum):
    # global phaseNum
    lastcols = matrix[:,[2*(phaseNum-1)-2,2*(phaseNum-1)-1]]
    newrow = np.zeros(2*(phaseNum-1))
    for i in range(phaseNum-1):
        newrow[2*i] = lastcols[i,1]
        newrow[2*i+1] = lastcols[i,0]
    step1 = np.vstack([matrix, newrow])
    newcol1 = np.arange(1,phaseNum+1)
    newcol2 = np.ones(phaseNum)*(phaseNum+1)
    newrows = np.c_[newcol1,newcol2]
    pairingMatrixT = np.c_[step1,newrows]
    "update the current phase number"
    phaseNum = phaseNum+1

    return pairingMatrixT
```

```
def sizeRatioRelation(x):
    output = 0.2 * ( np.power(x-4,3) + 3*np.power(x-4,2) )
    return output

def updateSizeRatios(pairingMatrix):
    global phaseNum, sizes
    "update size ratio matrix"
    sizeRatios = np.zeros([phaseNum-1,phaseNum-1])
    for i in range(phaseNum-1):
       for j in range(phaseNum-1):
           sizeInd1 = int(pairingMatrix[i,2*j])
           sizeInd2 = int(pairingMatrix[i,2*j+1])
           sizeRatios[i,j] = sizes[sizeInd1-1]/sizes[sizeInd2-1]
    return sizeRatios

def updateSegBs(pairingMatrix,sizeRatios):
    global phaseNum
    "update segregation rate matrix"
    segBs = np.zeros([phaseNum-1,phaseNum-1])
    for i in range(phaseNum-1):
        for j in range(phaseNum-1):
            ratio = sizeRatios[i,j]
            if ratio >=1:
                segBs[i,j] = sizeRatioRelation(ratio)
            if ratio <1:
                segBs[i,j] = -sizeRatioRelation(1/ratio)
    return segBs

"DEFAULT FUNCTIONS"
"-----------------------------------------------------------------"

def fluxFunX(phis,iZ):
    """
    Flux function in x-direction, collects velocity profile via global.
    input: a list of phi (e.g. [phi1,phi2]) of size phaseNum-1
    """
    global uTempCell, phaseNum

    output = np.zeros([phaseNum-1,1])
    for k in range(phaseNum-1):
        output[k] = phis[k] * uTempCell[iZ]
    return output

def fluxFunXdPhi(iZ):
    """
    Derivative of flux function in x-direction.
    """
    global uTempCell
    output = uTempCell[iZ]
    return output

def fluxFunZp(phisInput):
    """
    Flux function in z-direction, with prescribed dependency on velocity gradient.
    input: a list of phi (e.g. [phi1,phi2]) of size phaseNum-1
    """
    global C, segBs, phaseNum, pairingMatrix
    "first calculate solid volume fraction of remainder by mass conservation"
```

```
    phiSum = 0
    phisTemp = [[] for i in range(phaseNum)]
    for k in range(phaseNum-1):
        phisTemp[k] = phisInput[k]
        phiSum = phiSum + phisInput[k]
    lastPhi = 1 - phiSum
    phisTemp[phaseNum-1] = lastPhi

    pairsTemp = np.zeros([phaseNum-1,phaseNum-1])
    for i in range(phaseNum-1):
        for j in range(phaseNum-1):
            pairInd1 = int(pairingMatrix[i,2*j])
            pairInd2 = int(pairingMatrix[i,2*j+1])
            pairsTemp[i,j] = phisTemp[pairInd1-1]*phisTemp[pairInd2-1]
    "assemble all the terms to be added"
    terms = np.multiply((1/C)*segBs, pairsTemp)
    "add all the terms"
    output = np.sum(terms, axis=1)

    # bot = phi + s*(1-phi)
    # output = (1/C)*phi/bot
    return output

def fluxFunZJacobianp(phisInput,dudz):
    """
    Calculates Jacobian of the flux function in Z
    input: a list of phi (e.g. [phi1,phi2]) of size phaseNum-1
    """
    global phiSyms, polyDiff, phaseNum

    "For bi-disperse only"
    # phisInput = np.array([phisInput])

    "first calculate solid volume fraction of remainder by mass conservation"

    phiSum = 0
    phisTemp = [[] for i in range(phaseNum)]
    for k in range(phaseNum-1):
        phisTemp[k] = phisInput[k]
        phiSum = phiSum + phisInput[k]
    lastPhi = 1 - phiSum
    phisTemp[phaseNum-1] = lastPhi

    jacobian = np.zeros([phaseNum-1,phaseNum-1])
    for i in range(phaseNum-1):
        for j in range(phaseNum-1):
            replacements = []
            for k in range(phaseNum-1):
                replacements.append((phiSyms[k],phisTemp[k]))
            polyDiffTemp = polyDiff[i,j]
            jacobian[i,j] = polyDiffTemp.subs(replacements)

    return jacobian

def getRho(jacobian):
    w,v = LA.eig(jacobian)
    output = np.amax(np.abs(w))
    return output
```

```
def fluxFunDiffp(phisInput, dphidz, iZ):
    """
    Diffusion flux function, more modifications to be applied
    """
    global C, D, HP0, phaseNum, pairngMatrix
    global zBot, dz
    "pair-wise function"

    "first calculate solid volume fraction of remainder by mass conservation"
    phiSum = 0
    phisTemp = [[] for i in range(phaseNum)]
    for k in range(phaseNum-1):
        phisTemp[k] = phisInput[k]
        phiSum = phiSum + phisInput[k]
    lastPhi = 1 - phiSum
    phisTemp[phaseNum-1] = lastPhi

    pairsTemp = np.zeros([phaseNum-1,phaseNum-1])
    for i in range(phaseNum-1):
        for j in range(phaseNum-1):
            # pairInd1 = int(pairingMatrix[i,2*j])
            pairInd2 = int(pairingMatrix[i,2*j+1])
            pairsTemp[i,j] = phisTemp[pairInd2-1]
    "assemble all the terms to be added"
    terms = np.multiply(segBs, pairsTemp)
    "add all the terms"
    diffSums = np.sum(terms, axis=1)

    output = np.zeros([phaseNum-1,1])
    for k in range(phaseNum-1):
        diffTerm1 = (D/C)*dphidz[k]
        diffSum = diffSums[k]
        zTemp = zBot + (0.5+iZ)*dz
        diffTerm2 = (1/C)*(HP0 + (1-zTemp))*dphidz[k]*(1 + diffSum)
        # diffTerm2 = 0
        output[k] = diffTerm1 + diffTerm2


    return output

def minmod(a,b):
    """
    Returns the flatter local slope out of the given two.
    If the two slopes have different signs, then it returns 0.
    """
    output = 0.5*(np.sign(a)+np.sign(b)) * np.min([np.abs(a),np.abs(b)])
    return output

# def u(z,alpha,beta):
#     """
#     Prescribed bulk velocity profile, with its shape inspired by experimental
     results.
#     """
#     global uAmp
#     # global alpha
#     # global beta
#     L = 1 - 1/alpha
#     norm = np.tanh(beta/np.pi) - 1
#     if z >= L and z<= 1:
```

```
#           output = uAmp * (np.tanh(beta/(np.pi*alpha*(z-L)))-1)/norm
#       else:
#           output = 0
#       return output

def u(z,R,velTop):
    """
    Prescribed bulk velocity profile, with its shape inspired by experimental
        results.
    """
    # global alpha
    # global beta
    norm = np.tanh(R/np.pi) - 1
    norm = norm/velTop
    output = np.sign(R)*(np.tanh(R/(np.pi*z))-1)/norm

    return output

# def getList(alphaList, betaList, z):
#     # global alpha
#     # global beta
#     """
#     Get incremental values to be iterated and superposed for initialization
    process.
#     """
#     uNum = len(alphaList) + len(betaList)
#     uList = np.zeros([len(z),uNum])
#     "reverse the order of vector for top-down archiving"
#     # zRev = z[::-1]
#     # zRev = z

#     "loop over alpha list to record full evolution of velocity profile"
#     uI = 0
#     for alphaI in range(len(alphaList)):
#         alphaTemp = alphaList[alphaI]
#         betaTemp = betaList[0]
#         uTemp = np.zeros([len(z),1])
#         for i in range(len(z)):
#             alpha = alphaTemp
#             beta = betaTemp
#             # uTemp[i] = u(zRev[i],alpha,beta)
#             uTemp[i] = u(z[i],alpha,beta)
#             uList[i,uI] = uTemp[i]

#         uI = uI + 1

#     "loop over beta list to record full evolution of velocity profile"
#     uI = 0
#     for betaI in range(len(betaList)):
#         alphaTemp = 1
#         betaTemp = betaList[betaI]
#         uTemp = np.zeros([len(z),1])
#         for i in range(len(z)):
#             alpha = alphaTemp
#             beta = betaTemp
#             # uTemp[i] = u(zRev[i],alpha,beta)
#             uTemp[i] = u(z[i],alpha,beta)
#             uList[i,uI + len(alphaList)] = uTemp[i]
```

```
#           uI = uI + 1
#       return uList


# def getIncrements(alphaList, betaList, z, uList):
#       """
#       Take incremented values of velocity profiles based on given uList.
#       """
#       uNum = len(alphaList) + len(betaList)
#       duList = np.zeros([len(z),uNum])
#       duList[:,0] = uList[:,0]
#       for uI in range(1,uNum):
#           for i in range(len(z)):
#               duList[i,uI] = np.subtract(uList[i,uI], uList[i,uI-1])
#       return duList

"SOLVER"
"------------------------------------------------------------"

def getRHS(phis):
    """
    Semi-discrete solver that converts the PDE to a time-dependent ODE for
    each time-step. Right-hand-side value of the ODE dphidt = RHS is outputed
    along with other useful information.
    """
    global nZ, nX, dx, dz
    global direction, uTempCell, uTempGrdpt
    global zTop
    global phaseNum

    global phiC1
    phiC1 = phis[0]

    # """
    # Derive the last volume fraction via mass conservation condition
    # """
    # phiSum = 0
    # phisTemp = [[] for i in range(phaseNum)]
    # for k in range(phaseNum-1):
    #       phisTemp[k] = phis[k]
    #       phiSum = phiSum + phis[k]
    # lastPhi = 1 - phiSum
    # phisTemp[phaseNum-1] = lastPhi

    """
    Define the local slopes in x and z directions.
    Stored at cells.
    """
    phiXp = [[] for i in range(phaseNum-1)]
    phiZp = [[] for i in range(phaseNum-1)]
    for k in range(phaseNum-1):
        phiXp[k] = np.zeros([nZ,nX])
        phiZp[k] = np.zeros([nZ,nX])

    "Note: due to the zero initialization,"
    "phiX = 0 at x = {xStart,xEnd} and"
    "phiZ = 0 at z = {zBot,zTop} are automatically satisfied."
    "Therefore it remains to set other local slope entries "
    "using minmod limiter function."
```

```
for k in range(phaseNum-1):
    for i in range(nZ):
        for j in range(1,nX-1):
            left = (phis[k][i,j] - phis[k][i,j-1])/dx
            right = (phis[k][i,j+1] - phis[k][i,j])/dx
            phiXp[k][i,j] = minmod(left, right)


downs1 = [[] for i in range(phaseNum-1)]
ups1 = [[] for i in range(phaseNum-1)]
for k in range(phaseNum-1):
    downs1[k] = np.zeros([nZ,nX])
    ups1[k] = np.zeros([nZ,nX])

for k in range(phaseNum-1):
    for i in range(1,nZ-1):
        for j in range(nX):
            down1 = (phis[k][i,j] - phis[k][i-1,j])/dz
            up1 = (phis[k][i+1,j] - phis[k][i,j])/dz
            downs1[k][i,j] = down1
            ups1[k][i,j] = up1
            phiZp[k][i,j] = minmod(down1, up1)

"""
Define back and forward linear approximations of phi in x and z directions
based on local slopes.
Stored at Gridpoints.
"""
phiLp = [[] for i in range(phaseNum-1)]
phiRp = [[] for i in range(phaseNum-1)]
phiDp = [[] for i in range(phaseNum-1)]
phiUp = [[] for i in range(phaseNum-1)]
for k in range(phaseNum-1):
    phiLp[k] = np.zeros([nZ,nX+1])
    phiRp[k] = np.zeros([nZ,nX+1])
    phiDp[k] = np.zeros([nZ+1,nX])
    phiUp[k] = np.zeros([nZ+1,nX])

"Note: linear approximations at the boundary endpoints are assumed to be 0."
"This is because one of their neighbours is non-existant. This BC is "
"automatically satisfied via zero initialization."
for k in range(phaseNum-1):
    for i in range(nZ):
        for j in range(nX):
            "left(x-) linear approximation based on phiX"
            phiLp[k][i,j+1] = phis[k][i,j] + phiXp[k][i,j]*dx*0.5
            "right(x+) linear approximation based on phiX"
            phiRp[k][i,j] = phis[k][i,j] - phiXp[k][i,j]*dx*0.5
            "down(z-) linear approximation based on phiZ"
            phiDp[k][i+1,j] = phis[k][i,j] + phiZp[k][i,j]*dz*0.5
            "up(z+) linear approximation based on phiZ"
            phiUp[k][i,j] = phis[k][i,j] - phiZp[k][i,j]*dz*0.5

"""
Record the downstream velocity component values at cells and gridpoints
in order to calculate du/dz.
"""
uValCell = np.zeros([nZ,1])
uValGrdpt = np.zeros([nZ+1,1])
```

```
for i in range(nZ):
    uValCell[i] = uTempCell[i]

for i in range(nZ+1):
    uValGrdpt[i] = uTempGrdpt[i]

"""
Use minmod limiter to numerically calculate magnitude of local velocity
gradients du/dz at cells and gridpoints.
"""
dudzCell = np.zeros([nZ,1])
dudzGrdpt = np.zeros([nZ+1,1])

for i in range(1,nZ-1):
    L1 = (uValCell[i] - uValCell[i-1])/dz
    R1 = (uValCell[i+1] - uValCell[i])/dz
    dudzCell[i] = np.abs(minmod(L1,R1))

for i in range(1,nZ):
    L2 = ( uValGrdpt[i] -  uValGrdpt[i-1])/dz
    R2 = ( uValGrdpt[i+1] -  uValGrdpt[i])/dz
    dudzGrdpt[i] = np.abs(minmod(L2,R2))

"""
Define the maximum local speeds aX and aZ at cell endpoints in x and z
directions. Again additional row/col is required.
"""
# aUa = [[] for i in range(phaseNum-1)]
# aDa = [[] for i in range(phaseNum-1)]
# aX = [[] for i in range(phaseNum-1)]
# aZ = [[] for i in range(phaseNum-1)]
# for k in range(phaseNum-1):
#     aUa[k] = np.zeros([nZ+1,nX])
#     aDa[k] = np.zeros([nZ+1,nX])
#     aX = np.zeros([nZ,nX+1])
#     aZ = np.zeros([nZ+1,nX])

aUp = np.zeros([nZ+1,nX])
aDp = np.zeros([nZ+1,nX])
aXp = np.zeros([nZ,nX+1])
aZp = np.zeros([nZ+1,nX])

for i in range(nZ):
    for j in range(nX+1):
        for k in range(phaseNum-1):
            aL = np.abs(fluxFunXdPhi(i))
            aR = np.abs(fluxFunXdPhi(i))
            aXp[i,j] = np.max([aL, aR])

for i in range(nZ+1):
    for j in range(nX):
        phiSampleU = np.zeros(phaseNum-1)
        phiSampleD = np.zeros(phaseNum-1)
        for k in range(phaseNum-1):
            phiSampleU[k] = phiUp[k][i,j]
            phiSampleD[k] = phiDp[k][i,j]

        aUp[i,j] = getRho(fluxFunZJacobianp(phiSampleU,dudzGrdpt[i]))
```

```
            aDp[i,j] = getRho(fluxFunZJacobianp(phiSampleD,dudzGrdpt[i]))
            aZp[i,j] = np.max([aUp[i,j], aDp[i,j]])

    """
    Define the numerical flux terms Hx and Hz
    in x and z directions stored at cell endpoints.
    """
    HXp = []
    HZp = []
    for k in range(phaseNum-1):
        HXp.append(np.zeros([nZ,nX+1]))
        HZp.append(np.zeros([nZ+1,nX]))

    "First prescribe the inflow and outflow BCs in x, now periodic."
    for i in range(nZ):
        for k in range(phaseNum-1):
            if direction == 1:
                phiSample = np.zeros([phaseNum-1,1])
                for k1 in range(phaseNum-1):
                    phiSample[k1] = phis[k1][i,nX-1]
                inflow = fluxFunX(phiSample, i)
                HXp[k][i,0] = inflow[k,0]
                HXp[k][i,nX] = inflow[k,0]
            else:
                phiSample1 = np.zeros([phaseNum-1,1])
                for k1 in range(phaseNum-1):
                    phiSample1[k1] = phis[k1][i,1]
                inflow = fluxFunX(phiSample1, i)
                HXp[k][i,nX] = inflow[k,0]
                HXp[k][i,0] = inflow[k,0]

            for j in range(1,nX):
                phiLsample = np.zeros([phaseNum-1,1])
                phiRsample = np.zeros([phaseNum-1,1])
                for k1 in range(phaseNum-1):
                    # phiRsample[k] = phiL[k][i,j]
                    # phiLsample[k] = phiR[k][i,j]
                    phiLsample[k1,0] = phiLp[k1][i,j]
                    phiRsample[k1,0] = phiRp[k1][i,j]

                HXp[k][i,j] = 0.5*(fluxFunX(phiRsample,i)[k] \
                                + fluxFunX(phiLsample,i)[k]) \
                                -0.5 * aXp[i,j] * (phiRsample[k] - phiLsample[k
                                    ])

    "First prescribe top and bottom no-flux BC in z."
    for j in range(nX):
        for k in range(phaseNum-1):
            HZp[k][0,j] = 0
            HZp[k][nZ,j] = 0

            for i in range(1,nZ):
                phiUsample = np.zeros([phaseNum-1,1])
                phiDsample = np.zeros([phaseNum-1,1])
                for k1 in range(phaseNum-1):
                    phiUsample[k1] = phiUp[k1][i,j]
                    phiDsample[k1] = phiDp[k1][i,j]

                HZp[k][i,j] = 0.5*(fluxFunZp(phiUsample)[k] \
```

```
                                + fluxFunZp(phiDsample)[k]) \
                            -0.5* aZp[i,j] * (phiUsample[k] - phiDsample[k])

    """
    Define the numerical diffusion flux terms.
    Stored at gridpoints.
    """
    PZp = []
    for k in range(phaseNum-1):
        PZp.append(np.zeros([nZ+1,nX]))
    for j in range(nX):
        for i in range(1,nZ):
            dphidzs = np.zeros([phaseNum-1,1])
            for k in range(phaseNum-1):
                phiSample = np.zeros([phaseNum-1,1])
                for k1 in range(phaseNum-1):
                    phiSample[k1] = phis[k1][i,j]

                dphidzs[k] = (phis[k][i,j] - phis[k][i-1,j])/dz

                PZp[k][i,j] = fluxFunDiffp(phiSample,dphidzs,i)[k]
                # PZ[k][i,j] = 0

    RHSp = []
    for k in range(phaseNum-1):
        RHSp.append(np.zeros([nZ,nX]))

    for i in range(nZ):
        for j in range(nX):
            for k in range(phaseNum-1):
                RHSp[k][i,j] = -(HXp[k][i,j+1] - HXp[k][i,j])/dx \
                               -(HZp[k][i+1,j] - HZp[k][i,j])/dz \
                               +(PZp[k][i+1,j] - PZp[k][i,j])/dz
    return RHSp


def rk_timestep(phis):
    """
    Runge-Kutta timestepper to solve the ODE for each timestep.
    """
    global nX, nZ, dt
    global phaseNum
    eta = 0.5
    phiTemp = [[] for i in range(phaseNum-1)]
    for k in range(phaseNum-1):
        phiTemp[k] = np.zeros([nZ,nX])
    phiNew= [[] for i in range(phaseNum-1)]
    for k in range(phaseNum-1):
        phiNew[k] = np.zeros([nZ,nX])

    RHS = getRHS(phis)
    for i in range(nZ):
        for j in range(nX):
            for k in range(phaseNum-1):
                phiTemp[k][i,j]  = phis[k][i,j] + dt*RHS[k][i,j]

    RHSTemp = getRHS(phiTemp)
    for i in range(nZ):
        for j in range(nX):
```

```
            for k in range(phaseNum-1):
                phiNew[k][i,j] = eta*phis[k][i,j] + (1-eta) * (phiTemp[k][i,j] +
                    dt*RHSTemp[k][i,j])

    return phiNew

"MAIN"
"------------------------------------------------------------"
# start = tm.time()

sizeRatios = np.zeros([phaseNum-1,phaseNum-1])
segBs = np.zeros([phaseNum-1,phaseNum-1])

pairingMatrix = np.zeros([1, 2])
pairingMatrix[0,0] = 1
pairingMatrix[0,1] = 2
for i in range(phaseNum-2):
    phaseNumTemp = i+2
    pairingMatrix = pairingAdd(pairingMatrix, phaseNumTemp)

phis = [[] for i in range(phaseNum)]
for k in range(phaseNum):
    phis[k] = np.zeros([nZ,nX])
    sizeRatios = updateSizeRatios(pairingMatrix)
    segBs = updateSegBs(pairingMatrix,sizeRatios)


"generate symbolic variables to store pairwise polynomials"
global phiSyms
phiSyms = []
for k in range(phaseNum):
    symTemp = sym.Symbol(symbols[k])
    phiSyms.append(symTemp)

"genarate symbolic polynomials w.r.t. flux functions"
polys = []
for i in range(phaseNum-1):
    row = 0
    phiSumSyms = 0
    for k in range(phaseNum-1):
        phiSumSyms = phiSumSyms + phiSyms[k]

    for j in range(phaseNum-2):
        pairInd1 = int(pairingMatrix[i,2*j] - 1)
        pairInd2 = int(pairingMatrix[i,2*j+1] - 1)
        term = (1/C)*segBs[i,j] * phiSyms[pairInd1] * phiSyms[pairInd2]
        row = row  + term
    "prescribe the last columnn separately"
    row = row + (1/C)*segBs[i,phaseNum-2]*phiSyms[i]*(1-phiSumSyms)
    polys.append(row)

global polyDiff
polyDiff = sym.eye(phaseNum-1)
for i in range(phaseNum-1):
    for j in range(phaseNum-1):
        polyDiff[i,j] = sym.diff(polys[i],phiSyms[j])

"Initial volume fraction condition"
# phis[0] = np.concatenate([0.2*np.ones([int(nZ*0.5),nX]),0.8*np.ones([int(nZ*0.5)
```

```
    ,nX])])
phis[0] = np.ones([nZ,nX]) * (1/2)
# phis[1] = np.ones([nZ,nX]) * (1/3)
# phis[2] = np.ones([nZ,nX]) * (1/4)
# phis[3] = np.ones([nZ,nX]) * (1/5)
# phis[4] = np.ones([nZ,nX]) * (1/6)
# phis[5] = np.ones([nZ,nX]) * 0.1
# phis[6] = np.ones([nZ,nX]) * 0.1
# phis[7] = np.ones([nZ,nX]) * 0.1
# phis[8] = np.ones([nZ,nX]) * 0.1


"Generate velocity evolution data to iterate in time"
zCell = np.linspace(zBot,zTop-dz,nZ)+0.5*dz
# uListCell = getList(alphaList, betaList, zCell)
# duListCell = getIncrements(alphaList, betaList, zCell, uListCell)

uListCellHist = np.zeros([nZ,nT+np.size(nBreak)])
duListCellHist = np.zeros([nZ,nT+np.size(nBreak)])

# uListGrdpt = getList(alphaList, betaList, z)
# duListGrdpt = getIncrements(alphaList, betaList, z, uListGrdpt)

"Initialize matrix to store volume fraction and other data"
results1 = [[] for i in range(phaseNum-1+np.size(tBreak))]
for k in range(phaseNum-1):
    results1[k] = np.zeros([nZ,nT+1])
    # results1[k] = np.zeros([nZ,nT+1+np.size(nBreak)])

"Record IC in results"
for i in range(nZ):
    for k in range(phaseNum-1):
        results1[k][i,0] = 0.5*(phis[k][i,int(nX/2)] + phis[k][i,int(nX/2)+1])

"Index used to document evolution progress of velocity profile in time"
# indexI = 0
# indexR = len(alphaList) + len(betaList)

"Loop over time"
t = 0
iSeg = 0
iBreak = 0
while iSeg in range(nT):
    # "check whether the timestep is the start of the initialization period"
    # if indexI+1 <= len(alphaList)+len(betaList):
    #     for i in range(nZ):
    #         uTempCell[i] = uTempCell[i] + direction * duListCell[i,indexI]
    #     for i in range(nZ+1):
    #         uTempGrdpt[i] = uTempGrdpt[i] + direction * duListGrdpt[i,indexI]
    #         "update the counter"
    #     indexI = indexI + 1

    # "check whether the timestep is the start of the reversal period"
    # if iSeg in n0Rev:
    #     "reset the reversal progress counter and record change of direction"
    #     indexR = 0
    #     direction = -1*direction
    #     for i in range(nZ):
    #         uTempCell[i] = uTempCell[i] + 2 * direction * duListCell[i,indexR]
```

```
#       for i in range(nZ+1):
#           uTempGrdpt[i] = uTempGrdpt[i] + 2 * direction * duListGrdpt[i,indexR
      ]

#       "update the counter"
#       indexR = indexR + 1

# "check whether the timestep is within the reversal period"
# if iSeg not in n0Rev and indexR+1 <= len(alphaList)+len(betaList):
#       "add the prescribed increment assigned to this step"
#       for i in range(nZ):
#           uTempCell[i] = uTempCell[i] + 2 * direction * duListCell[i,indexR]
#       for i in range(nZ+1):
#           uTempGrdpt[i] = uTempGrdpt[i] + 2 * direction * duListGrdpt[i,indexR
      ]

#       "update the counter"
#       indexR = indexR + 1

"Second version of velocity profile evolution"
RTemp = 5*np.cos(2*np.pi*t/T)+5.5
velTopTemp = np.sin(np.pi*t/T)
direction = np.sign(velTopTemp)
for i in range(nZ):
    zTemp = zBot + (i+0.5)*dz
    uTempCell[i] = u(zTemp,RTemp,velTopTemp)
for i in range(nZ+1):
    zTemp = zBot + i*dz
    uTempGrdpt[i] = u(zTemp,RTemp,velTopTemp)

"record velocity evolution data"
for i in range(nZ):
    uListCellHist[i,iSeg] = uTempCell[i]

"record shear rate data"
for i in range(1,nZ-1):
    dudzTemp = minmod((uTempCell[i] - uTempCell[i-1])/dz, (uTempCell[i+1] -
        uTempCell[i])/dz)
    duListCellHist[i,iSeg] = dudzTemp

"determine whether it is a breakage timestep or convection-diffusion timestep"
if iSeg in nBreak:
    "step1 of breakage: re-distribution of phi"
    for i in range(nZ):
        for j in range(nX):
            for k1 in range(phaseNum-1):
                phis[k1][i,j] = phis[k1][i,j]- breakPlan[k1]*phis[k1][i,j]

    "store the central column for evolution graph"
    if np.mod(nX,2) == 0:
        for k in range(phaseNum-1):
            for i in range(nZ):
                results1[k][i,t+1] = 0.5*(phis[k][i,int(nX*0.5)] + phis[k][i,
                    int(nX*0.5)+1])

    elif np.mod(nX,2) == 1:
        for i in range(nZ):
            for k in range(phaseNum-1):
                results1[k][i,t+1] = phis[k][i,(nX+1)*0.5]
```

```
        print("breakage takes place")
        "counter"
        t = t + 1

        "step2 of breakage: use the re-distributed phi for evolution"
        "calculate phi for next timestep using KT solver"
        phiNew = rk_timestep(phis)
        phis = phiNew


        "store the central column for evolution graph"
        if np.mod(nX,2) == 0:
            for k in range(phaseNum-1):
                for i in range(nZ):
                    results1[k][i,t+1] = 0.5*(phis[k][i,int(nX*0.5)] + phis[k][i,
                        int(nX*0.5)+1])

        elif np.mod(nX,2) == 1:
            for i in range(nZ):
                for k in range(phaseNum-1):
                    results1[k][i,t+1] = phis[k][i,(nX+1)*0.5]
        "counter"
        print("timestep number ",iSeg ,"completed")
        iSeg = iSeg + 1
        t = t + 1

    else:
        "calculate phi for next timestep using KT solver"
        phiNew = rk_timestep(phis)
        phis = phiNew
        "store the central column for evolution graph"
        if np.mod(nX,2) == 0:
            for k in range(phaseNum-1):
                for i in range(nZ):
                    results1[k][i,t+1] = 0.5*(phis[k][i,int(nX*0.5)] + phis[k][i,
                        int(nX*0.5)+1])

        elif np.mod(nX,2) == 1:
            for i in range(nZ):
                for k in range(phaseNum-1):
                    results1[k][i,t+1] = phis[k][i,(nX+1)*0.5]

        # np.savetxt("results0_10_P0_0.5_spread_real2a1.csv", results1[0],
            delimiter=",")
        # np.savetxt("results1_10_P0_0.5_spread_real2a1.csv", results1[1],
            delimiter=",")
        # np.savetxt("results2_10_P0_0.5_spread_real6b1.csv", results1[2],
            delimiter=",")
        # np.savetxt("results3_10_P0_0.5_spread_real6e.csv", results1[3],
            delimiter=",")
        # np.savetxt("results4_10_P0_0.5_spread_real7.csv", results1[4], delimiter
            =",")
        # np.savetxt("results5_10_P0_0.5_spread_real2.csv", results1[5], delimiter
            =",")
        # np.savetxt("results6_10_P0_0.5_spread_real2.csv", results1[6], delimiter
            =",")
        # np.savetxt("results7_10_P0_0.5_spread_real2.csv", results1[7], delimiter
            =",")
        # np.savetxt("results8_10_P0_0.5_spread_real2.csv", results1[8], delimiter
```

```
            =",")

        "counter"
        print("timestep number ",iSeg+1 ,"completed")
        iSeg = iSeg + 1
        t = t + 1



    # "Calculate and store the shear rate data"
    # dudzCellTemp = np.zeros([nZ,1])
    # for i in range(1,nZ-1):
    #     L = (uTempCell[i] - uTempCell[i-1])/dz
    #     R = (uTempCell[i+1] - uTempCell[i])/dz
    #     dudzCellTemp[i] = np.abs(minmod(L,R))


#%%
# fig, axes =  plt.subplots(1,1,figsize = (5,5))
# plt.imshow(results1[0], interpolation='none', cmap='viridis',aspect=nT/nZ)
# plt.clim(0,1)
# plt.colorbar()
# fig.savefig(r"C:\Users\Jiaxin Zhang\Desktop\myimage", format='png', dpi=1200)
# plt.title("Multiple Datasets in One Plot")
# plt.draw()

#%%
# fig, axes =  plt.subplots(1,1,figsize = (5,5))
# plt.imshow(results1[0], interpolation='none', cmap='viridis',aspect=nT/nZ)
# plt.clim(0,1)
# plt.colorbar()

# fig, axes =  plt.subplots(1,1,figsize = (5,5))
# plt.imshow(results1[1], interpolation='none', cmap='viridis',aspect=nT/nZ)
# plt.clim(0,1)
# plt.colorbar()

# fig, axes =  plt.subplots(1,1,figsize = (5,5))
# plt.imshow(results1[2], interpolation='none', cmap='viridis',aspect=nT/nZ)
# plt.clim(0,1)
# plt.colorbar()

# fig, axes =  plt.subplots(1,1,figsize = (5,5))
# plt.imshow(np.ones([nZ,results1[0].shape[1]])-results1[0]-results1[1]-results1
    [2], interpolation='none', cmap='viridis',aspect=nT/nZ)
# plt.clim(0,1)
# plt.colorbar()

# fig, axes =  plt.subplots(1,1,figsize = (5,5))
# plt.imshow(results1[3], interpolation='none', cmap='viridis',aspect=nT/nZ)
# plt.clim(0,1)
# plt.colorbar()

# fig, axes =  plt.subplots(1,1,figsize = (5,5))
# plt.imshow(results1[4], interpolation='none', cmap='viridis',aspect=nT/nZ)
# plt.clim(0,1)
# plt.colorbar()
```

```
# fig, axes =  plt.subplots(1,1,figsize = (5,5))
# plt.imshow(np.ones([nZ,nT+1])-results1[0]-results1[1]-results1[2]-results1[3]-
    results1[4], interpolation='none', cmap='viridis',aspect=nT/nZ)
# plt.clim(0,1)
# plt.colorbar()


# np.savetxt("results0_10_P0_0_spread_leftside.csv", results1[0], delimiter=",")
# np.savetxt("results1_10_P0_0_spread_leftside.csv", results1[1], delimiter=",")
# np.savetxt("results2_10_P0_0_spread_leftside.csv", results1[2], delimiter=",")
# np.savetxt("results3_10_P0_0_spread_leftside.csv", results1[3], delimiter=",")
# np.savetxt("results4_10_P0_0_spread_leftside.csv", results1[4], delimiter=",")
# np.savetxt("results5_10_P0_0_spread_leftside.csv", results1[5], delimiter=",")
# np.savetxt("results6_10_P0_0_spread_leftside.csv", results1[6], delimiter=",")
# np.savetxt("results7_10_P0_0_spread_leftside.csv", results1[7], delimiter=",")
# np.savetxt("results8_10_P0_0_spread_leftside.csv", results1[8], delimiter=",")

# end = tm.time()

# resultsNew1 = results1[0]
# resultsNew2 = results1[1]
# resultsOld1 = resultsL
# resultsOld2 = resultsS

# fig, axes =  plt.subplots(1,1,figsize = (5,5))
# plt.imshow(resultsNew1-resultsOld1, interpolation='none', cmap='viridis',aspect=
    nT/nZ)
# plt.colorbar()

# fig, axes =  plt.subplots(1,1,figsize = (5,5))
# plt.imshow(resultsNew2-resultsOld2, interpolation='none', cmap='viridis',aspect=
    nT/nZ)
# plt.colorbar()
```

Appendix D

Poly-disperse simulation figures

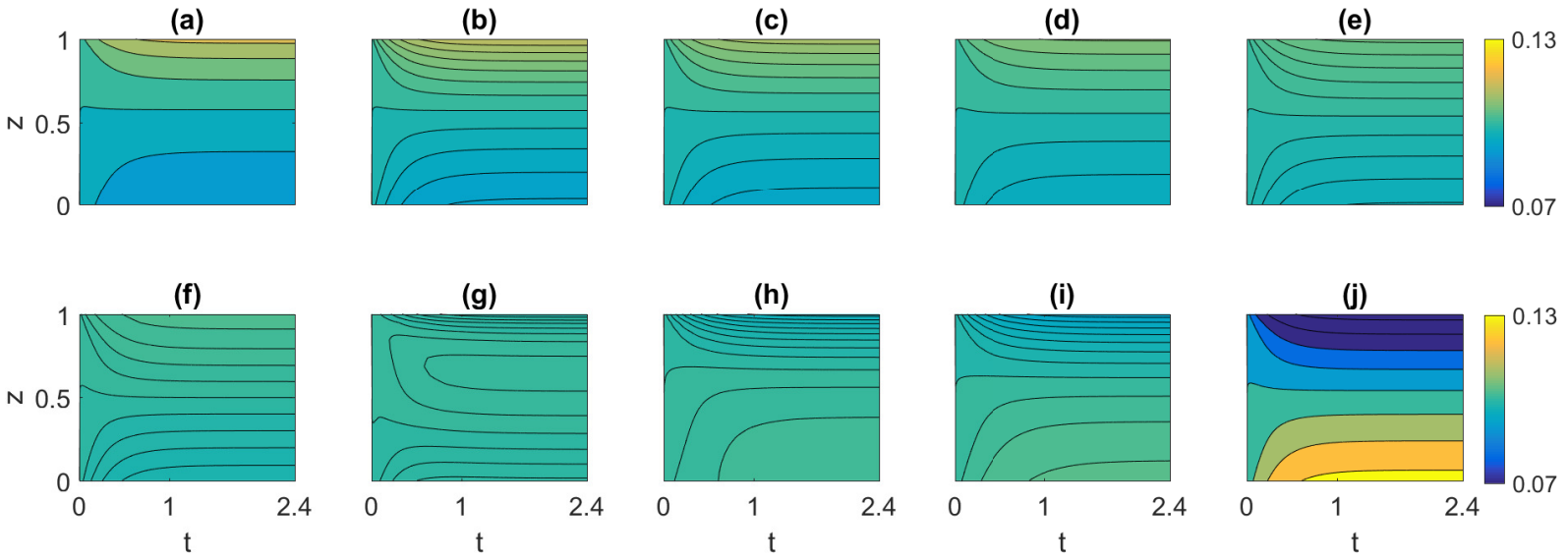


Figure D.1: Simulation results for small grain rich distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

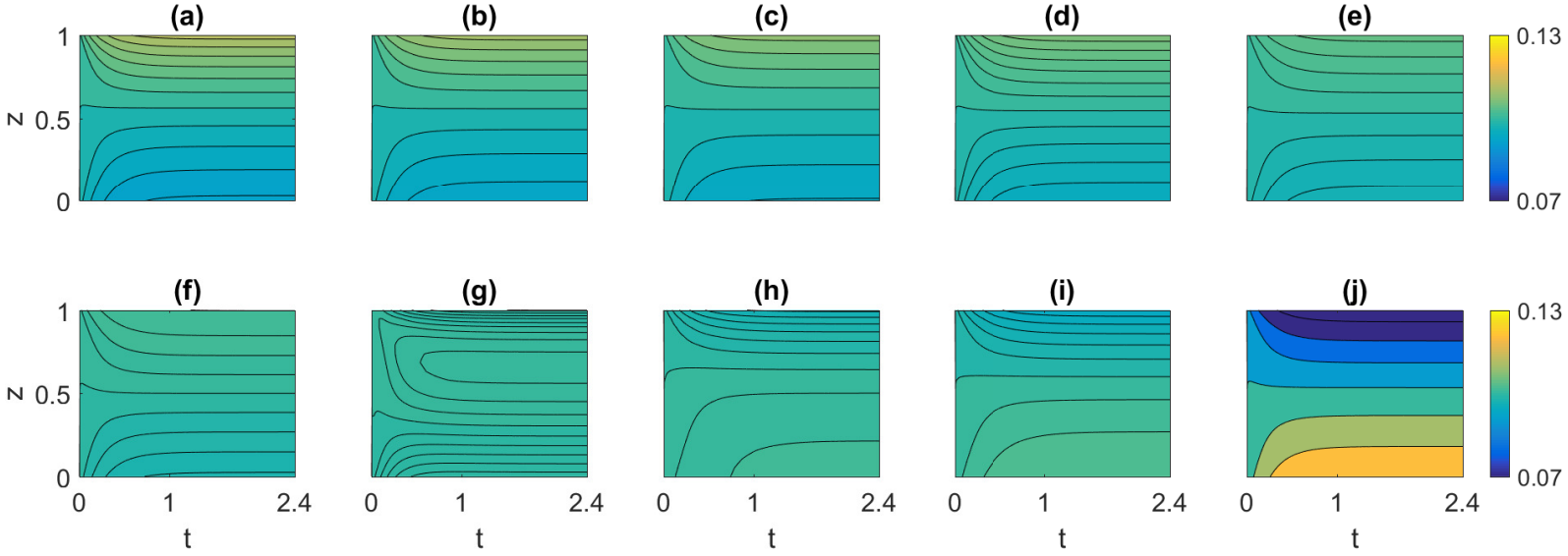


Figure D.2: Simulation results for small grain rich distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.2$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

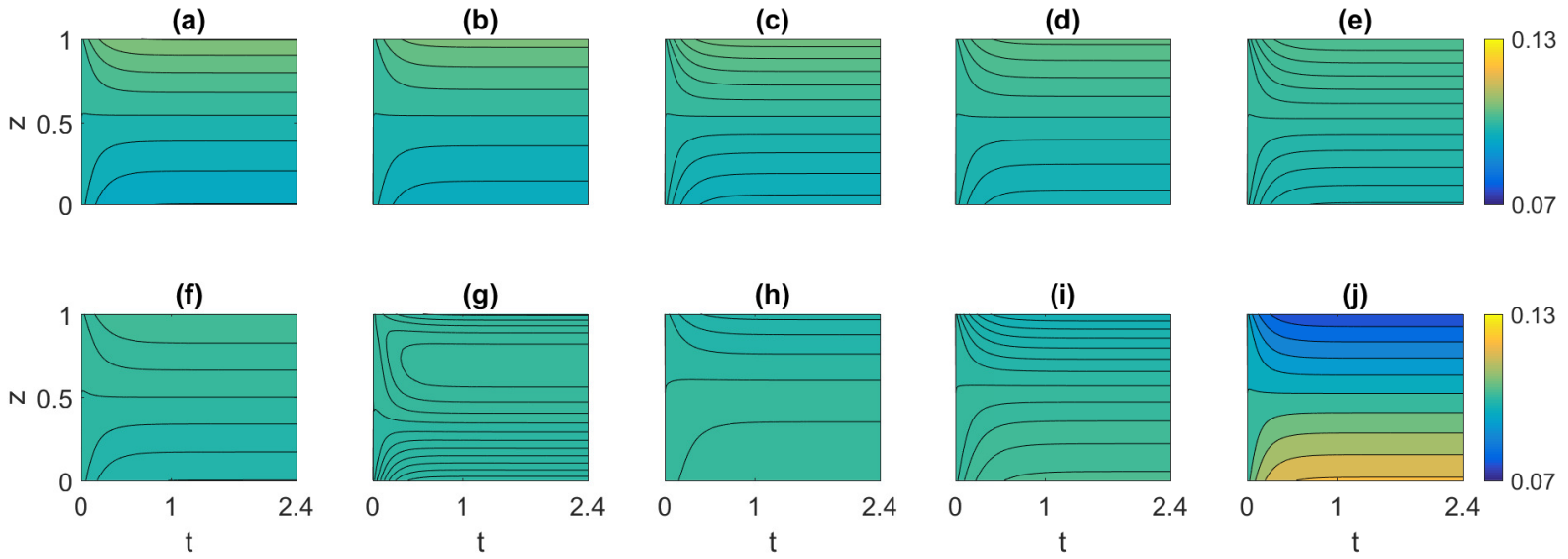


Figure D.3: Simulation results for small grain rich distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.5$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

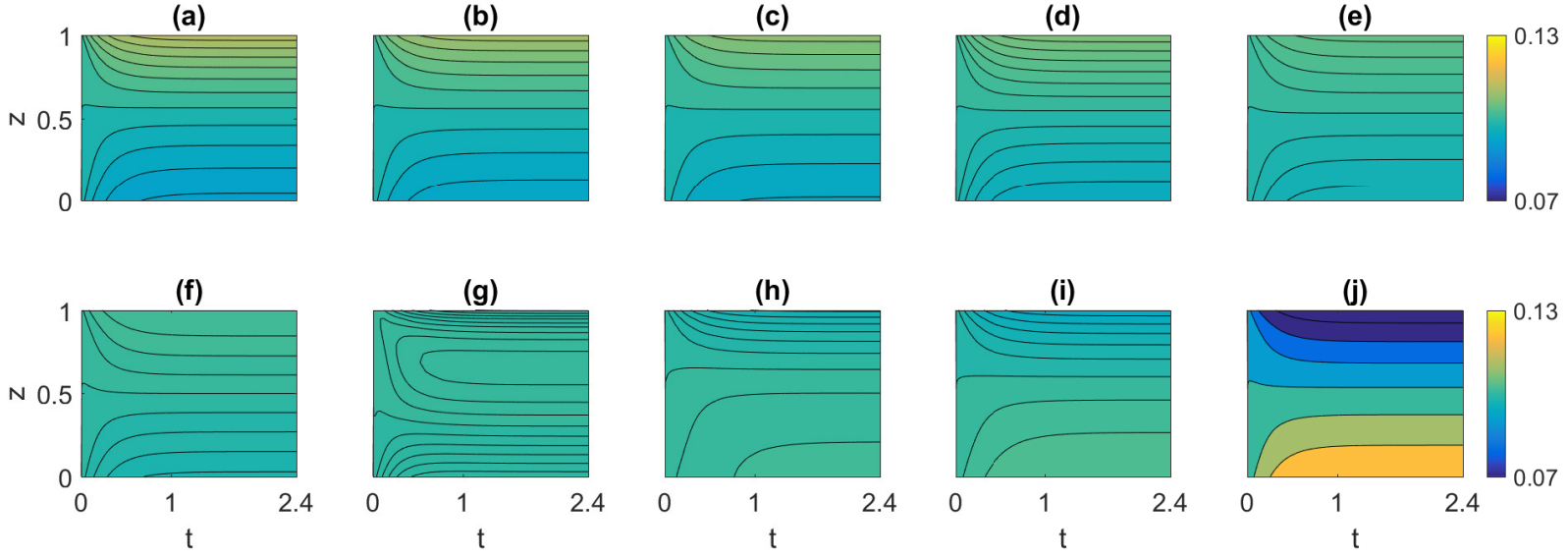


Figure D.4: Simulation results for small grain rich distribution with diffusion rate $\mathcal{D} = 0.2$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

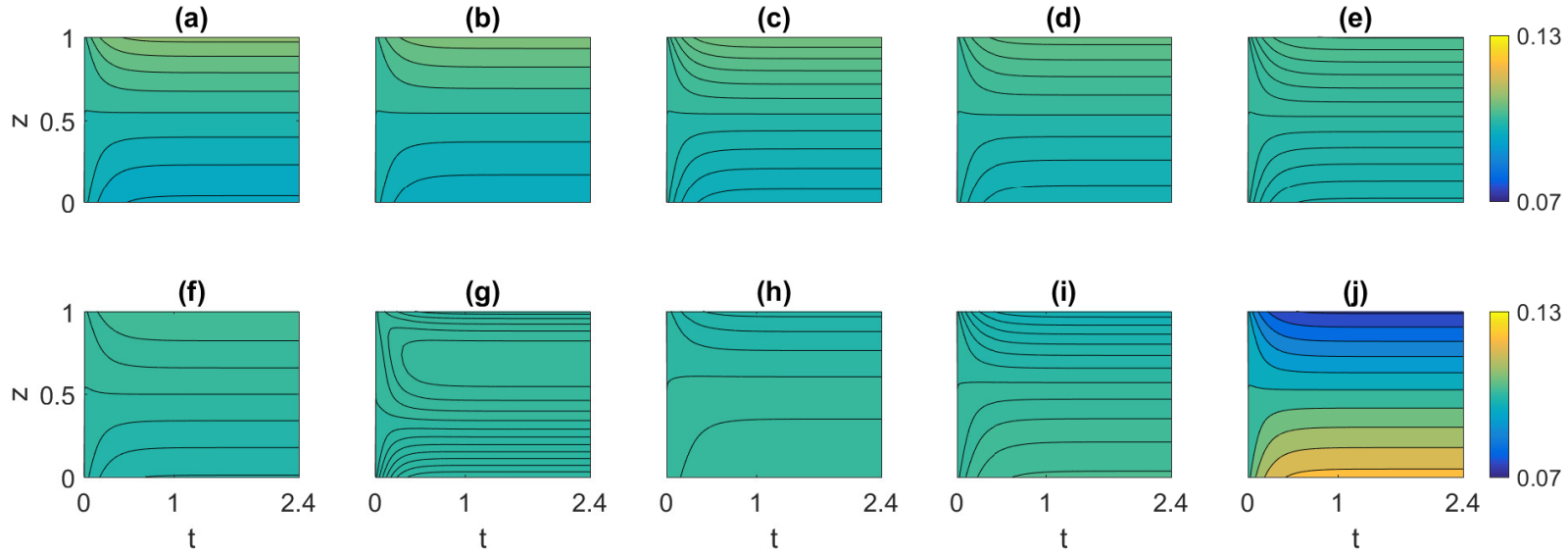


Figure D.5: Simulation results for small grain rich distribution with diffusion rate $\mathcal{D} = 0.5$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

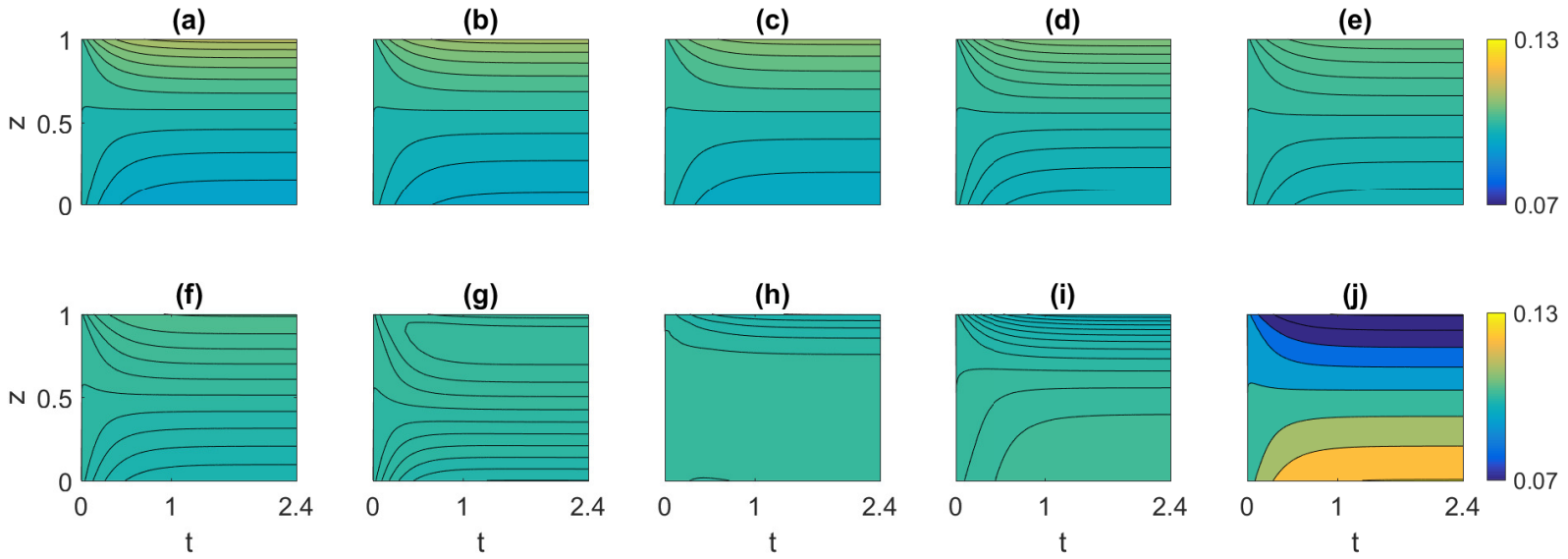


Figure D.6: Simulation results for large grain rich distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

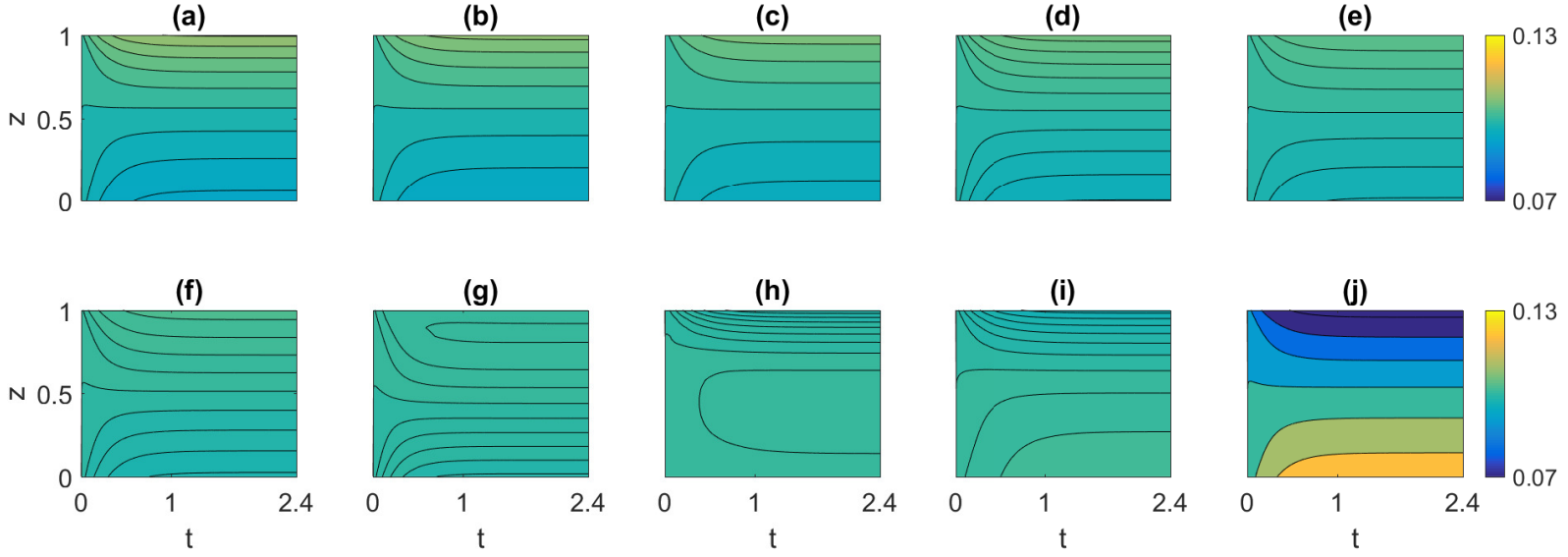


Figure D.7: Simulation results for large grain rich distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.2$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

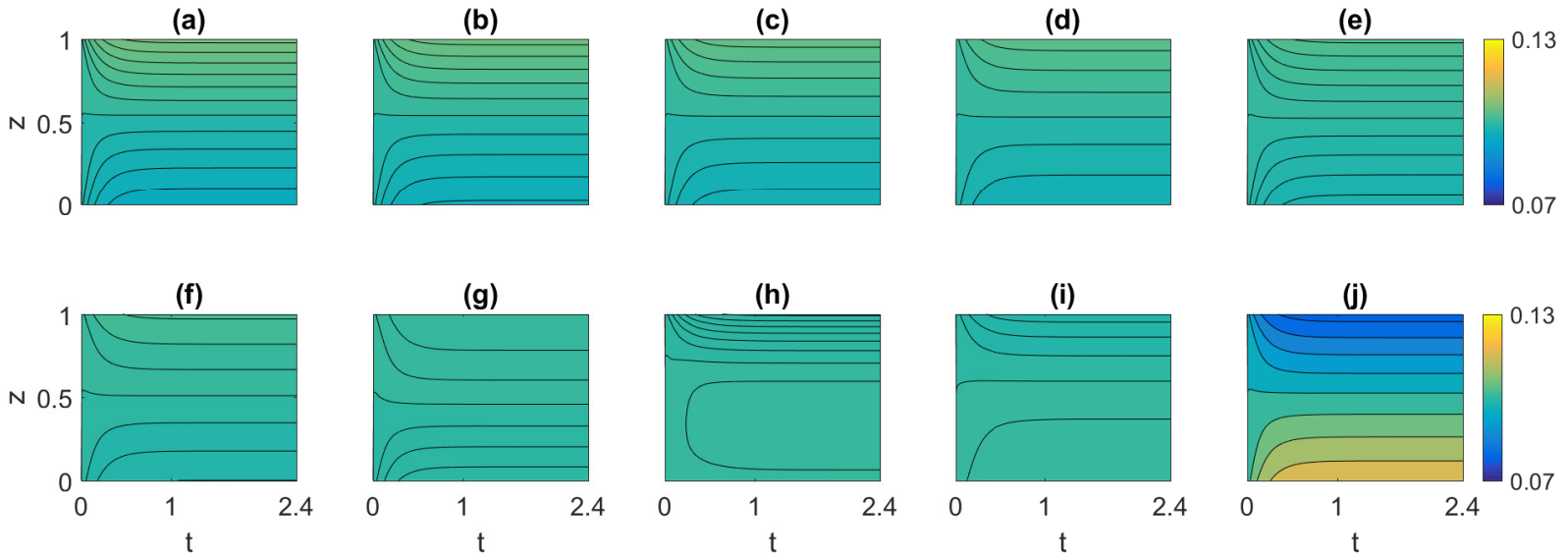


Figure D.8: Simulation results for large grain rich distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.5$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

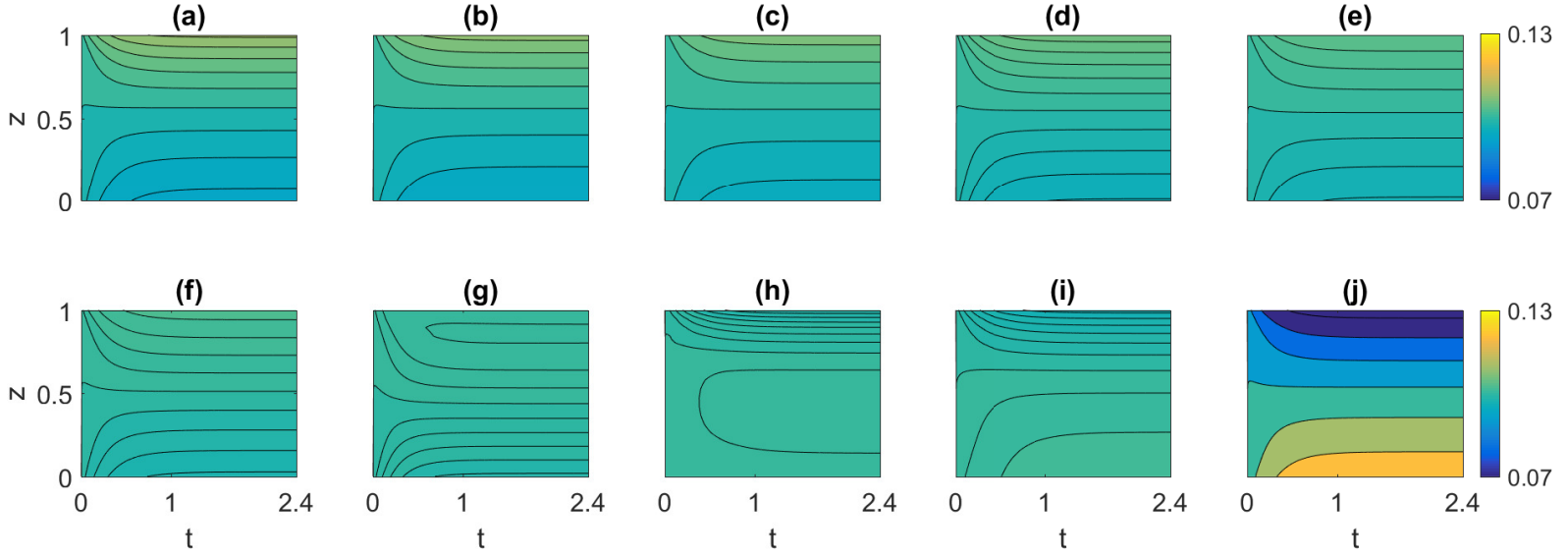


Figure D.9: Simulation results for large grain rich distribution with diffusion rate $\mathcal{D} = 0.2$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

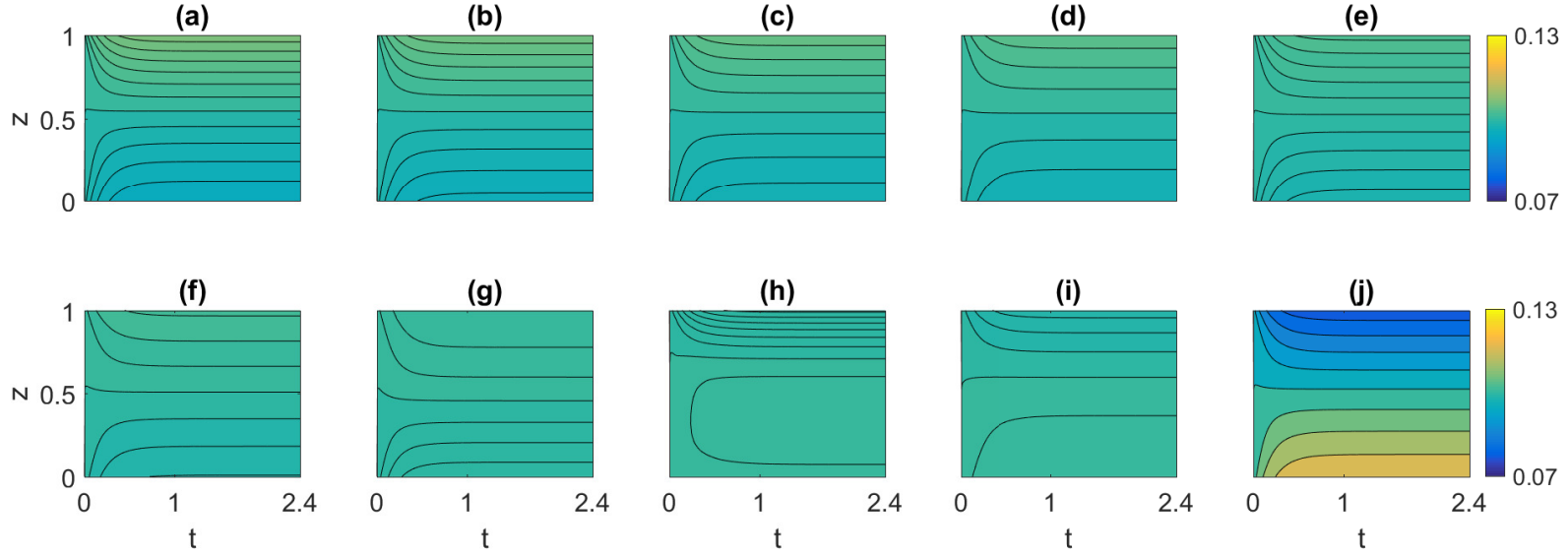


Figure D.10: Simulation results for large grain rich distribution with diffusion rate $\mathcal{D} = 0.5$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

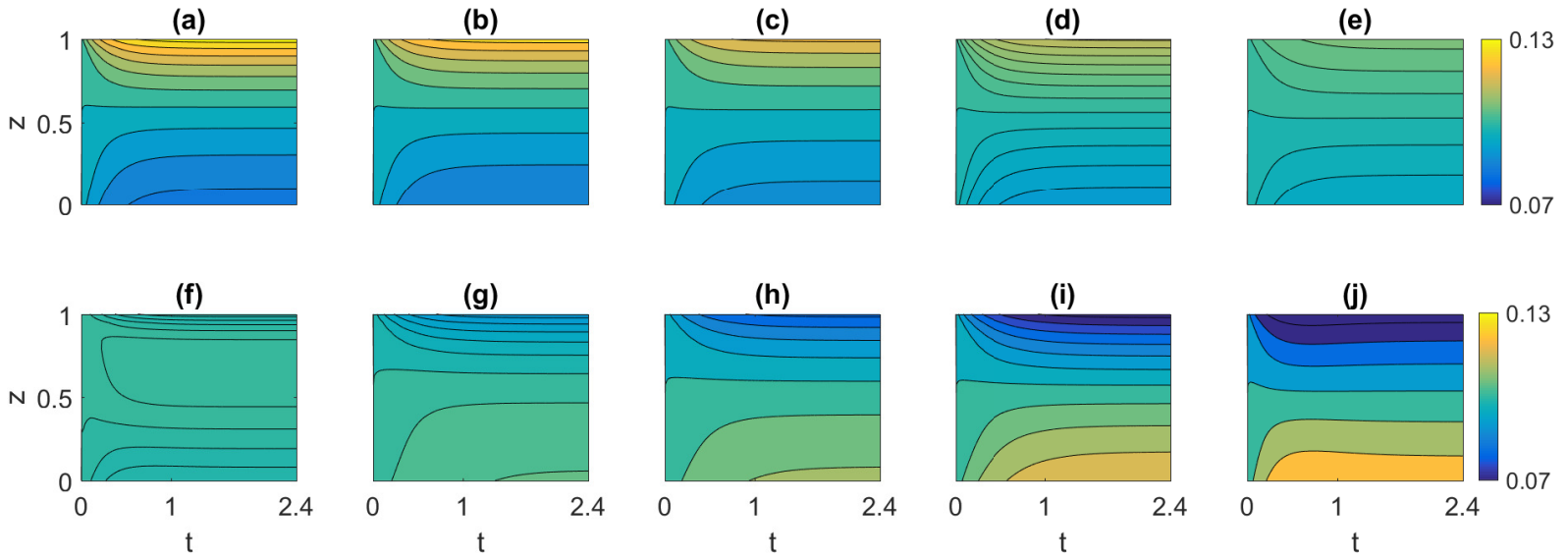


Figure D.11: Simulation results for peak spectrum distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

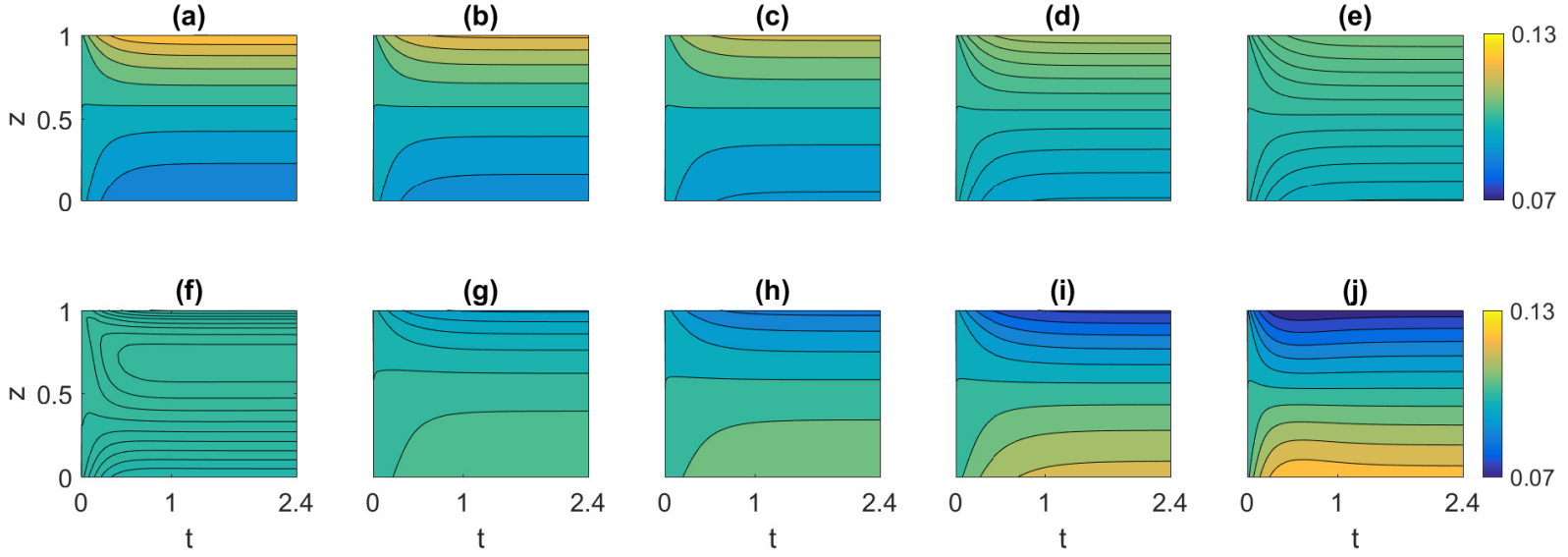


Figure D.12: Simulation results for peak spectrum distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.2$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

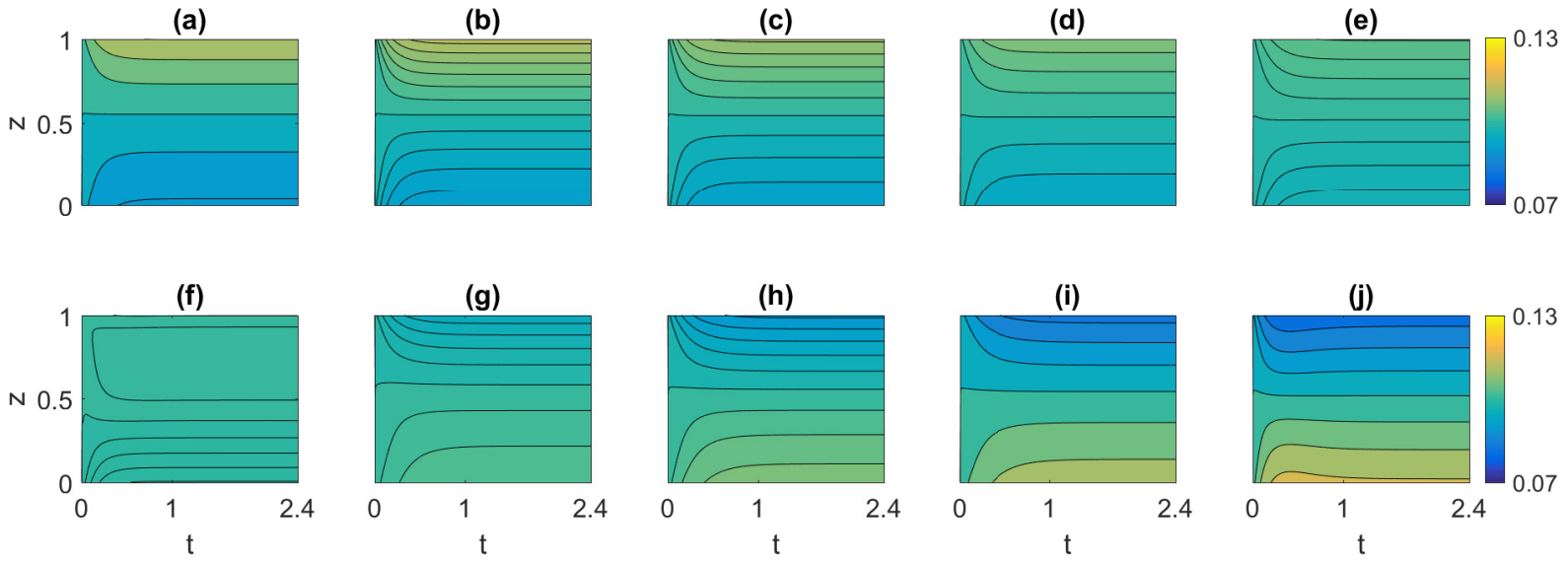


Figure D.13: Simulation results for peak spectrum distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.5$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

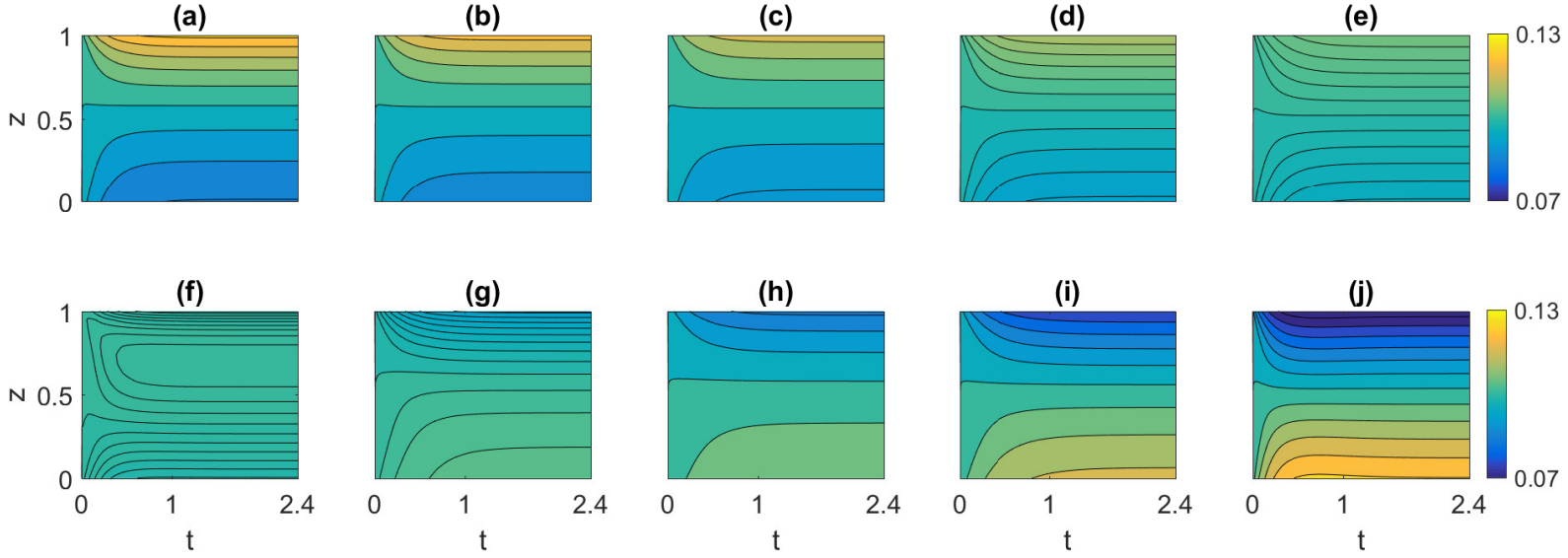


Figure D.14: Simulation results for peak spectrum distribution with diffusion rate $\mathcal{D} = 0.2$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

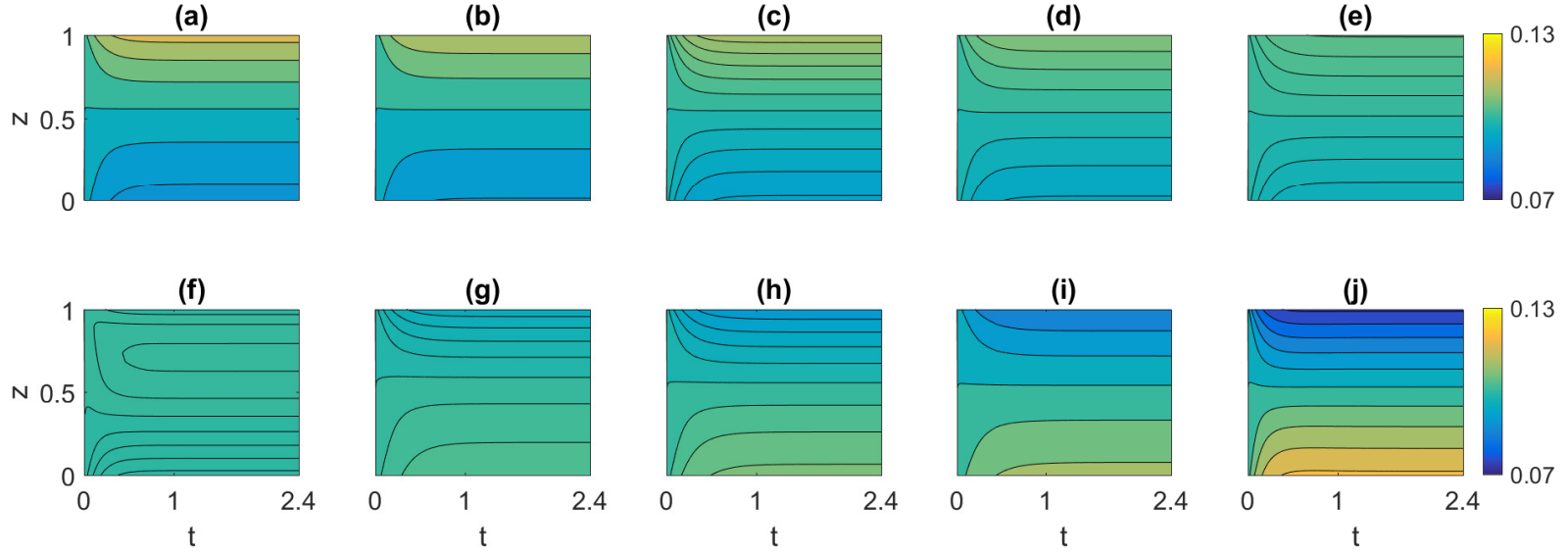


Figure D.15: Simulation results for peak spectrum distribution with diffusion rate $\mathcal{D} = 0.5$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

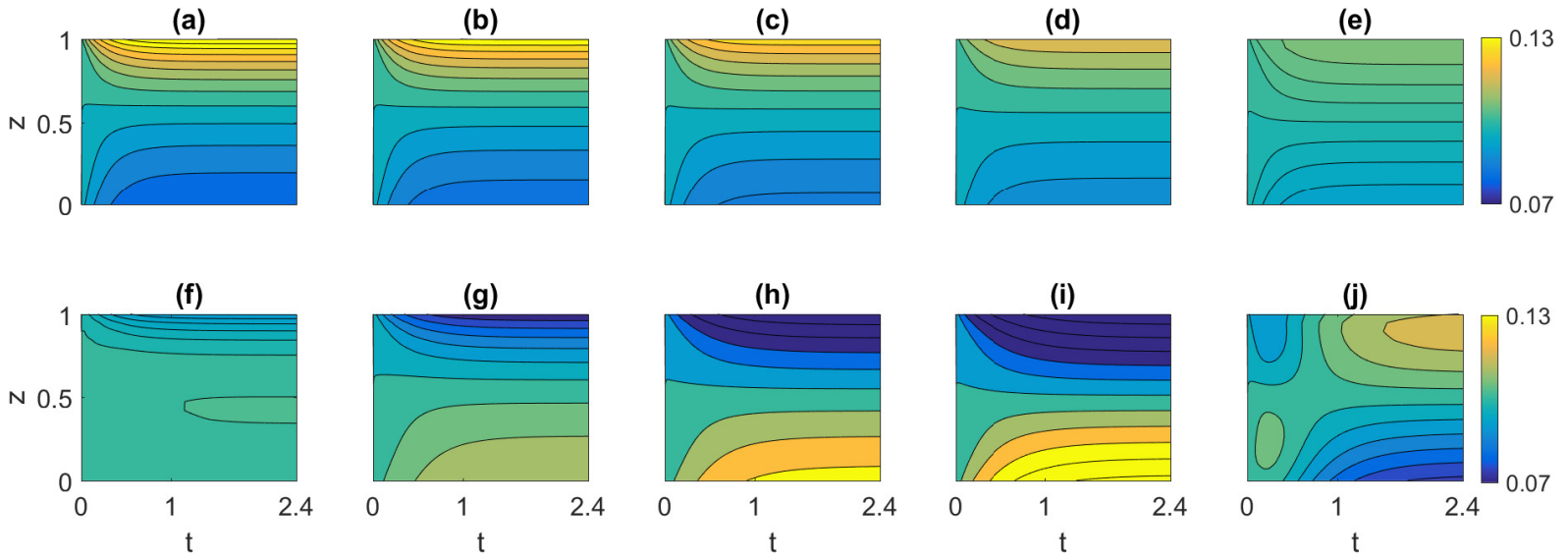


Figure D.16: Simulation results for broad spectrum distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

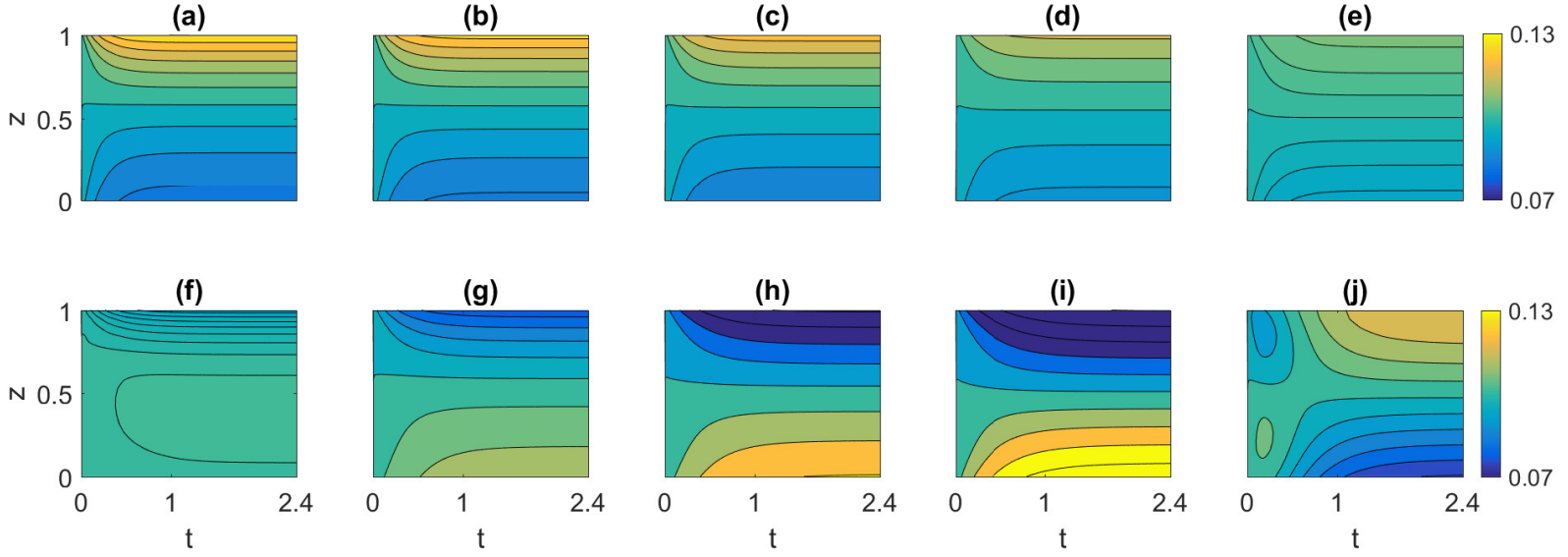


Figure D.17: Simulation results for broad spectrum distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.2$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

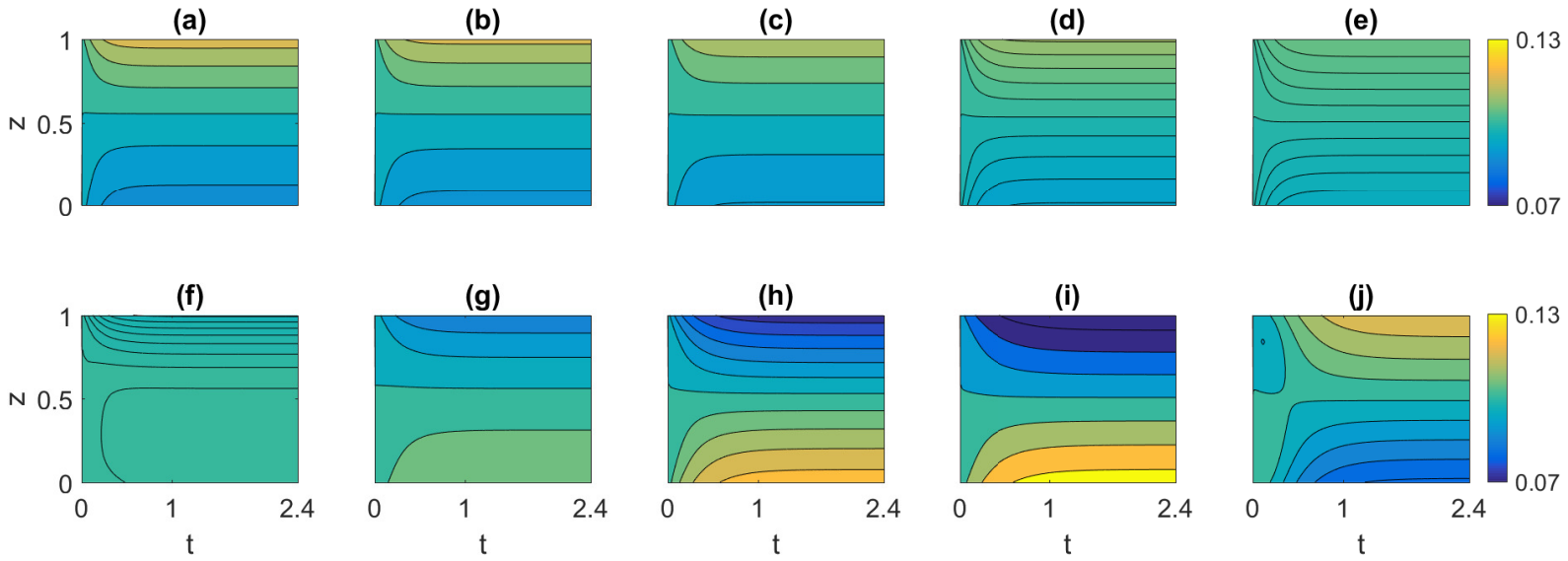


Figure D.18: Simulation results for broad spectrum distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.5$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

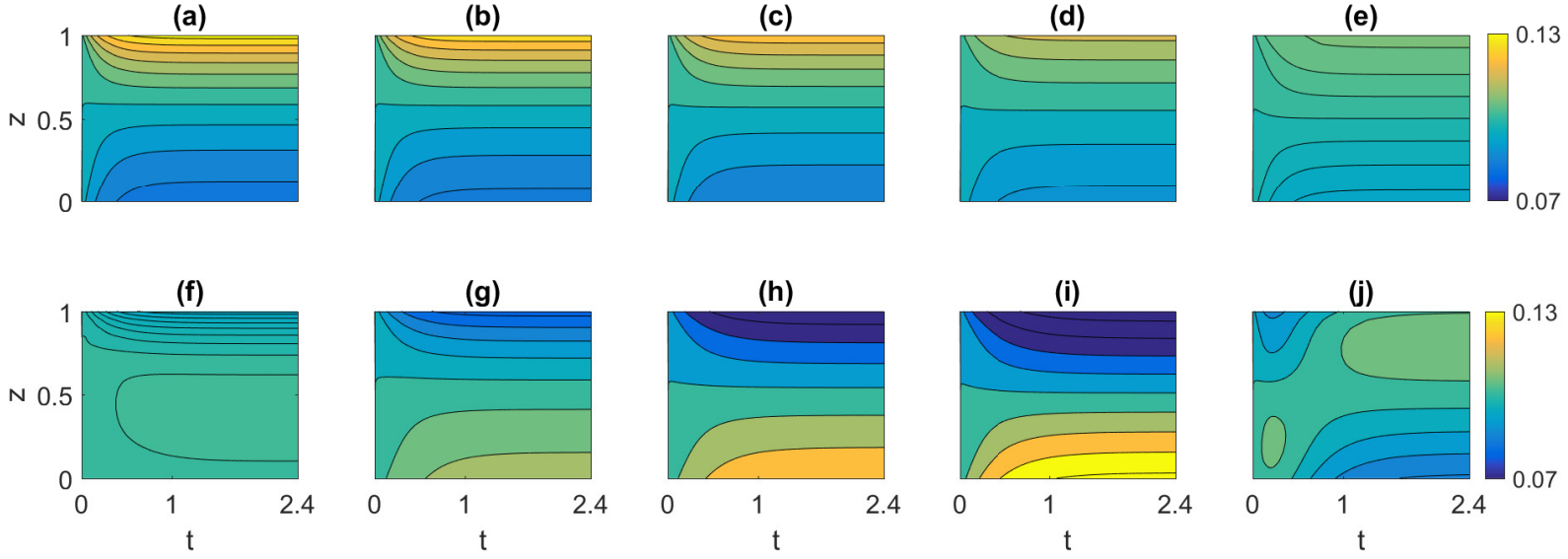


Figure D.19: Simulation results for broad spectrum distribution with diffusion rate $\mathcal{D} = 0.2$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

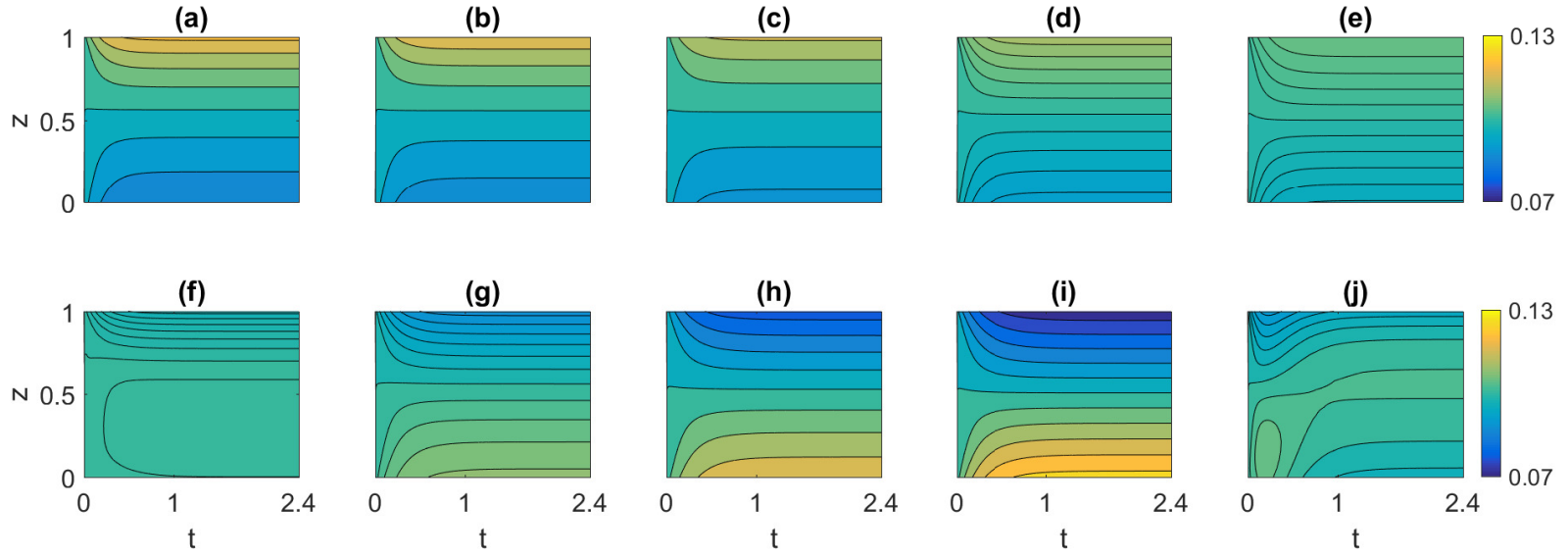


Figure D.20: Simulation results for broad spectrum distribution with diffusion rate $\mathcal{D} = 0.5$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

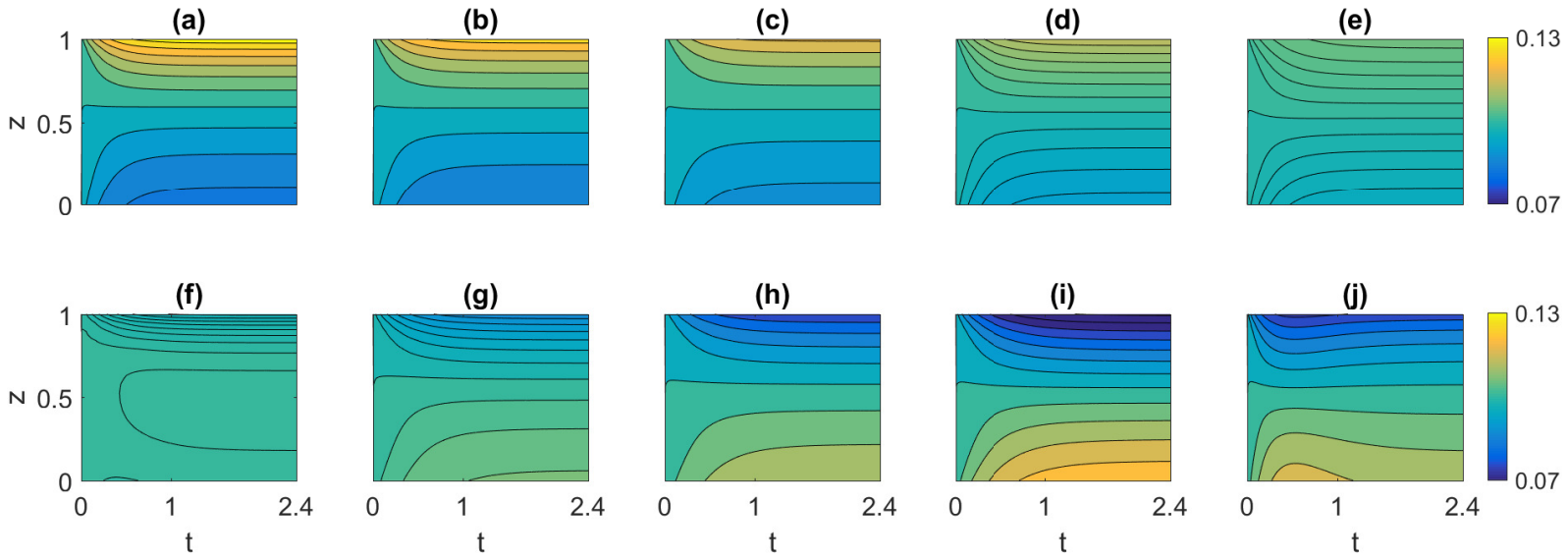


Figure D.21: Simulation results for 'realistic' distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

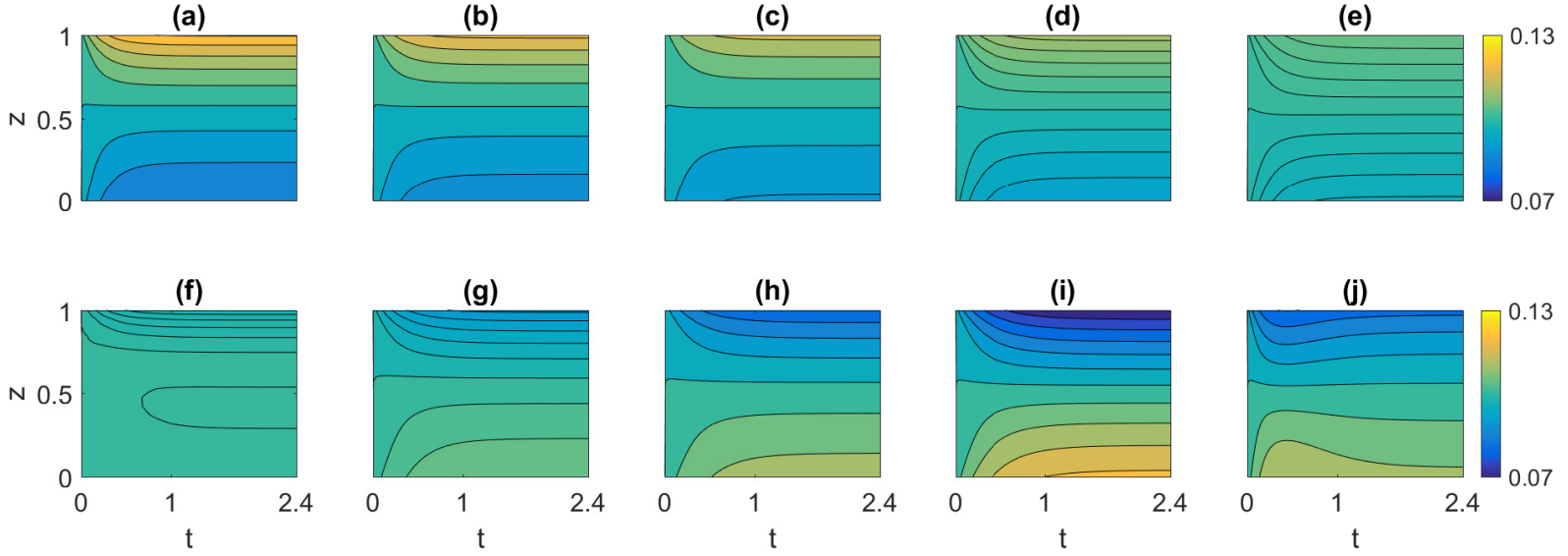


Figure D.22: Simulation results for 'realistic' distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.2$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

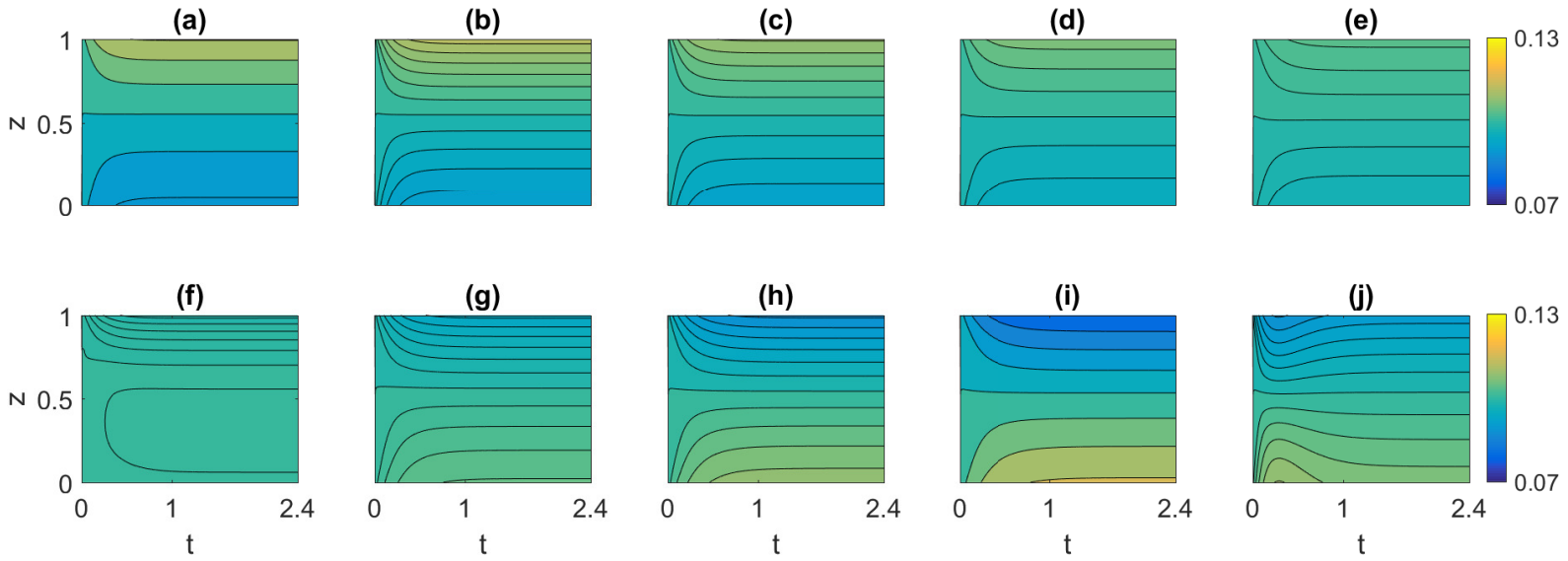


Figure D.23: Simulation results for 'realistic' distribution with diffusion rate $\mathcal{D} = 0.1$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.5$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

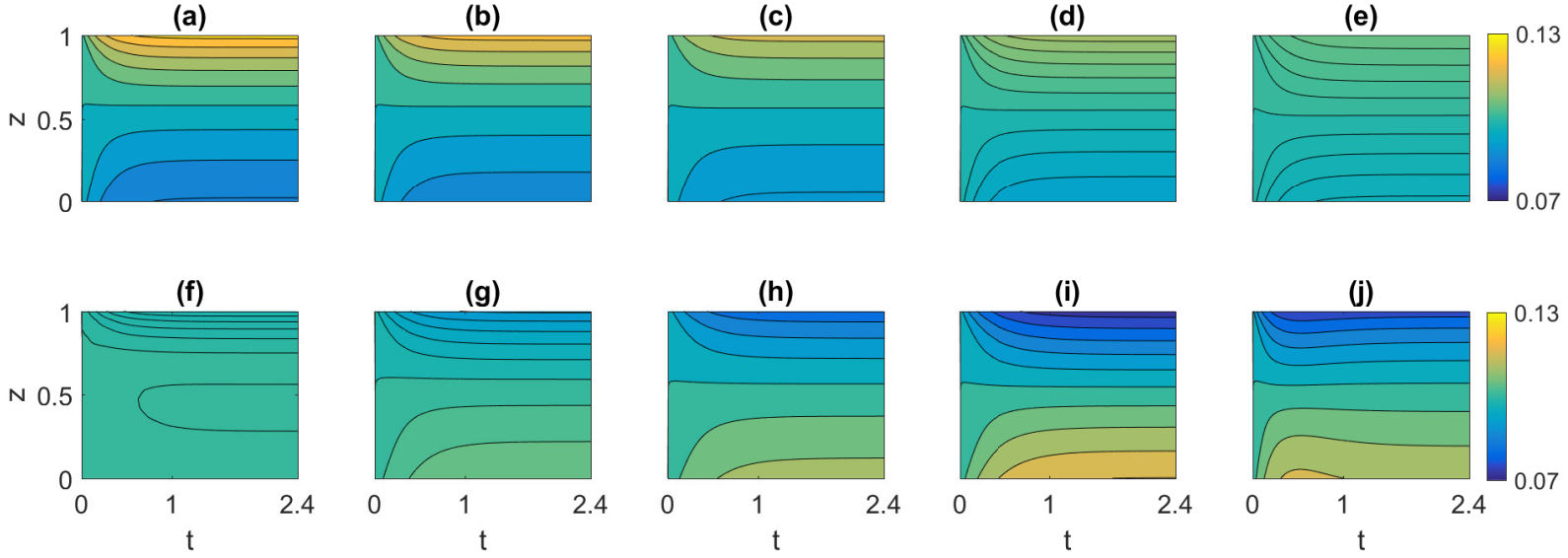


Figure D.24: Simulation results for 'realistic' distribution with diffusion rate $\mathcal{D} = 0.2$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

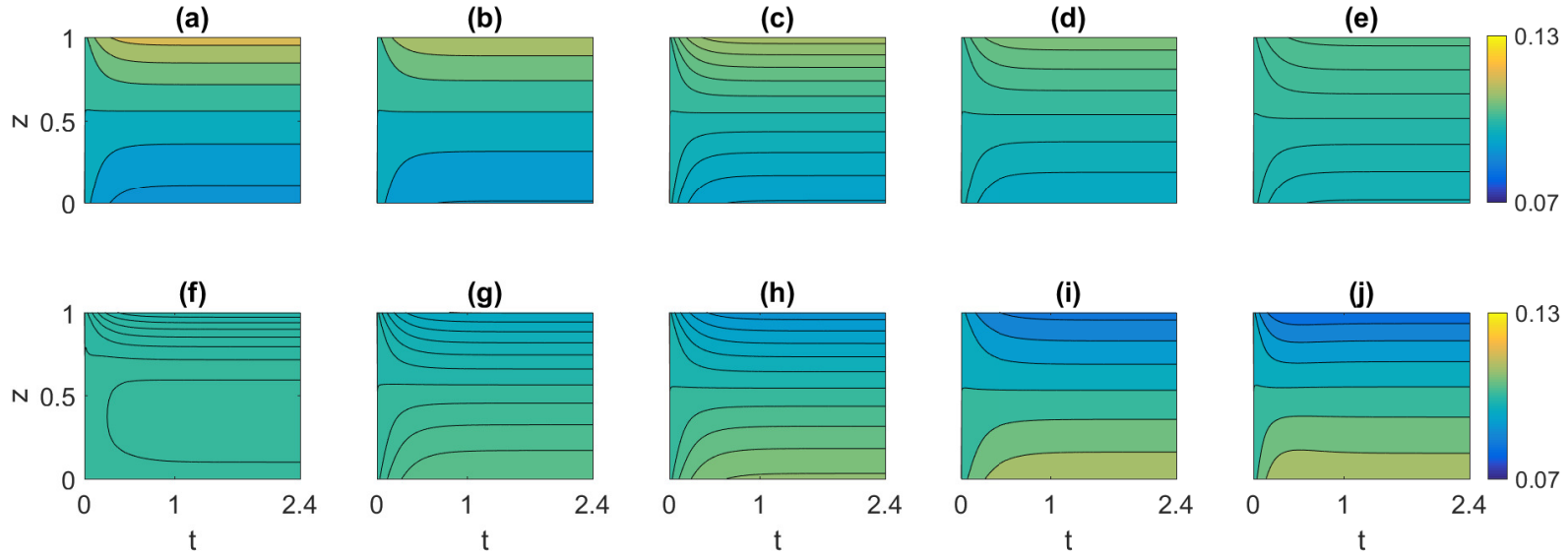


Figure D.25: Simulation results for 'realistic' distribution with diffusion rate $\mathcal{D} = 0.5$ and non-dimensionalized confining pressure $\mathcal{P}_0 = 0.1$. Sub-figures (a)-(j) correspond to the grain populations in descending order of size.

Appendix E

Matlab code for triangular breakage simulation

```
    clear
%% set control parameters
pBase = 0.8;
iterationNum = 100;
particleSideList = [3,3,3,3,3,3,3,3,3,3]';
sizeRatio = 1/3;

%% iterate automatically

for k = 1:iterationNum
    % make a temporary copy of data
    sideList_temp = particleSideList;

    % from the 3rd iteration of breakage, apply the D10 non-break filter
    if k > 2
        particleCounts = 1;
        for i = 1:size(sideList_temp,2)
            % count polygon number of each size category
            particleCounts(i,1) = length(nonzeros(sideList_temp(:,i)));
        end

        particleSizes = zeros(sum(particleCounts),1);
        sumNum = 0;
        for i = 1:length(particleCounts)
            % access particles of one size category
            sumNumNew = sumNum + particleCounts(i);
            % record the width of this category, setting the starting value
            % to be 1
            particleSizes(sumNum+1:1:sumNumNew) = 1*((1/3)^(i-1));
            sumNum = sumNumNew;
        end

        % get the D10 threshold
```

```
        threshold = quantile(particleSizes, 0.10);

        % apply the D10 filter:
        % Dilema: the sizes of polygons obey a discrete distribution, hence
        % the D10 threshold would only filter  polygons in big 'chunks'.
        % But for the time being, I am abusing this fact, just to see what
        % would happen. So I am looping over all the size columns, check
        % if the size of this column is smaller than D10 threshold. If so,
        % we make the algorithm think this class is 6-sided for this
        % iteration ONLY.
        for j = 1:size(sideList_temp,2)
            sizeTemp = 1*((1/3)^(j-1));
            if sizeTemp < threshold
                % if some column has size smaller than threshold,
                % we treat this column as hexagons for this iteration ONLY
                sideList_temp(sideList_temp ~= 0) = 6;
            end
        end
    end


    % check the number of 'corners' available to break
    sideList_temp = 6 - sideList_temp;
    sideList_temp = mod(sideList_temp,6);

    %possibleBreaksTotal = sum(sum(sideList_temp));

    % access each present particle
    for i = 1:size(sideList_temp,1)
        for j = 1:size(sideList_temp,2)
            nCorners = sideList_temp(i,j);

            % if it has 6 sides or is empty entry, skip
            if nCorners == 0
                continue
            end

            p1 = p(pBase,j);
            % simulate breakage for ONE particle with given probability
            breakOutcome = randsample([1,0], nCorners, true, [p1, 1 - p1]);
            % record how many breakages actually occur
            breakCount = length(nonzeros(breakOutcome));
            % update particle information: mother particle gets more sides
            particleSideList(i,j) = particleSideList(i,j) + breakCount;
            % check if there are some 'newer' class particles already
            if size(particleSideList,2) == j
                newParticleCount = 1;
            else
                newParticleCount = length(nonzeros(particleSideList(:,j+1))) + 1;
            end
            % update particle information: new particles are triangles
            for count = 1:breakCount
                particleSideList(newParticleCount,j+1) = 3;
                newParticleCount = newParticleCount+1;
            end
        end
    end
    % particleSideList
end
```

```
% calculate number of particles of each size (from largest to smallest)
particleCounts = 1;
for i = 1:size(particleSideList,2)
    particleCounts(i,1) = length(nonzeros(particleSideList(:,i)));
end

% population percentage of the two smallest sizes
sum(particleCounts([end-1,end])) / sum(particleCounts)

% assume the original particle has diameter 1, and each 'fragment' has
% 1/3 width of its mother particle
particleSizes = zeros(sum(particleCounts),1);
sumNum = 0;
for i = 1:length(particleCounts)
    sumNumNew = sumNum + particleCounts(i);
    particleSizes(sumNum+1:1:sumNumNew) = 1*((sizeRatio)^(i-1));
    sumNum = sumNumNew;
end

figure
p3 = cdfplot(particleSizes);
set(p3,'linewidth',2)
set(gca, 'XScale', 'log')
xlabel('Size')
ylabel('Cumulative percentage')
set(gca,'FontSize',15)
title('')
% print('-depsc','tribreak_cdf.eps')
```