

# PARTICLE SIZE-SEGREGATION IN CONVEX ROTATING DRUMS

A THESIS SUBMITTED TO THE UNIVERSITY OF MANCHESTER  
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY  
IN THE FACULTY OF SCIENCE AND ENGINEERING

2007

**Daniel Mouny**  
School of Mathematics

ProQuest Number: 13894564

All rights reserved

INFORMATION TO ALL USERS

The quality of this reproduction is dependent upon the quality of the copy submitted.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if material had to be removed, a note will indicate the deletion.



ProQuest 13894564

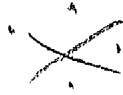
Published by ProQuest LLC (2019). Copyright of the Dissertation is held by the Author.

All rights reserved.

This work is protected against unauthorized copying under Title 17, United States Code  
Microform Edition © ProQuest LLC.

ProQuest LLC.  
789 East Eisenhower Parkway  
P.O. Box 1346  
Ann Arbor, MI 48106 – 1346

(ET6NO)



Th 30468



# Contents

<b>Abstract</b>	<b>8</b>
<b>Declaration</b>	<b>9</b>
<b>Copyright Statement</b>	<b>10</b>
<b>Acknowledgements</b>	<b>12</b>
<b>1 The rotating drum</b>	<b>16</b>
1.1 An introduction to the rotating drum . . . . .	16
1.2 Granular caustics . . . . .	17
1.3 The billiard mapping . . . . .	18
<b>2 Segregation in granular avalanches</b>	<b>27</b>
2.1 An introduction to segregation . . . . .	27
2.2 The avalanche . . . . .	28
<b>3 Segregation in rotating drums</b>	<b>35</b>
3.1 An introduction to segregation in rotating drums . . . . .	35
3.2 Theory . . . . .	36
3.2.1 The mapping approach . . . . .	36



3.3	Methods . . . . .	39
3.3.1	Locating the surface . . . . .	39
3.3.2	Particles on a string . . . . .	43
3.3.3	A finite volume approach . . . . .	52
3.3.4	Experimental method . . . . .	68
<b>4</b>	<b>Comparison of results in a triangular drum</b>	<b>75</b>
4.1	Initial comparison to experiment . . . . .	75
4.2	Varying ratios . . . . .	80
4.3	Varying fill levels . . . . .	81
4.4	Long time behaviour . . . . .	86
4.5	Special considerations . . . . .	90
4.5.1	Symmetry of the flow . . . . .	90
4.5.2	The effect of noise . . . . .	92
4.5.3	fifty percent fill . . . . .	95
<b>5</b>	<b>General convex drums</b>	<b>97</b>
5.1	The square drum . . . . .	97
5.1.1	Comparison to experiment . . . . .	97
5.1.2	Varying ratios . . . . .	100
5.1.3	Varying fill levels . . . . .	103
5.1.4	Long time behaviour . . . . .	103
5.1.5	fifty percent fill . . . . .	107
5.2	Other drums . . . . .	108
5.2.1	Curved drums . . . . .	109
5.2.2	Affine mapping . . . . .	111

<b>6 Discussion</b>	<b>117</b>
<b>Bibliography</b>	<b>119</b>
<b>A NOCD source code</b>	<b>123</b>
A.1 main.cc . . . . .	123
<b>B Particle on a string source code</b>	<b>136</b>
B.1 main.cc . . . . .	136
B.2 triangle.cc . . . . .	138
<b>C Volumetric method source code</b>	<b>149</b>
C.1 main.cc . . . . .	149
C.2 cdum.h . . . . .	151
C.3 cdum.cc . . . . .	152
C.4 csurface.h . . . . .	159
C.5 csurface.cc . . . . .	160

Word count 18953

# List of Figures

1.1	A typical rotating drum with the granular caustic highlighted . . .	17
1.2	Choosing the initial point for the billiard mapping iteration . . . . .	20
1.3	An example of a point being billiard mapped around the caustic. . .	21
1.4	Billiard mapping orbits around a circular caustic . . . . .	22
1.5	Billiard mapping orbits around an elliptic caustic . . . . .	23
1.6	An overview of billiard mapping orbits . . . . .	24
1.7	Billiard mapping orbits within the drum . . . . .	25
1.8	Billiard mapping orbits at various fill levels . . . . .	26
2.1	Overview of drum and avalanche parameters . . . . .	29
2.2	Graph of the concentration of the incoming material . . . . .	30
2.3	Numerical simulation of a plug flow in a parabolic avalanche . . . .	31
2.4	Numerical simulation of a shearing flow in a parabolic avalanche . .	33
3.1	A long exposure view of the avalanching region . . . . .	37
3.2	Close-up view of avalanching region. . . . .	38
3.3	Typical polygon vertex labelling. . . . .	40
3.4	Typical surface for a given $\theta$ . . . . .	41
3.5	The surface search parameter . . . . .	42
3.6	Quadrilateral region in which $p$ must lie. . . . .	43

3.7	Image of a typical experiment at an intermediate stage. . . . .	44
3.8	The intersections required for the particle on a string method . . .	45
3.9	Special cases that require treatment in a particle on a string scheme.	53
3.10	Potential problems in the particle on a string approach . . . . .	53
3.11	Typical results using the particle on a string approach . . . . .	54
3.12	An example of a nearest neighbour interpolated rotation . . . . .	56
3.13	An example of a linearly interpolated rotation . . . . .	57
3.14	An diagram of an angular increment of the surface . . . . .	58
3.15	A close-up view of the up-slope avalanche region . . . . .	59
3.16	Examples of intersection cases for a rectangle and a triangle . . . .	61
3.17	An illustration of the method by which an angular increment is chosen.	65
3.18	The method by which a minimal angular increment is chosen. . . .	66
3.19	Locating the segregation interface in the outgoing material . . . . .	67
3.20	Outgoing elements highlighted on the grid . . . . .	67
3.21	The equilateral triangular drum . . . . .	69
3.22	A series of photos of the rotating triangular drum . . . . .	73
4.1	Triangle experiment, $A = 75\%$ , $\Phi = 45\%$ , $\theta = \pi$ . . . . .	76
4.2	Triangle simulation, $A = 75\%$ , $\Phi = 45\%$ , $\theta = \pi$ . . . . .	77
4.3	Triangle experiment, $A = 75\%$ , $\Phi = 45\%$ , $\theta = 3\pi$ . . . . .	78
4.4	Triangle simulation, $A = 75\%$ , $\Phi = 45\%$ , $\theta = 3\pi$ . . . . .	79
4.5	Triangle experiment, $A = 75\%$ , $\Phi = 45\%$ , $\theta = 19\pi$ . . . . .	80
4.6	Triangle simulation, $A = 75\%$ , $\Phi = 45\%$ , $\theta = 19\pi$ . . . . .	81
4.7	Simulations of the triangle at various particle ratios . . . . .	82
4.8	Experiments with the triangle at various particle ratios . . . . .	83
4.9	Simulations of the triangle at various fill levels . . . . .	84

4.10	Experiments with the triangle at various fill levels . . . . .	85
4.11	A graph of the difference parameter $C$ for the triangle . . . . .	88
4.12	A graph of the limiting values of $C$ for the triangle . . . . .	89
4.13	Matching high and low fill levels for the triangular drum . . . . .	90
4.14	Rotated high and low fill levels for the triangular drum . . . . .	91
4.15	Two simulations of the triangle with added noise . . . . .	93
4.16	Simulations of the triangle with added noise at various parameters .	94
4.17	Triangle experiment, $A = 50\%$ , $\Phi = 50\%$ , $\theta = 20\pi$ . . . . .	95
4.18	Triangle simulation, $A = 50\%$ , $\Phi = 50\%$ , $\theta = 20\pi$ . . . . .	96
5.1	A series of photos of the rotating square drum . . . . .	98
5.2	Square experiment, $A = 75\%$ , $\Phi = 50\%$ , $\theta = 20\pi$ . . . . .	99
5.3	Square simulation, $A = 75\%$ , $\Phi = 50\%$ , $\theta = 20\pi$ . . . . .	100
5.4	Simulations of the square at various particle ratios . . . . .	101
5.5	Experiments with the square at various particle ratios . . . . .	102
5.6	Simulations of the square at various fill levels . . . . .	104
5.7	Experiments with the square at various fill levels . . . . .	105
5.8	A graph of the difference parameter $C$ for the square . . . . .	106
5.9	A graph of the limiting values of $C$ for the square . . . . .	107
5.10	Square experiment, $A = 50\%$ , $\Phi = 50\%$ , $\theta = 20\pi$ . . . . .	108
5.11	Square simulation, $A = 50\%$ , $\Phi = 50\%$ , $\theta = 20\pi$ . . . . .	109
5.12	Simulations of stadium and egg shaped drums. . . . .	110
5.13	Simulations of triangles with an affine mapped version . . . . .	113
5.14	Simulations of ellipses with an affine mapped version . . . . .	114
5.15	Simulations of quadrilaterals with affine mapped versions . . . . .	115
5.16	Experiments using parallelograms with an affine mapped version . .	116

# The University of Manchester

Daniel Mounty

Doctor of Philosophy

Particle size-segregation in convex rotating drums

December 31, 2007

Rotating drums are widely used in many processes in the pharmaceutical and bulk chemical industries to mix and segregate grains. Monodisperse flows exhibit strongly chaotic particle motion that leads to rapid mixing. When there are two or more grain sizes however, segregation imposes strong ordering on the system that leads to the formation of striking quasi-periodic patterns. These patterns are strongly dependent on the fill height and the mixing ratio of the particles. Despite the complex nature of the flow and the resulting patterns the essential features are captured by a simple model that accurately predicts the position of the lobes and arms. This novel approach can be applied to general convex drums and arbitrary initial conditions. The approach itself as well as its computational implementation will be described and compared to experiment.

# Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institute of learning.

# Copyright Statement

- i. The author of this thesis (including any appendices and/or schedules to this thesis) owns any copyright in it (the "Copyright") and s/he has given The University of Manchester the right to use such Copyright for any administrative, promotional, educational and/or teaching purposes.
- ii. Copies of this thesis, either in full or in extracts, may be made **only** in accordance with the regulations of the John Rylands University Library of Manchester. Details of these regulations may be obtained from the Librarian. This page must form part of any such copies made.
- iii. The ownership of any patents, designs, trade marks and any and all other intellectual property rights except for the Copyright (the "Intellectual Property Rights") and any reproductions of copyright works, for example graphs and tables ("Reproductions"), which may be described in this thesis, may not be owned by the author and may be owned by third parties. Such Intellectual Property Rights and Reproductions cannot and must not be made available for use without the prior written permission of the owner(s) of the relevant Intellectual Property Rights and/or Reproductions.
- iv. Further information on the conditions under which disclosure, publication



and exploitation of this thesis, the Copyright and any Intellectual Property Rights and/or Reproductions described in it may take place is available from the Head of the School of Mathematics.

# Acknowledgements

I would like to thank my supervisor professor Nico Gray for all his help and advise throughout this project as well as the University of Manchester School of Mathematics who provided me with funding. I would also like to thank the organisers of the conferences at which I have presented, in particular those at the Manchester centre for non-linear dynamics, the University of Bristol and the Southern Granular workshop in Chile.

## Objectives

We wish to develop a theory and computational method that will allow for the prediction of the segregation patterns that result from the slow rotation of poly-disperse granular materials in arbitrary convex drums. The theory should be as simple as possible while still providing predictive power. The computational method should be efficient enough for real-time, or faster, simulations on a typical desktop computer. The resulting method could provide the first steps towards simulating many industrial processes which typically take place at drum fill levels below 40%.

# Granular materials

The static and dynamic behaviour of granular materials are seen in a broad spectrum of naturally occurring and man made systems. The scales of the structures involved can vary hugely from the continental scales of pack ice floes around the poles to the formation of ridges on the shore by the ebbing ocean. The systems consist of particles that are small compared to the scale of the structure and typically too numerous to consider individually. In flows containing a variety of constituents the species may both mix and separate [Bri76].

With the increasing power of computer systems brute force simulation of individual particles become viable. Discrete element methods of the form commonly used in computational chemistry may be applied in such cases [Bar94]. Even with spatial partitioning and parallel approaches combined with the increasing availability of multi-core processors and multi-processor systems, it is still not viable to apply a direct simulation approach to typical full scale system [SC99].

One approach to simulating such flows is to consider the material to be a continuum, paralleling the typical Navier-Stokes type formulation used for fluids. The archetypal fluid like model is presented by Savage and Hutter [SH89]. Schemes of this sort provide good prediction in certain regimes but the formulation of general constitutive laws to encapsulate the energetics and state changes that may be observed, have proven elusive.

If one focusses more on the energetics one may apply an approach that mirrors the behaviour of a kinetic gas [JS83]. Such approaches appear to provide good predictive power in certain cases but their theoretic foundation does not allow for the prediction of dense flows which are often observed.

# Chapter 1

## The rotating drum

### 1.1 An introduction to the rotating drum

A common method used to mix granular materials is a rotating prismatic drum. The drum's major axis is often aligned horizontally or slightly downward to allow for gradual axial transport. In a mixture where the particles have sufficiently similar material properties they will all follow the bulk mass flow [HKG<sup>+</sup>99, Gra01, MSMJ95].

Such systems are often used in the pharmaceutical and bulk chemical industries [SM00, OK00]. In such cases it is extremely important that the mixing be thorough. The motion of the individual particles is chaotic [EV99]. In many cases this chaotic motion leads to thorough mixing of the materials. This is not, however, always the case.

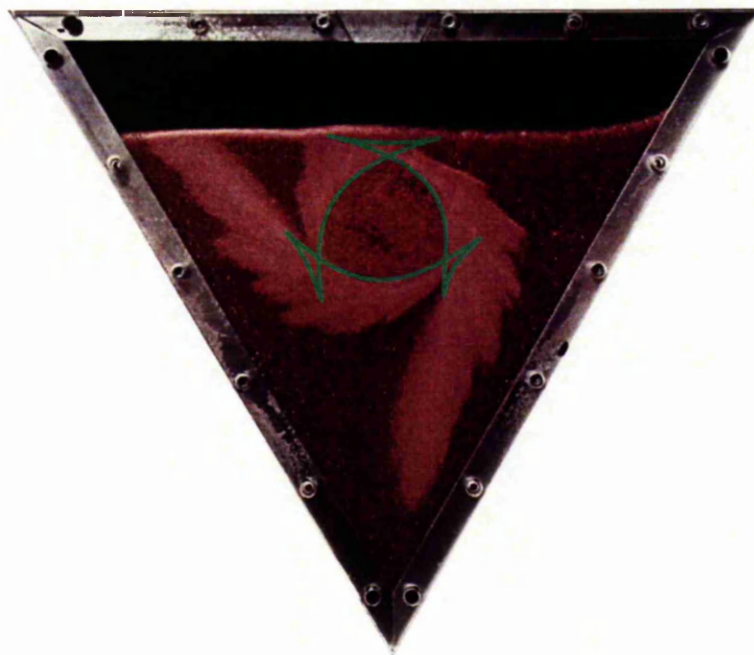


Figure 1.1: An illustration of a granular caustic in a typical rotating drum experiment. The full caustic matching the visible core of revolution is overlaid in green.

## 1.2 Granular caustics

Without even considering the material flow one useful piece of information can be obtained. In general there is a region of the drum that never passes through the surface. For fill levels below fifty percent this corresponds to a region outside the material within the drum and hence isn't clearly visible. When the fill level is above fifty percent this region corresponds to an obviously visible region of the material; figure 1.1 illustrates a typical experiment with the granular caustic overlaid. The outline of this core is a sub region of the caustic for the drum and is referred to as a 'Core of revolution' [GM07].

### 1.3 The billiard mapping

A parallel can be drawn between the path of a particle within the drum and an outer billiard mapping [Tab95] on the caustic curve that defines the core of revolution. To analyse these particle paths in greater depth a computational simulation of an outer billiard mapping representing the path of an individual particle through numerous revolutions of the drum is produced. In this case a single mapping step constitutes a particle having passed through the avalanching region and then deposited.

In section 3.3.1 a more generic method whereby the caustic for a given drum at a specified fill level can be generated, will be provided. For now an exact formulation of the caustic for a specific figure taken from [GM07] is used. The entire derivation and resulting formula are rather verbose and are omitted for brevity as the details are not necessary for the analysis undertaken.

In order to carry out the required mapping the caustic curve  $\mathbf{r}_c$  is parametrised appropriately

$$\mathbf{r}_c(\theta) = \begin{pmatrix} X_c(\theta) \\ Z_c(\theta) \end{pmatrix}. \quad (1.1)$$

A function,  $F(\theta)$  that represents the dot product of the normal to  $\mathbf{r}_c(\theta)$  and the point to iterate  $\mathbf{p}_n$  must be produced

$$F(\theta) = \begin{pmatrix} \frac{\partial Z_c}{\partial \theta} \\ -\frac{\partial X_c}{\partial \theta} \end{pmatrix} \cdot (\mathbf{p}_n - \mathbf{r}_c(\theta)). \quad (1.2)$$

The function has zeroes when the tangent to  $\mathbf{r}_c$  passes through  $\mathbf{p}_n$ . Given a general curve  $\Gamma$  there may be an arbitrary number of zeroes in this function. By conservation of mass we know that for any given angle,  $\theta$  there is a unique solution for the surface position, since the surface is a line and the curve is continuous there



are, almost everywhere, two possible solutions, one corresponding to the in-flow and one to the out-flow. The only exception to this is points that lie within the “ears” of the caustic.

The forms of the caustic in general may be very complex and equation 1.2, may not permit exact solution for zeroes. In fact the exact functional form of the caustic may not be known and a numerical approximation used. For this reason the zeroes must be found using a numerical method. There are several approaches to numerical solution of general functions with various advantages and disadvantages. One of the simplest is an interval bisection approach. There is a sign change of  $F$  around each root so a given interval may be subdivided until a sign change region has been located to some appropriate degree of accuracy. To solve for a function with a single zero this is typically a trivial procedure, however this function has two. This means any chosen interval may potentially span both roots adding further complications. There is no refined control as to which root the method will obtain; in a sense the search is blind. This problem can however be completely avoided if one of the solutions is known in advance. Assuming one root is known to lie at  $F(\theta_0)$  and that the function repeats every  $2\pi$  the search may be limited to  $\theta \in (\theta_0 + \delta\theta, \theta_0 + 2\pi - \delta\theta)$  for some appropriately small  $\delta\theta$ . Since there are two zeroes one must lie within this interval and an interval bisection method is guaranteed to succeed. In order to obtain the first zero there are two options. First of all an interval bisection method may be used to find a root of the equation then locate the other and pick the root that provides the appropriate sense of rotation. Alternatively an arbitrary point on the caustic may be selected and any point on the family of curves that lie on the tangent to that point may be chosen. For simplicity and in order to investigate the families of curves, the latter

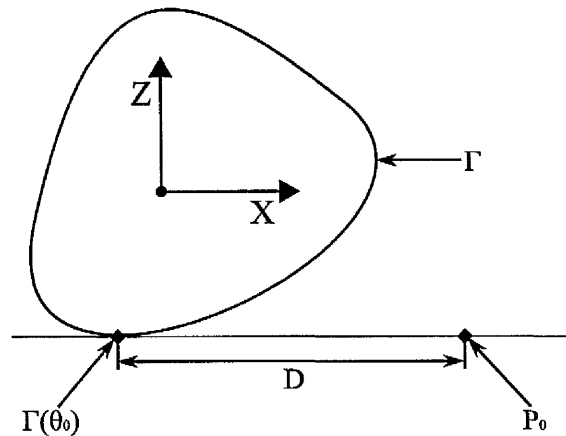


Figure 1.2: The method used to choose an initial point  $\mathbf{p}_0$  for use in the billiard mapping iteration.

approach is chosen.

A base point  $\mathbf{r}_c(\theta_0)$  is selected and the initial point  $\mathbf{p}_0$  is considered to lie at a distance  $D$  along the tangent in the positive direction, as shown in figure 1.2. Given this initial point it may be uniquely iterated to the next in the sequence.

$$\mathbf{p}_{n+1} = 2\mathbf{r}_c(\theta_n) - \mathbf{p}_n. \quad (1.3)$$

In this way a given point may be iterated arbitrarily around any caustic. Figure 1.3 illustrates several iteration steps for a typical point. One point of note is that if one chooses a negative value of  $D$  the result corresponds to rotation in the opposite direction.

To confirm the correct function of the billiard map iterator two simple cases are examined. In each case nine orbits containing fifty thousand points are plotted. Figure 1.4 contains an iteration for the circle. As expected concentric circular orbits at the distances dictated by the choice of initial conditions are obtained. Figure 1.5 contains a comparable iteration for an ellipse. Once again concentric elliptical orbits are seen at the distances dictated by the initial conditions. Given

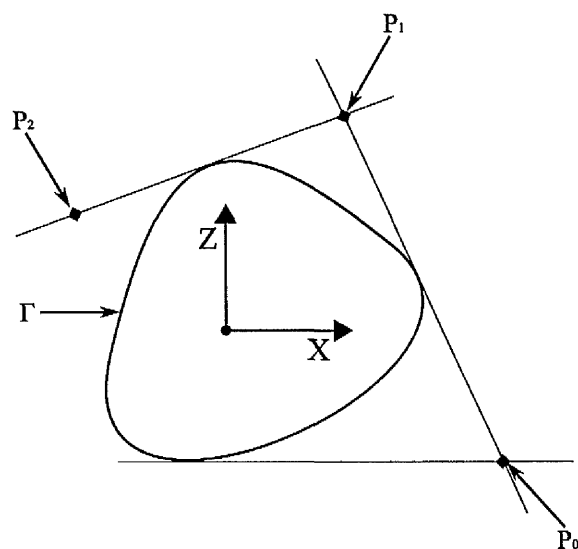


Figure 1.3: An example of a point being billiard mapped around the caustic.

the agreement of these known simple cases with expected results, the method is performing as expected.

In a typical outer billiard mapping system it is mathematically relevant to consider the iteration of any point outside the curve. Traditional billiard mappings are also applied to convex caustics so this approach is in fact a generalisation of the classical approach. In general billiard mapping one may make mathematical statements about, for example, the far field behaviour of the orbits. Compared to typical billiard mapping this method has a definite domain from which to choose points to iterate: those that lie within the drum that generated the caustic. Far field behaviour and other such analysis normally associated with billiard mappings are hence not physically relevant. Despite this, figure 1.6 illustrates the orbits of various points overlaid onto a single diagram, including some of those theoretical orbits outside the drum.

Clearly there are some interesting mathematical features outside and inside

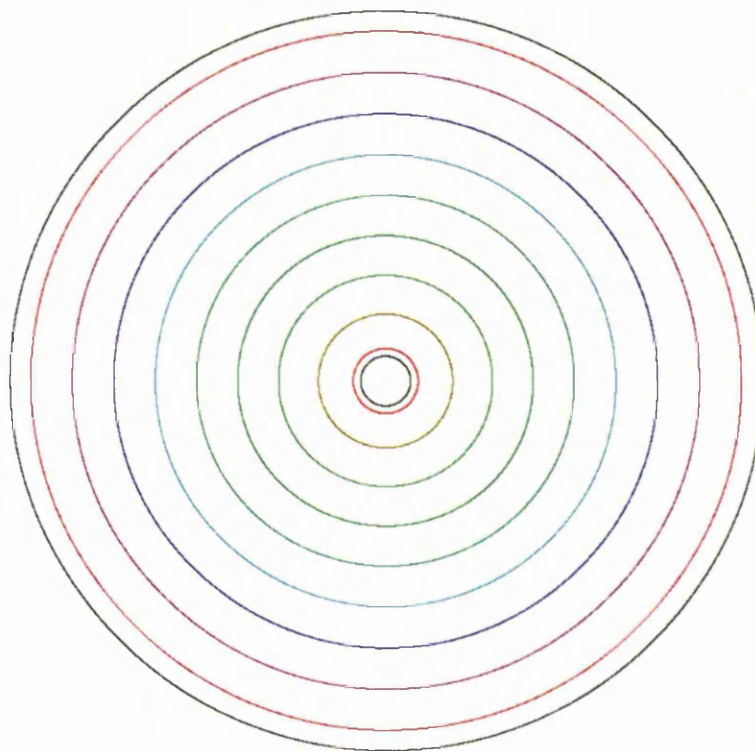


Figure 1.4: The orbits of nine points at various distances from a circular caustic displayed in different colours.

the drum, most notably points with complex chaotic orbital structures as well as strongly structured orbits. For granular materials the region within the drum is of primary interest. Figure 1.7 takes a closer look at the orbits present within the drum itself and attempts to identify specific regions of interest. There are three distinct regions within the drum. First of all there is the core of revolution. Note that the regions of the caustic beyond the self intersection points are not included within this core region. The next distinct region contains the bulk of the drum. Particle paths within this region appear to have covered this entire area. Note that the paths shown represent the orbits of nine points on a straight line from the caustic to the edge of the drum. Given that this region is generated by the orbits

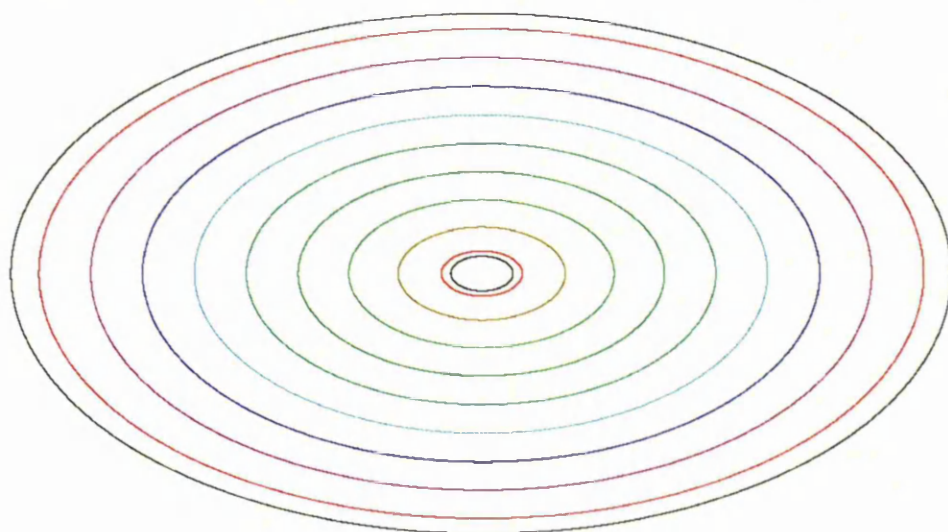


Figure 1.5: The orbits of nine points at various distances from an elliptical caustic displayed in different colours.

of only six points the orbital behaviour within this region is chaotic. The third and final region is the set of three stable orbit families, presenting themselves as concentric triangular regions aligned with the corners of the drum. Though these families lie on lines rather than extended regions as is the case for the chaotic orbit over the bulk of the drum the period of these orbit families is large, if not infinite, with a single point at the centre of each family of period three. The presence of space filling orbits, line filling orbits and period three points all point to chaotic behaviour of various types throughout the drum.

At a fill level of exactly 50% each point's orbit is period two. It is transported to its counter part at the other side of the drum and back again. At fill levels very close to 50% the majority of points proceed very slowly around the drum in concentric orbits. To further illustrate the behaviour, figure 1.8 shows the orbits at various fill levels including some near 50%. At all fill levels illustrated the chaotic region still appears to be present. It also appears that the triangular regions



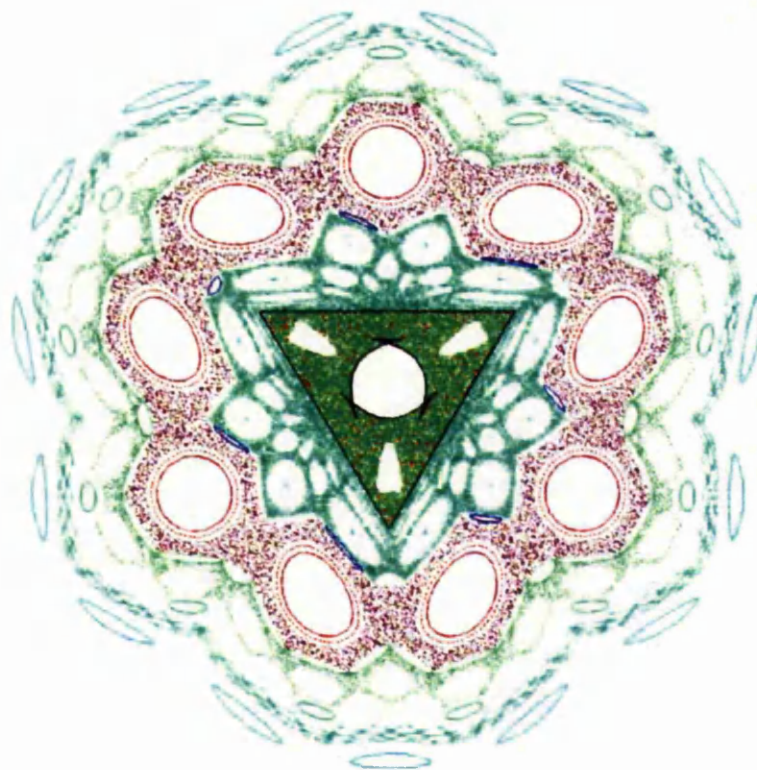


Figure 1.6: The orbits of several points at various distances from the caustic, including several outside the drum, displayed in different colours with the drum outline and caustic overloaded in black.

aligned with the corners become increasingly complex as the fill level approaches 50%.

The results presented are striking similar to the work in [EV99]. The major difference here is that in this case the method was “caustic centric” rather than drum centric which allowed for more general treatment paralleling outer billiard map. Long time orbits are also presented at all fill levels, rather than having a primary focus around 50%. In practice both methods are capable of producing identical results but are arrived at via a different thought processes.

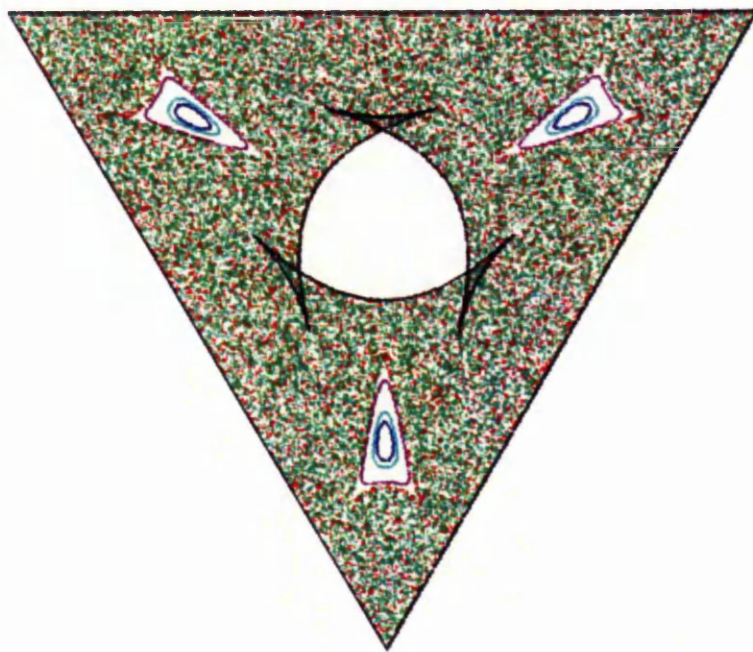


Figure 1.7: The orbits of nine points at various distances from the caustic, including only points within the drum, coloured as in figure 1.6.

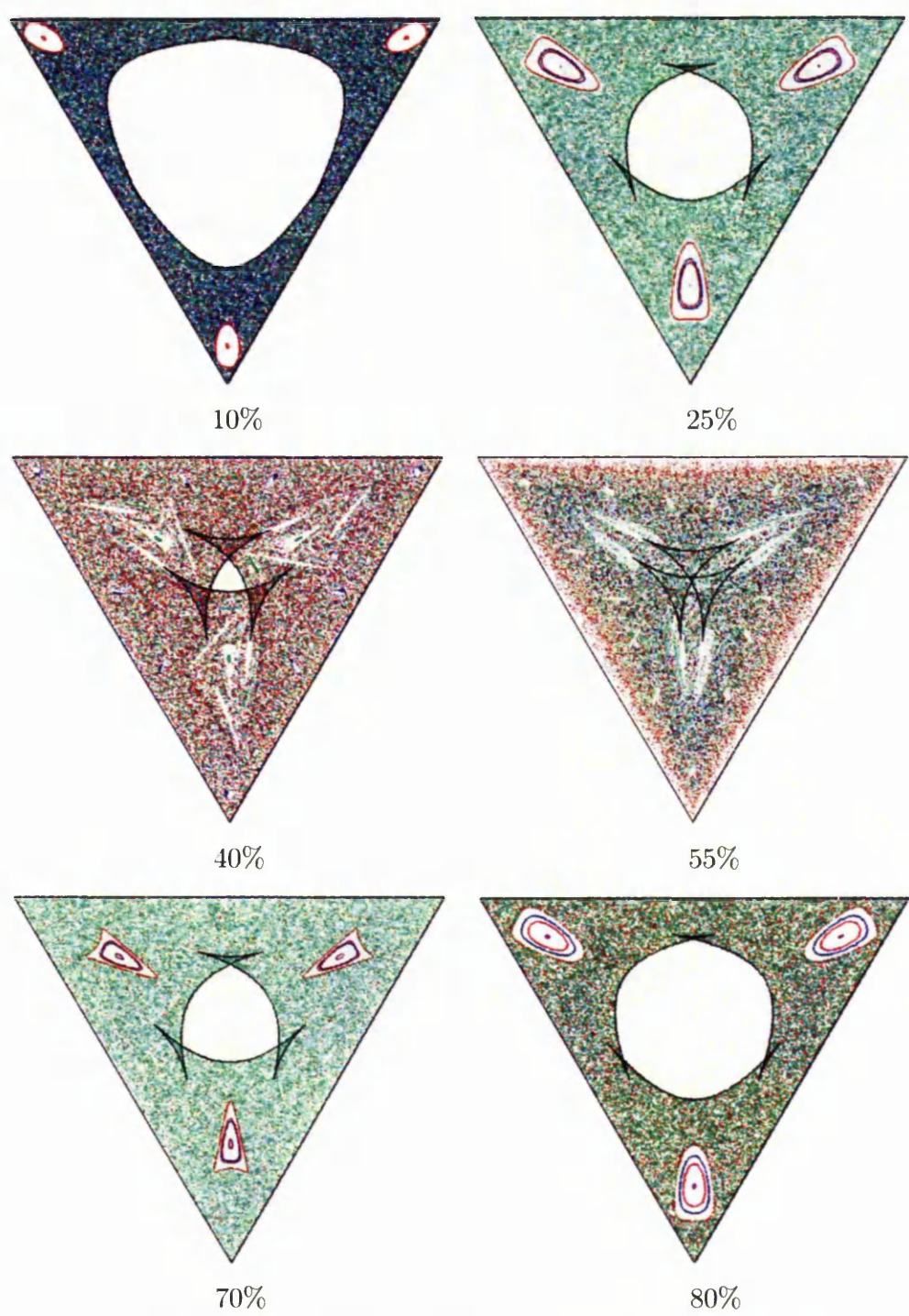


Figure 1.8: Figures of the orbits of ten points at various distances from the caustic overlaid onto a diagram of the drum and caustic at various fill levels.



# Chapter 2

## Segregation in granular avalanches

### 2.1 An introduction to segregation

When granular materials with differing properties flow together, the constituents may separate. This separation is known as segregation. The materials may differ in various ways such as density, particle size and particle shape. Various mechanisms for this process have been proposed and described, but in the flows that shall be considered the primary mechanism is kinetic sieving [SL88].

This segregation mechanism is present in dense fluid like flows. As the material flows spaces of various sizes open and close within the material. For some mean aperture size the probability of a small particle falling into the opening is greater than that for a large. For this reason the smaller particle species migrates toward the bottom of the flow displacing the larger particles. If the size difference is great enough the small particles may percolate through the large even when at rest [EJK<sup>+</sup>95].

Particle size segregation in dense flows is commonly observed in industrial processes [Wil88, SDS01]. In some cases the segregation is considered to be beneficial but often it is an unwanted and uncontrollable factor. When in motion this mechanism is constantly driving the system towards a segregated state and as the system is brought to rest this structure is “frozen” into the static formation.

## 2.2 The avalanche

Models for simulating segregation within granular avalanches are given in [GT05, TGH06, GC06]. In particular the method given in [GC06] is used. Given a velocity profile in a flow, these models can predict the segregation of particles:

$$\frac{\partial \phi}{\partial t} + \frac{\partial}{\partial x}(\phi u) + \frac{\partial}{\partial y}(\phi v) + \frac{\partial}{\partial z}(\phi w) - \frac{\partial}{\partial z}(S_r \phi(1 - \phi)) = \frac{\partial}{\partial z}\left(D_r \frac{\partial \phi}{\partial z}\right), \quad (2.1)$$

where  $\phi$  is the concentration of the small particles per unit granular volume,  $u$ ,  $v$  and  $w$  are the components of velocity in the down-slope, cross-slope and normal directions  $x, y, z$ ,  $S_r$  is the segregation velocity and  $D_r$  is the rate of diffusion. The equation is presented in a non-dimensional form whereby  $\hat{x} = Lx$  and  $\hat{z} = Hz$  with  $L$  and  $H$  being a typical length and depth of the avalanche. Figure 2.1 illustrates all the system’s parameters. In essence the equation is a straightforward advection diffusion equation, though diffusion is limited to one axis, with an additional non-linear segregation term.

Assuming that the bulk flow is aligned with the free-surface, i.e.  $v = 0$  and  $w = 0$ , the segregation velocity  $S_r$  is constant and there is no diffusive remixing,  $D_r = 0$ , then the equation reduces to the hyperbolic segregation equation,

$$\frac{\partial \phi}{\partial t} + \frac{\partial}{\partial x}(\phi u) - S_r \frac{\partial}{\partial z}(\phi(1 - \phi)) = 0. \quad (2.2)$$

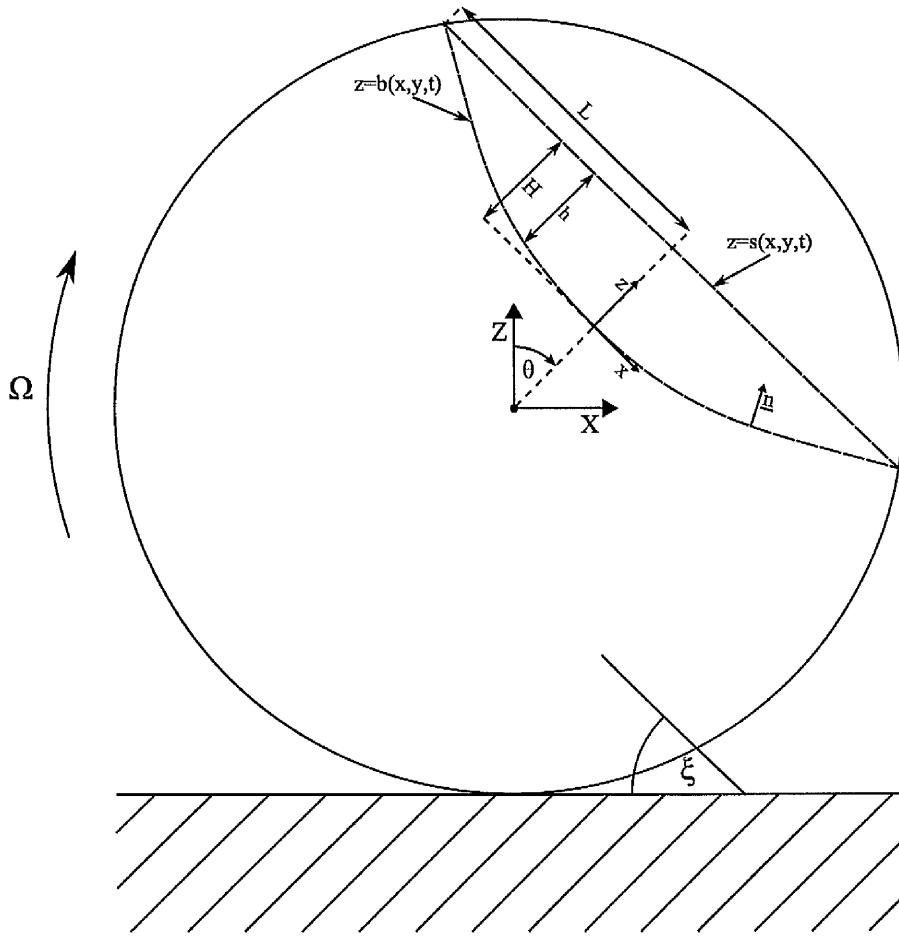


Figure 2.1: Overview of drum and avalanche parameters. Note that  $\xi$  is the angle of dynamic repose of the avalanche.

This must be solved in the avalanche domain subject to the condition that there is no transport of material through the free surface, ie.

$$\phi(1 - \phi) = 0 \text{ at } z = s(x, y, t). \quad (2.3)$$

The lower boundary however, must be considered somewhat more carefully. The governing equation below the avalanching region is that of rigid body rotation. Within the avalanche however the velocity profile and set of governing equations are different. This requires a jump condition for the interface.

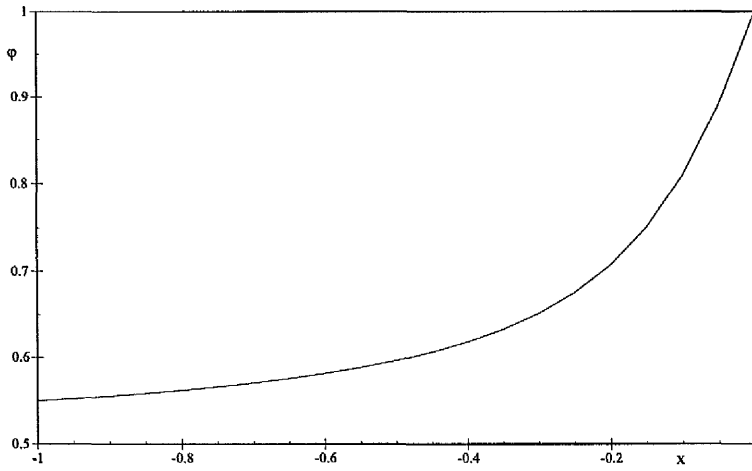


Figure 2.2: A graph of the incoming material concentration across the incoming portion of the basal interface for  $\Phi = 0.5$  and  $S_r = 0.5$ .

A constant concentration in-flow, subject to the jump condition as used in [Gra01] is adopted:

$$\begin{aligned} \text{a) } \llbracket \rho(\mathbf{u} \cdot \mathbf{n} - v_n) \rrbracket &= 0 \\ \text{b) } \llbracket \rho\phi(\mathbf{u} \cdot \mathbf{n} - v_n) \rrbracket &= 0 \end{aligned} \quad (2.4)$$

where  $\rho$  is the fixed density of the material,  $\mathbf{u}$  is the material velocity,  $\mathbf{n}$  is the normal to the interface,  $v_n$  is the normal velocity of the interface and the jump bracket,  $\llbracket f \rrbracket = f^+ - f^-$ , where  $f^+$  and  $f^-$  represent the value of the function on either side of the jump. This jump ensures that the masses of the respective species are conserved. These jump conditions along with the prescribed velocity profiles, plug or shearing flow inside the avalanche and rigid body rotation elsewhere, yield an incoming material concentration of the form

$$\frac{\sqrt{(x + S_r)^2 - 4S_r x \Phi} + x + S_r}{2S_r}, \quad (2.5)$$

where  $\Phi$  is a parameter that specifies the average value of  $\phi$  for the incoming material. Figure 2.2 illustrates its functional form.

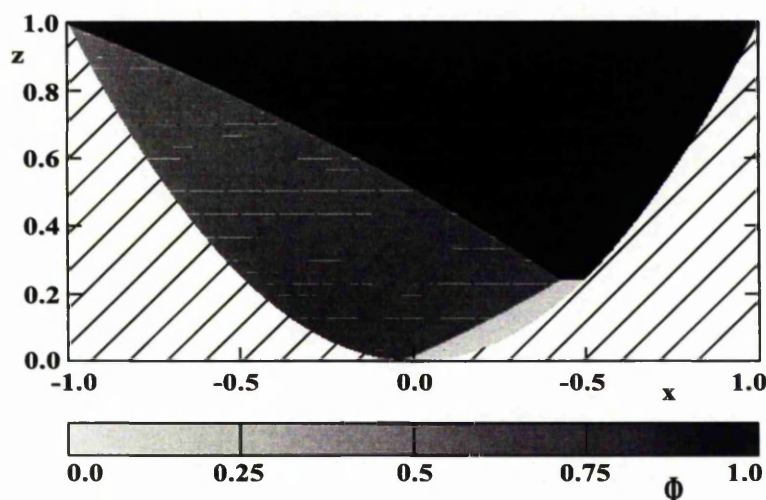


Figure 2.3: NOCD simulation of avalanching region with parabolic base, in-flow determined by equation 2.4 and velocity profile a uniform plug flow, having reached a steady state. Note that the hashed region is not a part of the avalanche.

These equations do not admit exact solutions for non trivial cases. A numerical method for solution must therefore be used. The method chosen is a non oscillatory central difference scheme (NOCD) as detailed in [JLL<sup>+</sup>98] but implemented on a staggered grid. In contrast to some other NOCD schemes, this scheme is genuinely two-dimensional. This allows for a relatively efficient and straightforward numerical implementation while maintaining good accuracy levels. Its staggered nature avoids the need for the back projection step that's required by non staggered NOCD methods.

Figure 2.3 illustrates a solution using the in-flow conditions determined by the jump condition, equation 2.4, and a constant velocity profile known as a plug flow. For the purposes of illustration a segregation rate  $S_r = 1$  and velocity  $u = 1$  are used and the system is allowed to reach a steady state before the results are extracted; the full source code used to generate this solution is provided in appendix A. The small particles migrate downwards displacing the larger particles

upwards within the flow region. This structure with large particles on top of small is typically referred to as an inverse-grading. This distribution is transported to the out-flow boundary, resulting in small particles being deposited near the centre of the avalanching region and large particles towards the edge. An alternative velocity profile will now be considered:

$$u = \alpha + 2(1 - \alpha)\frac{z}{h(x)}, \quad 0 \leq \alpha \leq 1, \quad (2.6)$$

where the parameter  $\alpha$  generates the constant profile for  $\alpha = 1$ , a simple shearing flow for  $\alpha = 0$  and a linear shear flow with basal slip for all other values. As before  $v = 0$ , however in order for mass to be conserved, the normal velocity  $w$  is calculated for the incompressibility condition

$$\frac{\partial u}{\partial x} + \frac{\partial w}{\partial z} = 0. \quad (2.7)$$

Substituting 2.6 in and evaluating the derivative yields

$$-\frac{2(1 - \alpha)z}{h^2} \frac{\partial h}{\partial x} + \frac{\partial w}{\partial z} = 0. \quad (2.8)$$

Integrating with respect to  $z$  and assuming no flow through the surface,

$$w = \frac{(1 - \alpha)z^2}{h^2} \frac{\partial h}{\partial x}. \quad (2.9)$$

This is believed to more accurately reflect the velocity profile of a granular flow over a base at which there is erosion and deposition, as is the case here. Employing the same numerical method as before, figure 2.4 is obtained. Though the bulk of the flow is somewhat different, there is the same qualitative structure. The incoming material is segregated fully before it is deposited.

As with the in-flow jump condition there is an equivalent out-flow jump, i.e. equation 2.4 with the same parameters within the avalanche but rotation corresponding to deposition. The nature of the out-flow material however yields a

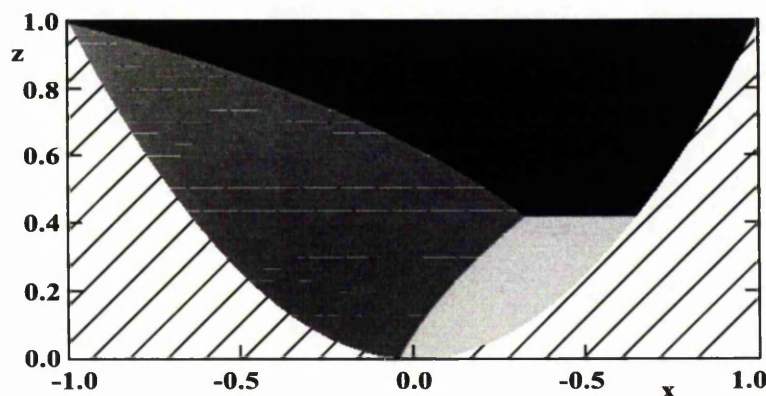


Figure 2.4: NOCD simulation of avalanching region with parabolic base, in-flow determined by equation 2.4 and velocity profile determined by equation 2.6 with  $\alpha = 0.3$ , having reached a steady state. Note that the hashed region is not a part of the flow. Shaded as in figure 2.3.

somewhat different structure. The pure phases are simply advected within the avalanching region as is the case in the rotating bulk. This implies that only material that is not in the pure phase will have its distribution altered. In practice even for strongly segregating material there is some gradation at the interface due to diffusion, but for the sake of clarity this is omitted from the plots.

In order to determine the appropriateness of choice of segregation rate compared to experimental systems,  $S_r$ 's functional form must be examined. The rate of segregation must be proportional to the bulk downstream transport time scale as no motion would require an infinite segregation rate. The rate must also be inversely proportional to the time scale for segregation. Due to the normalised coordinates, this obtains

$$S_r \sim \frac{qL}{HU}, \quad (2.10)$$

where  $U$  is a typical bulk velocity and  $q$  is the segregation velocity. Clearly all the parameters can be obtained directly, except  $q$  so in practice one may directly measure  $S_r$  based on the segregation behaviour observed. To determine if the

value is relevant the real experimental set-up must be observed. After a single pass through the avalanche the material appears to be fully segregated. Additionally the aspect ratio is very long compared to the avalanches depth. This implies that the segregation rate must be relatively high, specifically high enough for complete segregation to occur within the avalanche. Based on the globally similar behaviour between what is observed experimentally and predicted in the simulation, the value of  $S_r$  used must be at least physically relevant.



## Chapter 3

# A mapping approach to segregation in rotating drums

### 3.1 An introduction to segregation in rotating drums

Bringing together the rotating drum and size segregation from chapters 1 and 2 adds a further dimension to the behaviour. As has been discussed, the mixing of materials in a rotating drum is a common industrial process. Typically the intention is to provide a complete homogenisation of several components. In many cases however, differing material properties of the individual species cause the material to segregate.

Despite the chaotic nature of the individual particle paths, the segregation taking place imposes a strong and regular ordering on the system. In particular intricate lobe and arm structures are formed. Some experimental and computational treatment has been carried out in the past [HKG<sup>+</sup>99]. Several mathematical

approaches have been suggested but none provide a complete or intuitive overview of the system's behaviour and dynamics [MLJ07].

## 3.2 Theory

There are many approaches one may take to simulating segregation of granular materials in a rotating drum, ranging in technical complexity, as well as in the level of complexity of behaviour they may encapsulate. A first order approximation that provides the maximum in predictive power while remaining simple enough to allow for intuitive comparison and insight into the system is sought. To this end the system as a whole is considered.

The system consists of a lower rigid body rotating region that simply advects the material around. The thin surface region consists of a dynamic avalanche, the volume of which is fixed by in- and out-flow of material via erosion from and deposition to the rotating bulk. The long aspect ratio of the avalanching region suggests that the avalanche thickness may be insignificant. The limiting case as the avalanche's thickness tends to zero is hence considered.

### 3.2.1 The mapping approach

It is important for any such approach to work that the avalanching is continuous while not being overly energetic, that is to say the avalanche must not stop and start and particles must not take flight as they reach the surface. This choice of regime will become important when developing the mapping. In order to carry out a mapping the domain and co-domain must be established, and the precise nature of the mapping required determined. To ensure that the chosen mapping

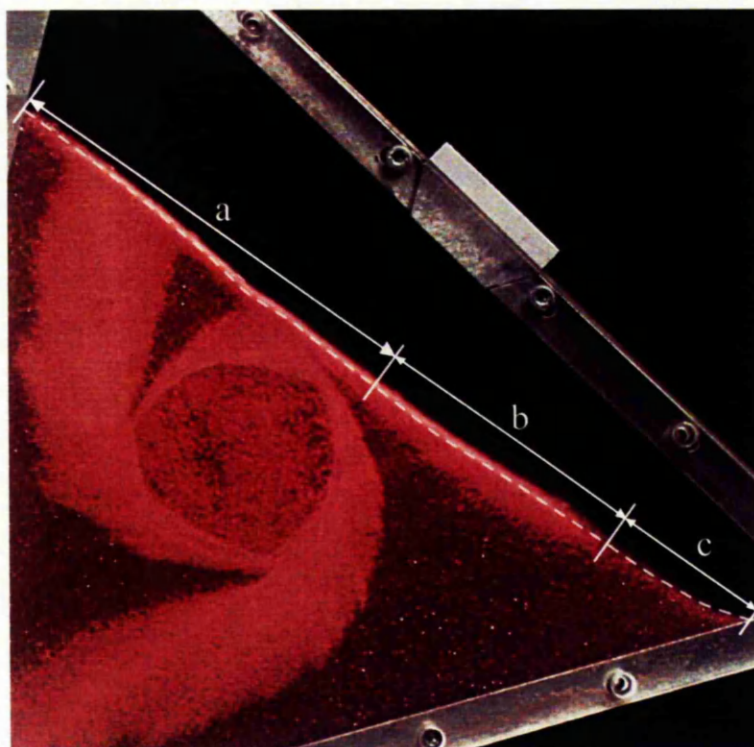


Figure 3.1: A long exposure image of a typical rotating drum experiment. The dotted line indicates the base of the avalanching region. Below this point the particles are well resolved, above this they are blurred. Region *a* corresponds to the in-flow region, *b* to the out-flow of small particles and *c* to the out-flow of large particles.

matches real systems, a typical system must be examined. Figure 3.1 shows a long exposure image of the surface and helps to illustrate that the movement within the surface avalanche region is extremely rapid compared to that of the rotating bulk below the surface. It is also observed, that the segregation of interest only takes place when the particles are moving within the avalanche. The granular portion of the system has two regions: A rapidly moving avalanche above the dotted line, and a large region of material rigid body rotating below. There are three interfaces of note: The free surface; The up-slope region, part *a* of the dotted line and the down-slope region, parts *b* and *c* of the dotted line. As the drum rotates

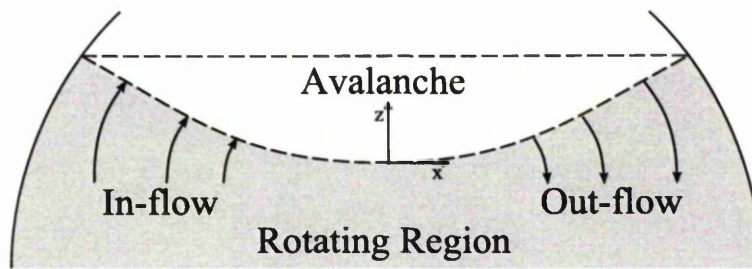


Figure 3.2: Close-up view of avalanching region.

material from the up-slope portion breaks off at some critical angle, known as the angle of dynamic repose, and is eroded into the avalanching region; material from the avalanching region proceeds down-slope and is frozen, as it comes into contact with the rotating bulk below, hence being deposited. The difference in velocities of motion in the rotating bulk compared to the avalanching region as well as the small thickness of the avalanching region compared to other macroscopic scales, allow a mapping approach to be employed to simulate the movement and segregation of particles through the surface avalanche. Also of note, is that the avalanching region is very thin perpendicular to its bulk flow; this supports the supposition that a mapping may be an appropriate method to apply. The domain of the mapping must therefore be the up-slope portion of the avalanche and the co-domain must be the down-slope portion.

Having decided where the mapping should be placed, the exact form of the mapping must be established. As observed previously the material is segregated as it flows through the avalanche. Figure 3.2 takes a microscopic view of the situation, highlighting two significant points. The only in- and out-flow regions are the interfaces of the avalanche and the rigid body rotating region below, up and down-slope respectively. Secondly, the avalanche itself is the only region in which particle size segregation takes place leading to an inverse-grading. The radial

segregation must thus be related to the segregation within the avalanche. In order to relate these two phenomena the shape of the avalanche must be considered. In practice the avalanche itself has a slight 'S' shape but for these purposes the free-surface is considered to be flat. Since the mapping takes place on a line the basal topology is of no longer of relevance.

### 3.3 Methods

Rather than providing details of the destination of particular particles, the mapping approach will provide a global picture of the system. For any given time, which in this case is synonymous with the angle  $\theta$ , the interface between the large and small particles is of primary interest. For many problems it proves complex to exactly locate the surface. A computational method is therefore developed.

#### 3.3.1 Locating the surface

In order to implement the mapping computationally it is necessary to be able to, given an arbitrary convex drum, fill level  $A$  and rotation angle  $\theta$ , locate the avalanching surface. Before moving on to the details of the actual surface location process the computational representation of the drum must be considered. For these purposes the drum is represented as a cyclic, ordered, doubly linked list of vertices. That is, the last and first element are connected. The vertices are stored in the order they appear around the drum and one may iterate through the list in either direction. This allows trivial traversal of the list in either direction returning to the point of origin. Since the list is cyclic there is nothing special about the first or last vertex in the list. The surface of the drum will be prescribed by the

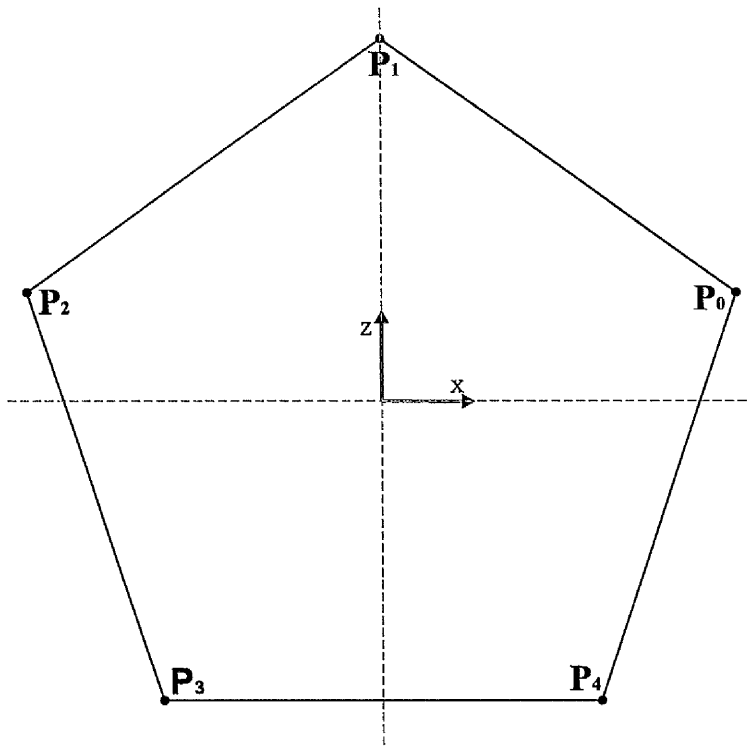
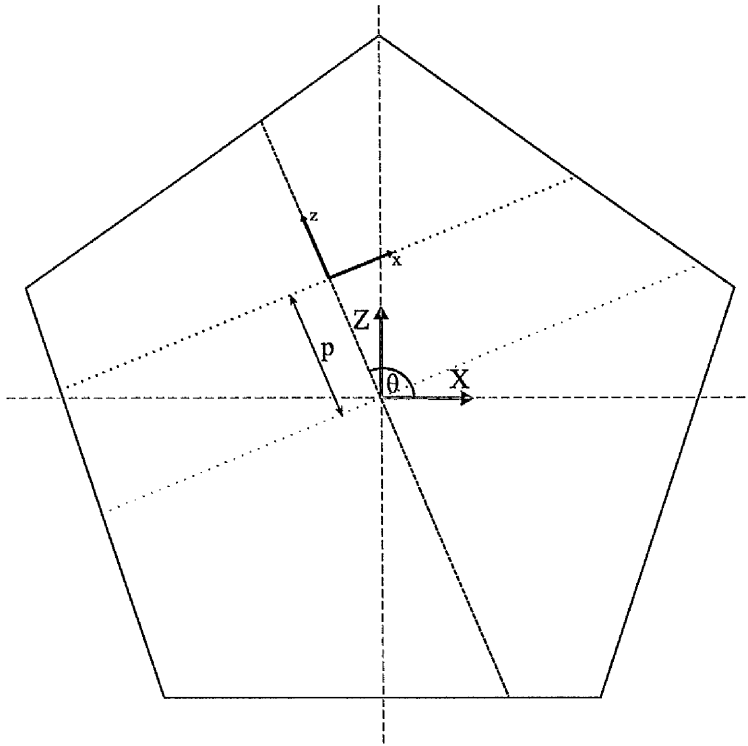


Figure 3.3: Typical polygon vertex labelling.

line segments connecting adjacent vertices. These vertices  $P_n$ , will be labelled in an anticlockwise order. A typical configuration is illustrated in figure 3.3.

For a given angle  $\theta$  the surface must now be located. The surface lies perpendicular to the radial line at  $\theta$ , as illustrated in figure 3.4. The problem has been reduced to locating a single parameter  $p$ , being the distance of that given surface along the  $\theta$  line in the positive  $\theta$  direction.

In order to pick the correct surface out of the family obtained the fill fraction of the drum must be considered. The fill fraction,  $A$ , determines the ratio of the total area of the drum to the area filled with material. Each surface in the family corresponds to a unique fill fraction of the drum. The surface required may hence be located. For a general figure however, the formula representing the fill fraction

Figure 3.4: Typical surface for a given  $\theta$ .

for a given  $p$  is not trivial, moreover it is determined piecewise for an  $n$  sided figure, between  $\frac{n}{2}$  and  $n - 1$  pieces. For a general polygon this may not be too large, however for a figure that is a piecewise approximation to a generic convex curve, this may prove very large. The location of the solution within these piecewise linear function segments must be established. A discrete set of surfaces from this family, namely those which intersect a vertex of the figure, are considered. These vertices represent the extents of the piecewise fill fraction functions. For computational efficiency, they are considered in ascending numerical order  $V_n$ , where the sort parameter is  $p$ . The fill fraction below each of these surfaces is then calculated by adding the area of the parallelogram formed directly below to a running sum which was set to zero at  $V_0$ . This summing continues until a fill fraction greater

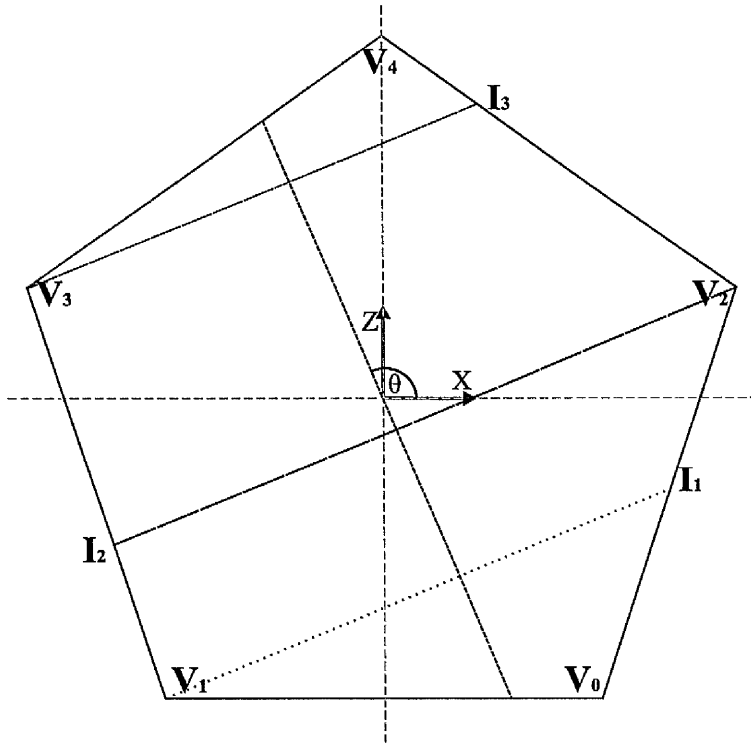


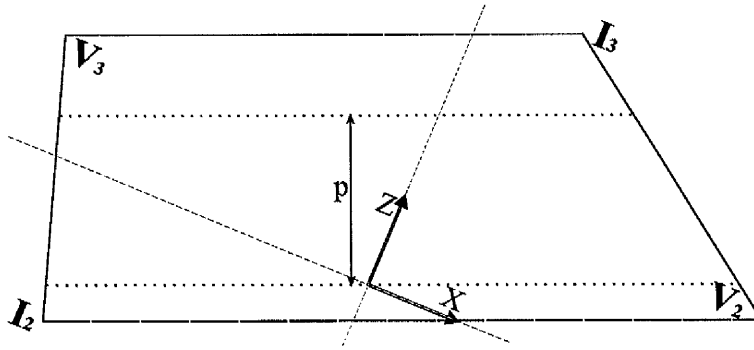
Figure 3.5: The upper and lower bounds for search parameter  $p$ .  $V_n$  represent the vertices numbered from bottom to top relative to the surface,  $I_n$  represented the corresponding intersection vertices.

than that required is found. Figure 3.5 illustrates the some typical regions.

The true surface must thus lie within this region as illustrated in figure 3.6. A quadratic formula for the fill fraction within this region can be straightforwardly obtained. The region is necessarily a trapezium as the intersection lines are all parallel. If the region considered lies at the top or bottom of the drum one side of the trapezium is degenerate; this yields no further complications. The length of the upper and lower parallel sides are  $a$  and  $b$  respectively and the distance up the trapezium is parametrised with parameter  $t$ .

$$\frac{t}{2}(2b + (a - b)t) = 0 \quad (3.1)$$



Figure 3.6: Quadrilateral region in which  $p$  must lie.

must hence be solved. The resulting formula is quadratic and thus yields two real solutions. One solution corresponds to the solution required, and the other corresponds to a projection of the line segment edges outside the domain of interest. There is therefore only one solution such that  $t \in [0, 1]$ . The exact solution can hence be found for the chosen parameters.

The approach developed to locate the surface may be applied equally well to the billiard mapping scheme in section 1.3. The billiard mapping approach may hence be applied to any convex polygon's caustic.

### 3.3.2 Particles on a string

Having developed a generic method of locating the surface for any given angle the mapping may be implemented. Once again the experiment, figure 3.7, is examined for guidance. By conservation of mass the surface must proceed continuously around the drum with respect to  $\theta$ . Also note that for a continuously varying incoming distribution of material, the outgoing material must also be continuous. The segregation interface extends all the way from the in-flow interface of the avalanche, to the out-flow. For a typical configuration, i.e. one where the material

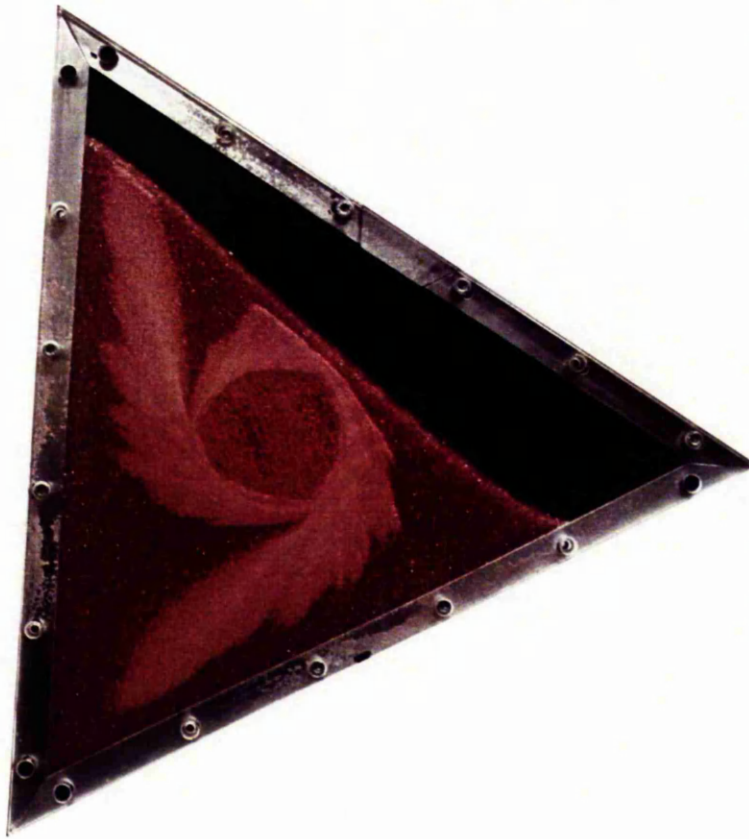


Figure 3.7: Image of a typical experiment at an intermediate stage.

has had a chance to flow around the drum at least once, the segregation interface intersects the in-flow region of the avalanche at least once. Unless the incoming material is in a pure phase, the segregation interface may not intersect the wall of the drum. These facts prove important in developing this method.

The interface will be represented in much the same way as the drum: A doubly linked list. For any physically relevant combination of convex drum and segregation interface, the change of the segregation interface by the transport of material through the avalanche must be predicted using a mapping method. There are, in fact, several issues that are at this stage not apparent so dealing with the incoming

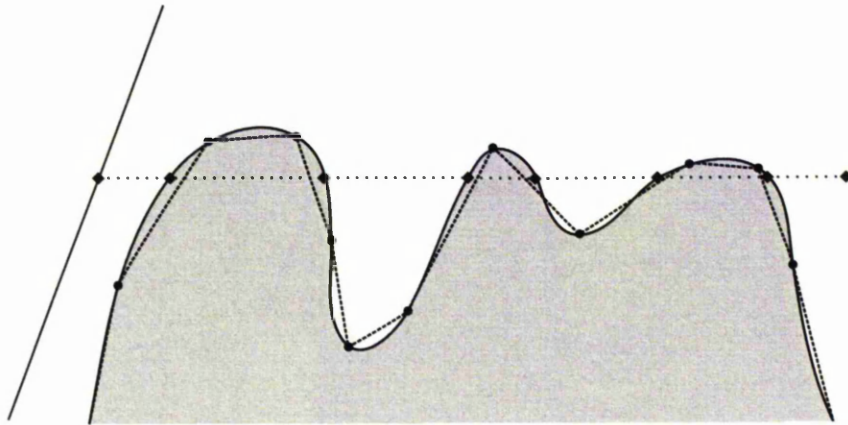


Figure 3.8: Segregation interface and surface intersections. A true interface is indicated with a solid line. The vertices and the piecewise linear reconstruction of the interface are illustrated with a dashed line. The intersections of the interface and the surface are indicated with diamonds.

material will be the initial focus.

As with the drum, the curve is defined by the piecewise linear connection of an ordered list of vertices. For a given  $\theta$  the intersection of each line segment and the surface is found. These intersection vertices are stored in a second list then sorted using the distance from the wall of the drum as the sort parameter. When all such intersections are located, a vertex representing the intersection of the surface and the wall is added to the beginning of the list and a vertex representing the centre of the avalanche is added to the end. Each intersection represents a change in composition of the material beyond that vertex. This is illustrated in figure 3.8. One could conceive of several special cases which might need specific treatment. If the segregation interface is tangent to the surface at a single point, an intersection will be detected but in reality there is no change in material composition to either side. An equally concerning case is when two or more line segments of the segregation interface lie on the surface. Again one might presume a special treatment would be required for this case. These issues can be

avoided if each line segment's intersections are considered separately. To avoid points being considered twice, one typically parametrises each line segment with a parameter, say  $t$ , whereby  $t = 0$  represents the initial point and  $t = 1$  represents the final point. For each line segment, with potential special cases for the first and last line segment, consider  $t \in [0, 1)$ . This ensures that if a vertex coincides with the curve of interest, only one intersection is indicated. All the special cases can be dealt with trivially by a simple change to these intervals, and a second minor check. Only values in  $t \in [0, 1]$  are admitted, meaning that if the vertex being considered is a turning point, or the beginning or end of a region of coincidence, two intersection values are obtained. Though this is not physically meaningful, it allows the continued assumption that the material composition alternates at intersections. There is one additional check required when the intersection is at  $t = 1$ . The next vertex must lie on the opposite side of the line to  $t = 1$  and if this is the case, it may not be included in the intersection list. With these two special treatments, a usable and practically meaningful intersection list can be obtained.

The small particle concentration  $\phi$  must now be integrated over the up-slope region of the avalanche. There is however, another consideration to make. The incoming material is being transported by rigid body rotation. For a general surface,  $x_a < x < x_b$ , the normal flux of small particles and the total particle flux across an interface with normal velocity  $V_n$  due to a vector field  $u$  are

$$\begin{aligned} \text{a)} \quad & \int_{x_a}^{x_b} \phi(\mathbf{u} \cdot \mathbf{n} - V_n) dx = 0 \\ \text{b)} \quad & \int_{x_a}^{x_b} \mathbf{u} \cdot \mathbf{n} - V_n dx = 0 \end{aligned} \quad , \quad (3.2)$$

In this case the velocity field is a rigid body rotation at a rate  $\Omega$  and the surface normal is  $\mathbf{n} = (0, 1)$  therefore,

$$\mathbf{u} \cdot \mathbf{n} = \Omega x. \quad (3.3)$$

Substituting this back into equation 3.2 a and b, implies

$$\begin{aligned} \text{a)} \quad \int_{x_a}^{x_b} \phi(\Omega x - V_n) dx &= 0 \\ \text{b)} \quad \int_{x_a}^{x_b} \Omega x - V_n dx &= 0 \end{aligned} \quad (3.4)$$

Integrating the mass balance equation, b, and solving for  $V_n$  yields

$$V_n = \Omega \left( \frac{x_a + x_b}{2} \right). \quad (3.5)$$

Clearly  $(x_a + x_b)/2$  corresponds to the centre of the interface, henceforth  $x_c$ , thus

$$V_n = \Omega x_c. \quad (3.6)$$

Substituting this back into the small particle flux formula, equation 3.4 a

$$\Omega \int_{x_a}^{x_b} \phi(x - x_c) dx = 0. \quad (3.7)$$

Considering the incoming material only,

$$\Omega \int_{x_a}^{x_c} \phi(x - x_c) dx. \quad (3.8)$$

To simplify the formula slightly the following substitution is applied

$$t = 2 \frac{x - x_c}{x_a - x_b} \quad (3.9)$$

Substituting back into expression 3.8

$$-\frac{(x_a - x_b)^2}{4} \Omega \int_0^1 \phi t dt. \quad (3.10)$$

Applying the same logic to the out flowing material, the in-flowing and out-flowing material, which must match due to the balancing constraints, can be parametrised as

$$F = \int_0^1 t \phi_{in}(t) dt \quad (3.11)$$

which shall be referred to as the “normalised small particle flux” through the appropriate surface region. Where

$$\phi_{in}(t) = \phi\left(\frac{xt}{L}\right) \quad (3.12)$$

Computationally this integration can be carried out for the in-flow region, simply by integrating the regions between each intersection, taking  $\phi_{in}$  to be alternately 0 and 1 over each interval.

The corresponding outgoing  $\phi_{out}(t)$  will represent the fully segregated material. This function will however take a special form due to the segregation process. Since the large particles end up at the outer edge of the drum and the segregation is sharp two distinct regions will be present.

$$\phi_{out}(t) = \begin{cases} 1 & \text{for } t \in [0, T) \\ 0 & \text{for } t \in [T, 1] \end{cases}, \quad (3.13)$$

where  $T$  is the parameter indicating where the segregation interface lies on the outgoing interface. Thus equation 3.11 is simplified and all that remains is to solve

$$\int_0^T t dt = F. \quad (3.14)$$

The integral may be evaluated explicitly and the equation solved for  $T$  yielding

$$T = \sqrt{2F}. \quad (3.15)$$

Clearly  $F \geq 0$  since it is the positive integral of quantities that are greater than or equal to zero. There is therefore necessarily one solution for  $T$  if  $F = 0$ , or two, one positive and one negative, otherwise. In this case however, the function  $\phi_{out}$  was only defined for  $t \in [0, 1]$ . This means a unique solution is necessarily obtained. This equation can be used directly with the computational implementation and

the value of  $T$  can be mapped onto the down-slope region of the surface resulting in a vertex which can be added to the beginning of the list of line segments defining the segregation interface.

An approach to deal with the addition of vertices in the list must now be considered; An appropriate time to remove the vertices must also be established. The most obvious approach might be to remove every vertex above the interface. Looking back once again to figure 3.8, a potential problem can be seen. If the vertices are removed immediately, the line segment connecting them to the previous point on the interface would be removed. Since it's these line segments that define the interface, there would be no intersection between the segregation interface and the surface. This is required for the algorithm to operate correctly thus an alternative method must be sought. Another approach might be to remove every vertex in the list beyond the first point above the surface. This first vertex can be considered a ghost point to allow it to meaningfully define a line segment without actually being on the curve. Once again a potential problem with this approach presents itself. If the interface passes through the surface at several points then this approach would remove several other, potentially important, ghost points. Theoretically the interface could be searched and any vertices at least one point above the surface could be removed, but this proves to be rather awkward and unnecessarily complicated. A more straightforward approach is taken whereby ghost points may remain until they end up falling below the outgoing interface. This has a potentially useful consequence. The segregation interface is not only defined below the surface of the material where it would be visible, but around the entire drum. It is a feature that initially highlighted the properties that will be introduced in section 4.5.1.

There are several special cases for the list itself that need consideration:

1. Two vertices coincide, figure 3.9 (a).
2. A vertex lies on the surface, figure 3.9 (b).
3. An edge lies on the surface, figure 3.9 (c).

All these cases are dealt with implicitly or explicitly. Figure 3.9 illustrates all these cases.

One last point to consider is the angular increment  $\delta\theta$  used between successive mappings. The most obvious approach might be to choose a  $\delta\theta$  so that the surface intersects the next vertex in the segregation interface list. This appears to be the largest possible increment that would encode the topology of the incoming material. It would also guarantee that the same number of points on the segregation interface as were initially prescribed was maintained. In order to implement such a scheme, one must be able to calculate for which  $\theta$  the surface would intersect any given point. Since the function to locate the surface for an arbitrary figure is potentially extremely complex it is not viable to directly solve this problem. A numerical iteration to obtain the angles is required, and hence the optimal  $\delta\theta$ . However this apparently optimal choice proves to be inadequate. Given the non-linearity of the function defining the location of the surface for a given angle, and the nature of the transport, the first order line segments become curved when transported through a general surface. It is thus not sufficient to merely transport the vertices. In fact there is no truly optimal choice of  $\delta\theta$  that one can make. One could attempt to ensure accurate representation of the segregation interface by first choosing a  $\delta\theta$  using the previous method and comparing the location of the outgoing interface point to the previously placed one. If the distance between these points was above some predetermined threshold value the value of  $\delta\theta$  could



be halved and the check repeated until the distance fell below some threshold. This algorithm also has an unfortunate shortcoming: If there is a discontinuity in the outgoing segregation interface, for example when the incoming segregation was tangent to the surface, one may never obtain a value below the threshold. This would require that a lower limit be placed on  $\delta\theta$  to avoid an infinite loop developing. In practice this method would yield an optimal representation of the outgoing curve, but it would also require numerous evaluations of the surface location and evaluations of the transport integral. In effect the improved representation of the segregation would be offset by the huge increase in runtime required. A fixed  $\delta\theta$  is therefore used, the order of the lower angular increment that might have been chosen for the angular bisection method.

With this “time stepping” method for the segregation interface, the initial state of the segregation interface must be considered. In a real experiment a homogeneous mixture of the two species at some given ratio is sought. The first choice one might make is a purely arbitrary curve, for example a scaled down version of the drum might be chosen as the initial interface, with the scaling factor determined by the ratio of the species in the mixture. If, as experiments have suggested, the interface is convergent in some sense in the long term, the initial form of the interface will have little impact on the long term results. In the short term however, that is to say for the first few revolutions of the drum, this will have introduced a disparity between the predicted segregation interface and the experimentally observed interface. A more refined approach is thus sought.

Considering the experiment once again, the incoming material throughout the first revolution is of fixed composition. Potentially a segregation interface that reflects this may be generated. In effect  $F$  in equation 3.15 may be prescribed.

In this case a curve representing the segregation interface after the first revolution is produced. This may seem like an ideal solution but unfortunately it doesn't necessarily match the real experiment. In practice the material is not of a fixed composition for the entire first revolution. Before one entire revolution is completed the segregated material enters the avalanching region. The interface between the initial homogeneous mixture and the segregated material is not, in general, tangent to the surface. This means that a pure phase of a single species enters the avalanche along with some of the homogeneous mixture as seen in figure 3.10. This results in a discrepancy between the proposed initial segregation interface and that observed in experiment.

Figure 3.11 demonstrates a typical result obtained using the particle on a string method. The full source code used to generate this diagram is provided in appendix B. This method is clearly capable of producing interesting and potentially useful results. Though the results obtained may be useful but it has several shortcomings. It is unable to deal with the formation of "islands" of material and lacks generality in the choice of initial conditions. A more generic method is therefore sought.

### 3.3.3 A finite volume approach

Due to the limitations of the particle on a string method illustrated in section 3.3.2, a more complete representation of the material was required. Rather than merely representing the segregation interface the entire  $\phi$  field would be represented throughout the drum. The representation will take the form of a structured Cartesian grid overlaid over the entire cross-sectional area of the drum. In practice this means that the edge of the drum does not correspond to the edge of grid elements but due to the method chosen and the resolution chosen for typical simulation,

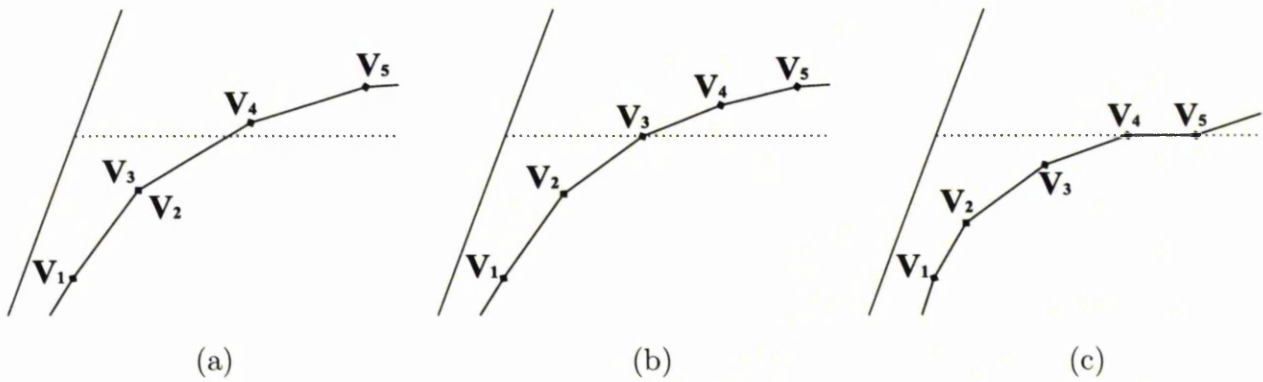


Figure 3.9: Special cases that require treatment in a particle on a string scheme.

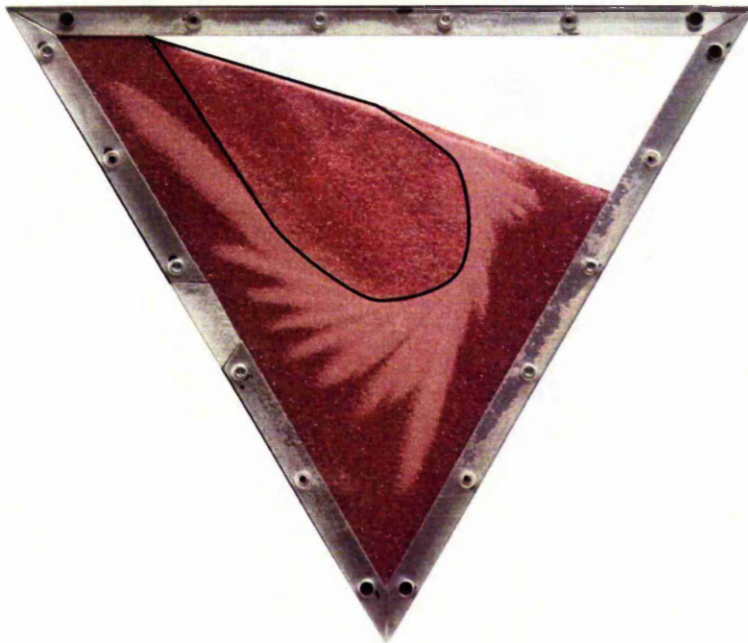


Figure 3.10: An example of the potential problems with the particle on a string approach to simulation. The dark line outlines the material that has not yet passed through the avalanche.

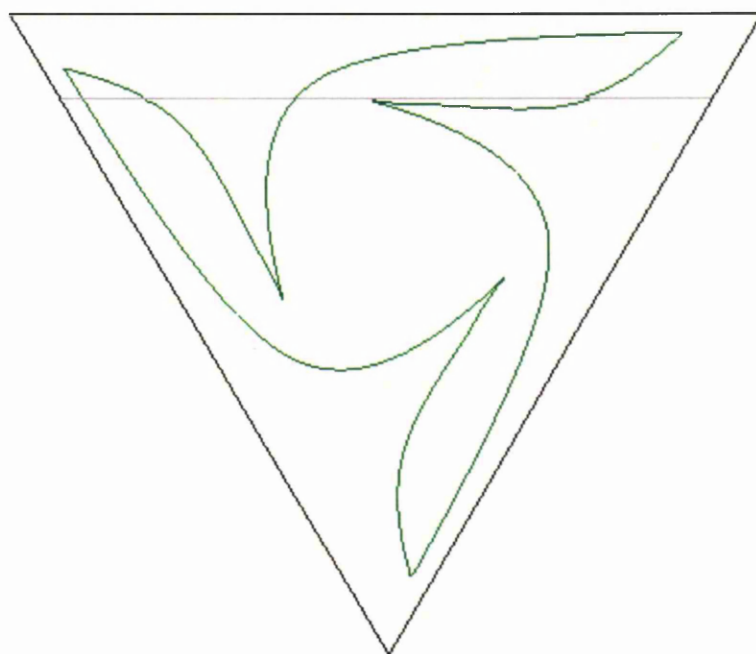


Figure 3.11: An example of the results generated by the particle on a string method with a 75% fill level and a 50% particle ratio. The black line is the drum, the grey line is the current surface position and the green line is the segregation interface.

this will prove irrelevant as the integration is carried out exactly within the grid cells so mass will be conserved and the errors will lie outside the true perimeter of the drum. The only case where this would genuinely be an issue is if the interface were to lie within the outermost grid cell representing the drum, which would only occur if the incoming particles were an almost pure phase of small particles.

The choice of frame of reference in a scheme of this sort becomes critical. Mathematically any frame of reference may be chosen but computationally this is not the case. If a frame of reference fixed with the viewer is chosen, then at every time step, or in this case angular increment, the rigid body rotating bulk below the surface would have to be rotated. On a grid, an arbitrary rotation does not generally identify grid elements to grid elements; one cell would typically span several other cells. To counteract this problem the nearest neighbouring cell might be selected, forcing a one to one correspondence. Clearly, as the result does not correspond to the true rotation, a degree of distortion would be introduced. For a single rotation step this distortion may be acceptable. When the rotation increment is sufficiently small however, there is some interval around the centre of rotation that remains unaltered. Figure 3.12 illustrates the results of repeated application of this method. The cumulative distortion and unrotated region at the centre are clearly visible and unacceptable. To minimise distortion one might take a more refined approach to the rotation. The grid could be rotated and the cells in the newly rotated grid could obtain their values by integrating across the underlying, original, grid elements. This approach would yield a more accurate representation of the rotated material, but it has a shortcoming. In reality the value stored for a grid element is simply an average value for that cell. In integrating, any subregion of that cell similarly reflects that average. Visually this averaging is far less

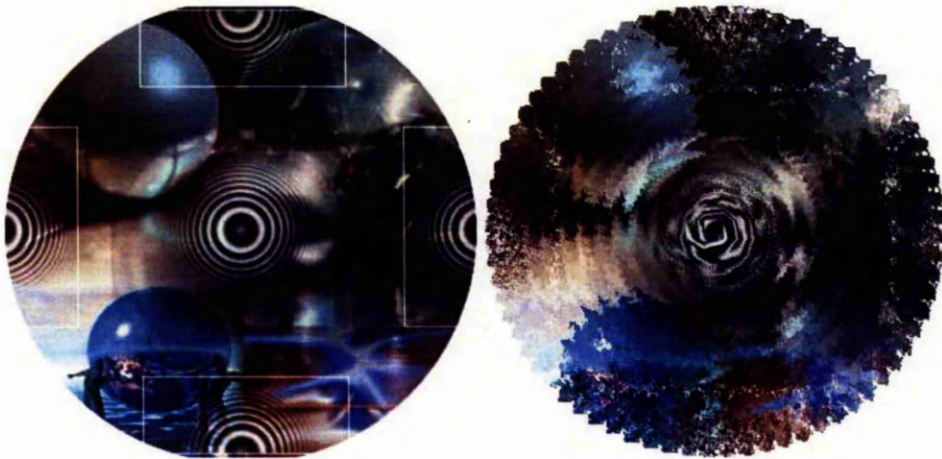


Figure 3.12: A comparison of a typical image and the same image rotated  $360^\circ$  in  $5^\circ$  increments using the nearest neighbour interpolation method. Note that this test image was obtained from <http://www.antigrain.com>.

disturbing than the distortion caused by the previous algorithm for a single step but once again iterating numerous times can cause a problem. Repeated averaging over a gradually shifting window in the data results in all values drifting towards the local average. The meaning of local in this case is determined by the angular increment. Visually this has the effect of smoothing the data as shown in figure 3.13. Given that the data will contain discontinuities, ie. the segregation interface, this is also unacceptable. A frame of reference must therefore be chosen that rotates with the drum. In this frame of reference the rotating bulk of the material is static. A single revolution of the drum will thus result in the surface apparently counter rotating with respect to its original position. The resulting data may be post rotated for the sake of visualisation but this post rotation is carried out in a single step for each image meaning there is no cumulative image distortion or smoothing.

The definition of  $\phi_{x,z}$ , the value of  $\phi$  at a specific grid element, is simply the average value of  $\phi$  in that cell. The values of  $\phi$  will be maintained throughout the



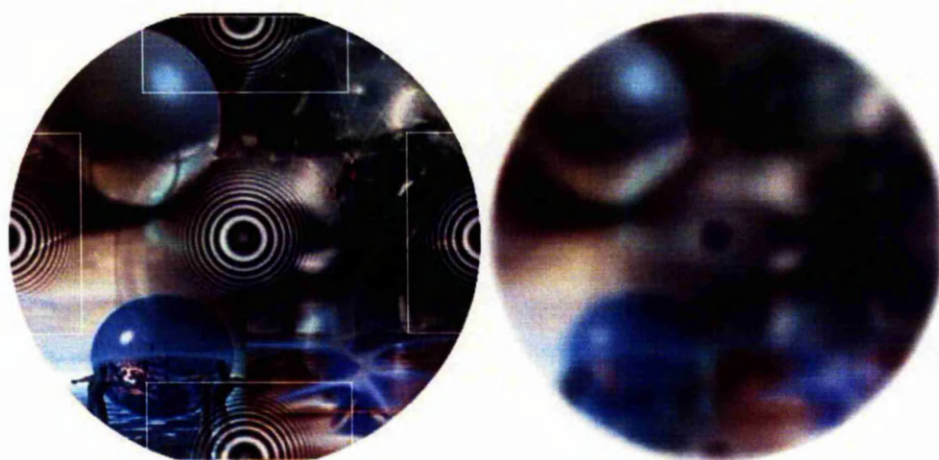


Figure 3.13: A comparison of a typical image and the same image rotated 360° in 5° increments using the integration interpolation method.

grid. This introduces a degree of redundancy in that grid elements exist outside the area represented by the drum, but since the simulation is typically limited by runtime and not memory capacity, this has no impact on performance. As with the particle on a string approach, section 3.3.2, the material will remain defined even after it has passed through the surface. Since the exact location of the surface and the drum is known, the grid cells that do not actually represent material can easily be masked off. In this simulation scheme no special treatment has to be given to this “ghost material”. It will simply be overwritten as appropriate. It is within this framework that the following algorithm will be applied:

1. Locate the surface by method described in section 3.3.1 and store coordinates.
2. Locate the surface at the next angular increment.
3. Integrate  $\phi$  over the polygon formed between the subsequent surfaces on the up-slope side.
4. Fill grid elements in down-slope polygon according to the segregation rule.

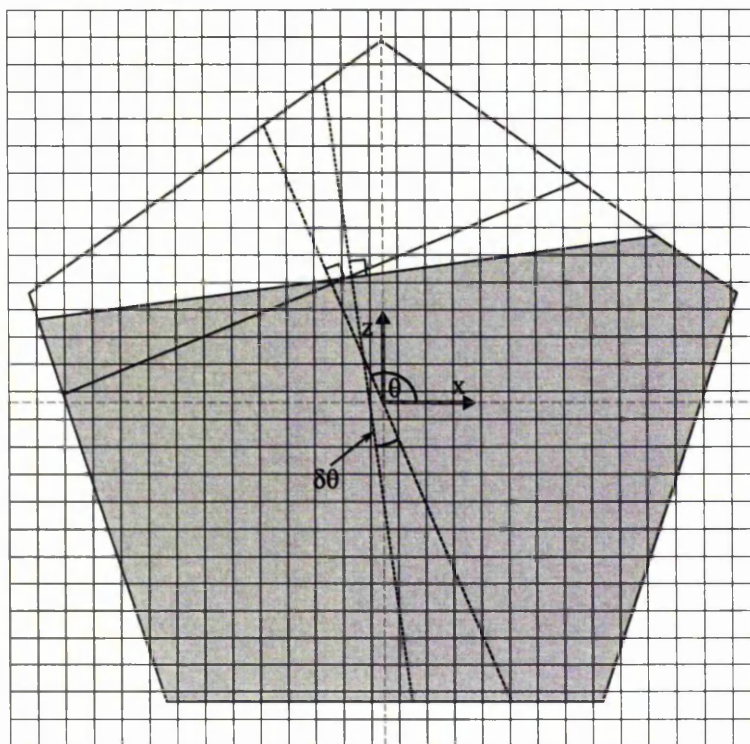


Figure 3.14: An overview of a typical drum with surfaces at  $\theta$  and at  $\theta + \delta\theta$  marked with solid black lines and the material before transport marked in grey.

5. Go to 2.

Having already detailed how to locate the surface for a given angle, section 3.3.1, the up-slope material must now be integrated. Referring to figure 3.14, the material is being transported in what appears to be the up-slope direction. This is merely an artefact of the chosen frame of reference however. For clarity the diagram is representing a uniform mixture of material below the surface. This would thus represent an initial configuration before any segregation has taken place. The algorithm which will be described is equally applicable to arbitrary distributions of material.

An obvious initial approach to this integration might be to simply sum the



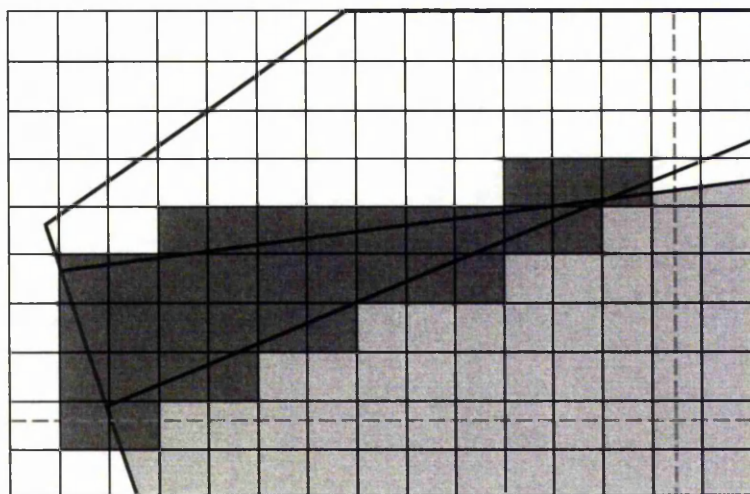


Figure 3.15: A close-up view of the up-slope region of the grid with material marked in grey and the grid cells that overlap the up-slope region highlighted in dark grey.

value of all grid elements whose centre lies within the up-slope region and take an average. Indeed this approach will approximate the integral well for a large  $\delta\theta$  and for sufficiently high grid resolution. In practice however, given a large  $\delta\theta$  and a rapidly changing concentration of incoming material, the segregation interface will be poorly defined. A more accurate approach to the integration must be obtained. Looking more closely at the region in question, figure 3.15, a typical grid element does not lie entirely within the region. The integration process begins by quickly locating the cells that overlap the region being considered. The intersection of two subsequent surfaces form two semi infinite wedges. The intersection of such a wedge and an arbitrary convex figure, in this case the drum, is necessarily convex. There are two such regions representing the in and out-flow regions, their area can be easily calculated by subdivision into triangles. Their area may be calculated by summing the area of the sub triangles; For angles where the surface does not approach a corner, these regions are triangular and thus this subdivision step may

be omitted. Each sub triangle may then have a slightly modified scan conversion method applied to it. Scan conversion is a well known method in computer graphics for finding all pixels within a given polygon and numerous implementations exist, eg. [Gla90]. A scan conversion routine outputs the horizontal extents of the figure for each vertical grid line into which it protrudes. The key advantage of a scan conversion routine of this sort is that it is “polygon centric” and not “grid centric”. This means that what determines the runtime of this algorithm is the perimeter of the triangle which varies linearly with the resolution, as opposed to the number of grid elements which varies as a square of the resolution. These scan converted regions can be trivially combined into a single scan converted region. The resulting region represents every grid element that need be considered in the subsequent integration process. Given the requirement to produce a real time or near real time simulation, this is an important feature as every grid element no longer requires consideration when carrying out the integration step. In practice this means the time taken for integration increases with the linear dimension of the grid as opposed to the total number of cells.

Having located the grid elements that overlap the up-slope region the integration process may now proceed. For each element within the test group the intersection of the cell and the up-slope region must be evaluated. Since the region may be decomposed into triangles, or indeed be triangular, and the grid cells will be rectangular the intersection of a triangle and a rectangle must be calculated. A typical approach might be a super sampling method. In this approach a set of points in each grid element is tested, themselves arranged in a sub grid, to see whether they lie within the triangle. In the limit of sub-grid resolution tending

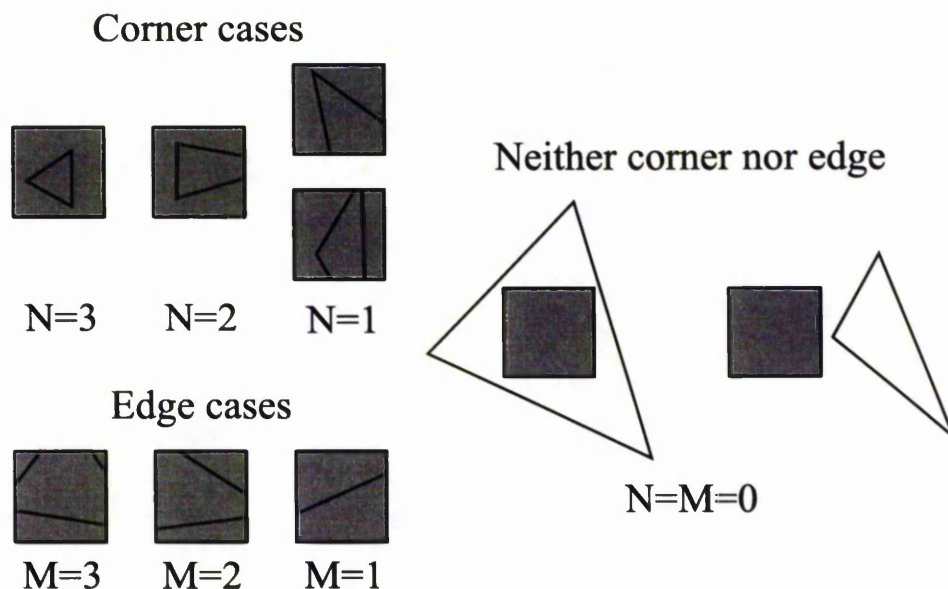


Figure 3.16: An example of every possible triangle and rectangle intersection case that the code must be able to handle.

to infinity, this provides an exact result. For the level of accuracy required a sub-grid would require in the order of, one hundred points. This requires one hundred triangle point intersection tests and should be accurate to around 1%. Even with the savings the scan conversion provides, this is still too costly a procedure to carry out for a real time simulation on any reasonable scale.

A custom algorithm is developed for the purpose of calculating this intersection area precisely. Figure 3.16 illustrates all possible intersection cases. The following pseudo code is carried out for each grid element outputted in the scan conversion pass:

```

N = Number of triangles corners within the rectangle
IF N = 3
    RETURN Area of the triangle
ELSEIF N = 2

```

```
Calculate intersection of both triangle edges that connect\
to the corner outside the rectangle
RETURN Area of region defined by two intersections, rectangle\
corners as appropriate, and triangle corners within rectangle
ELSEIF N = 1
Check for intersection of triangle edge formed with both\
triangle corners outside the rectangle and the rectangle
RETURN Area of region defined by intersected edge as\
appropriate, rectangle corners as appropriate, and triangle\
corner within rectangle
# Note N must be 0 at this point
M = Number of triangle edges within the rectangle
IF M = 3
Calculate intersection of all triangle edges and the rectangle
RETURN Area of region defined by the intersections of the\
triangle edges with the rectangle and the rectangle
ELSEIF M = 2
Calculate intersection of triangle edges within the rectangle\
and the rectangle
RETURN Area of region defined by the intersections of the\
triangle edges with the rectangle and the rectangle
ELSEIF M = 1
Calculate intersection of triangle edge and the rectangle
RETURN Area of region defined by the intersections of the\
triangle edge with the rectangle and the rectangle
```

```

# Note M must be 0 at this point
IF Any an arbitrarily chosen corner of the rectangle lies within\
    the triangle
    RETURN Area of the rectangle
ELSE
    RETURN 0

```

The returned value for grid element  $(x, z)$  shall be referred to as  $A_{x,z}$  and the area of the triangle as  $T$ . Given that  $A_{x,z} = 0$  for all grid elements not considered,

$$A_a = \frac{1}{T} \sum_{\forall x, \forall z} A_{x,z}. \quad (3.16)$$

Where  $A_a$  is the average value of  $\phi$  over the region which was considered.

### Placing outgoing material

By considering conservation of area, which is relied upon in placing the surface, the area of the outgoing region matches that of the incoming region. The form of the interface to place in the outgoing material region must be chosen. The ideal placement would be to use the function defining the surface position combined with the value of the average incoming material values to produce an accurate curve. Since the function to calculate the surface's location is non trivial this curve ends up being expensive to calculate to any reasonable accuracy. Since the average values in each region is not equal to that in the previous region this would produce a discontinuous curve. A more refined method might be to interpolate between the centre lines of each region and integrate using the appropriate averages. This would produce a continuous curve which would once again be more accurate but is still not the true value. A compromise is thus sought. Due to the choice of small angular

increment, a straight interface may be chosen. It is possible to pick a similar value to the one the might be chosen averaging method mentioned. This method is reasonably accurate but it requires that the outgoing material lag behind the incoming material by half an angular increment. For many purposes this may prove an acceptable compromise, while having a synchronised incoming and outgoing material without requiring an examination of the material that has not yet been transported. An arbitrary straight line is thus chosen for the interface. The interface will be placed perpendicular to the angular bisector of the region under consideration. Clearly this will yield discontinuities in the segregation interface as with the initially suggested method. In a scheme of this sort this is unavoidable.

The angular increment is chosen so the distance between the projections of the two surface drum intersection points onto the plane in which the interface will lie is exactly one grid element wide, this distance is illustrated in figure 3.17. The plane onto which the projection is taking place and the intersection points also depend on the choice of angular increment. This method thus requires iterative re-evaluation of the surface position as the choice of angular increment is refined. A scheme of this sort would produce an optimal choice of angular increment. This optimal choice has the cost of an iterative evaluation until some suitable error margin around the true value is reached. This means that the runtime required for evaluation may vary considerably depending on the local topology of the drum. To avoid this unpredictability an angular increment that represents the minimum required for the drum is chosen. This is simply an angular increment that produces a one grid element distance at the furthest point from the drum's centre, illustrated in figure 3.18. Having chosen an angle to allow no more than a single grid element be spanned by the outgoing material region, irrespective of the choice of interface

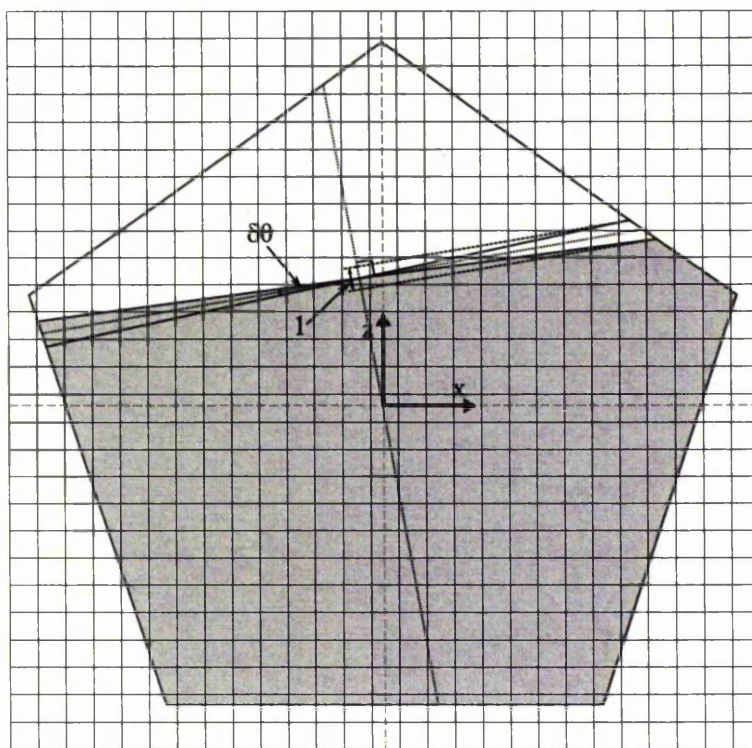


Figure 3.17: An illustration of the method by which an angular increment is chosen.

angle the maximum error will be on the order of half a grid element.

To place the material in the outgoing region a scan conversion routine is used, similar to that which was used in the incoming region. The difference in this case is that only grid elements whose centre lies within the outgoing region are considered. This approach ensures that each grid element will only have material mapped onto it once per revolution. In the worst case this will introduce a half cell error at a concentration jump, but there is no skew in this one way or other so on average the total material concentration is fixed. Similarly this worst case is only at concentration jumps, i.e. the segregation interface, which, in any usefully high resolution, is a very small proportion of the total quantity of material. The same method as was used to locate the surface within the drum as a whole at a given



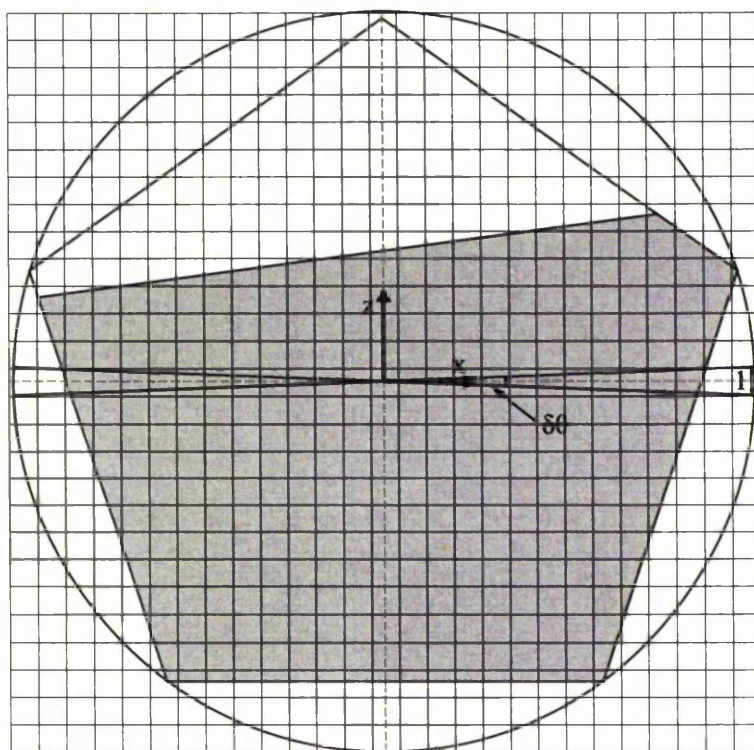


Figure 3.18: The method by which a minimal angular increment is chosen.

angle is then applied as shown in figure 3.19. Given that the outgoing material is being mapped onto discrete grid elements and the interface cannot be guaranteed to lie on a grid line; the outgoing material must be appropriately placed. Clearly the material towards the centre of the surface must be small particles,  $\phi = 1$ , and the outer material is pure large particles,  $\phi = 0$ . For cells well away from the segregation interface the value to be placed is clear. For cells which intersect the segregation interface however cannot simply be filled with a value of 1 or 0 based on their position because for a cell whose centre lies almost exactly on the interface, would cause half a cells material to be placed on the incorrect side of the interface. To deal with this problem a form of anti-aliasing is introduced. The signed perpendicular distance, in grid units, of the centre of each cell from the



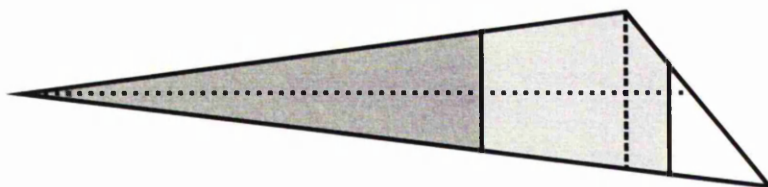


Figure 3.19: The method for locating the segregation interface within the outgoing material region. Two example surfaces are illustrated with solid lines and the small particles are highlighted in light grey.

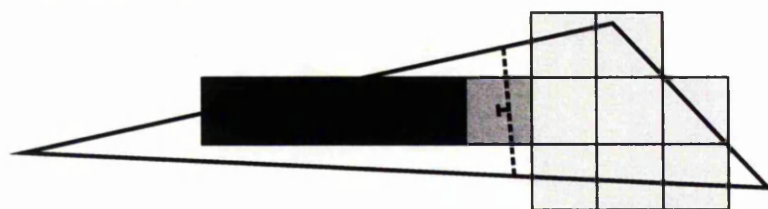


Figure 3.20: Grid element values in a typical outgoing material region. Only grid elements with highlighted grid lines were mapped to in this pass. Shaded as in figure 2.3.

interface  $d$  is used. The cells are then filled using

$$\phi_{x,z} = \begin{cases} 0 & d < -0.5 \\ 0.5 - d & -0.5 < d < 0.5 \\ 1 & 0.5 < d \end{cases} \quad (3.17)$$

The grid element values are illustrated in figure 3.20. The angular increment illustrated is larger than that used in the simulation to improve clarity and demonstrate how the  $\phi_{x,z}$  are chosen. Note that there are grid elements that would be overlapping this outgoing region that are not shaded in this pass. These cells will either have been filled in a previous step or will be filled in a future one.

One final factor needs to be dealt with before an easily interpretable image of the field  $\phi$  can be generated. Material that has been mapped is not deleted, it is simply overwritten when the drum has rotated sufficiently. This is not an issue per se but when visualising the results it can make the image harder to interpret.

To this end a second grid is maintained, matching the previous one, but with each element simply being flagged as visible or not. This is considered a masking layer to be used in conjunction with the  $\phi$  field. Typically a scan conversion routine on the polygon as a whole is used to initially mask all areas of the field that lie outside the drum from view. The first surface position is calculated, before the iteration begins, using the same routine to mask the area above the surface as invisible. At every subsequent step the scan conversion method applied to the outgoing material region is also applied to the incoming material region. The outgoing material region is then marked as visible and the incoming region as invisible. The nature of the routines combined with the initial configuration ensures that, when combined with the masking, only material that would be visible in practice is visible when visualised.

As a post processing step the image may be rotated to give the impression of viewing it from a fixed frame of reference. The result of this numerical scheme is a simulation that runs in real time or faster, with real time visualisation, at an acceptable resolution on a moderately specified desktop computer. For some purposes a reduced resolution may be usable which yields a considerable performance boost, as the costs are essentially proportional to the number of grid elements. In this case the simulation may be run for thousands of revolutions in a matter of minutes. The full source code is provided in appendix C.

### 3.3.4 Experimental method

A typical drum used in the experiment is pictured in figure 3.21. The upper and lower plates are made from seven millimetre thick perspex. The spacer is made from three millimetre thick aluminium. The three layers are attached via screw

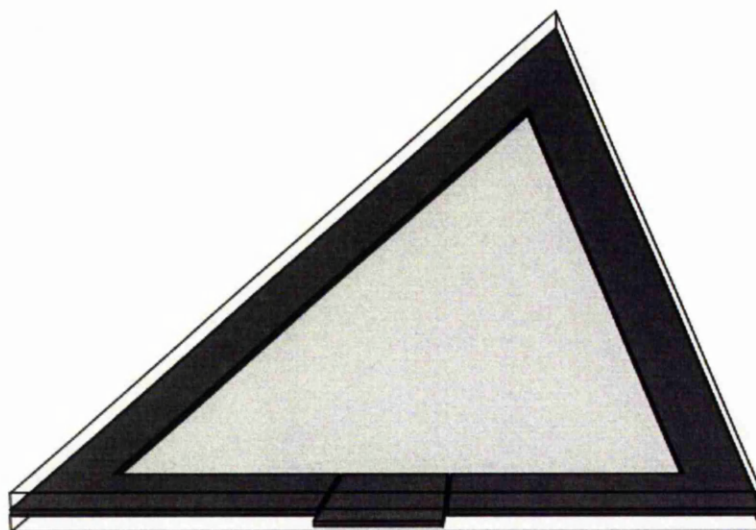


Figure 3.21: An illustration of an equilateral triangular drum rendered in perspective.

threads that run through the upper perspex, spacer and half way through the lower perspex. The entire construction is tightly sealed by the bolts except for a portion of the spacer that protrudes from between the perspex layers on one side that may be pulled out to fill the drum. This piece of spacer acts as a “stopper” that simply fits tightly between the perspex layers to avoid material leakage. The drum has eight screw threads passing half way through the back perspex arranged in an octagon around the barycentre; These allow the drum to be securely affixed to a device to rotate the drum.

The rotation mechanism is a planetary geared electric motor connected to a pulse width modulation speed control unit. The motor is mounted on an axle to which, via a damped universal joint to minimise vibration, a plate is fitted with holes matching the octagonal arrangement on the rear of the drum. The drum is mounted in a vertical plane by bolting to the plate.

The rotation mechanism produces rotation rates between zero and ten revolutions per minute. The rotation may be in the clockwise or anticlockwise direction although for symmetric drums this has no impact on the result up to reflection. For an asymmetric drum however, the rotation sense must be preserved. Clockwise rotation for symmetric drum experiments is arbitrarily chosen. The mapping method assumes that the experiment is in a continuous avalanching regime; that is to say that the particles reaching some interval about the surface will flow immediately, and particles are not so energetic as to take flight when reaching the surface. This energetic flight mode can however be used to mix the particles. The optimal rotation rate for different material choices may vary somewhat but the typical rotation rates are on the order of one revolution per minute.

The material itself consists of small glass beads in two size ranges, 150 – 300 micron and 600 – 750 micron. Both being made from the same material ensures they have the same bulk density. The particles are also sufficiently similar in size that the percolation mentioned in section 2.1 does not occur.

Due to the bidisperse nature of the material, when mixed the volume is less than the sum of the volumes of the constituent parts. In order to control the total volume a specific method must be employed. When the mixture is made entirely of large or small particles, the volume of the mixture equals exactly the volume of the constituents. When the mixture contains half each of large and small particles, the total volume is some other smaller value. For any given ratio a parameter,  $\nu$ , which will represent the ratio of the total volume to the sum of the constituents, must be found. The value of  $\nu$  at the half and half ratio will be known as  $N$ , ie.

$N = 1$  represents no loss in volume. There are thus three known values for  $\nu(\phi)$

$$\begin{aligned} \text{a) } \phi &= 0 \Rightarrow \nu = 1 \\ \text{b) } \phi &= \frac{1}{2} \Rightarrow \nu = N . \\ \text{c) } \phi &= 1 \Rightarrow \nu = 1 \end{aligned} \tag{3.18}$$

Knowing three points in the functional relationship, the simplest single polynomial that can fit is a quadratic. To this end a relationship of the form

$$\nu(\phi) = A\phi^2 + B\phi + C \tag{3.19}$$

is sought. Substituting in equation 3.18 a yields

$$C = 1. \tag{3.20}$$

Substituting in equation 3.18 c yields

$$B = -A. \tag{3.21}$$

Lastly, substituting in equation 3.18 b yields

$$A = 4(1 - N). \tag{3.22}$$

Substituting these back into equation 3.19 obtains

$$\nu = 4(1 - N)\phi^2 - 4(1 - N)\phi + 1. \tag{3.23}$$

This means that to obtain a total volume of  $V$  the sum of the constituent volumes must equal

$$V_t = \frac{V}{4(1 - N)\phi^2 - 4(1 - N)\phi + 1}. \tag{3.24}$$

Meaning that the volumes required of large and small particles are

$$V_t(1 - \phi) \text{ \& } V_t \phi \tag{3.25}$$

respectively. In the case of the material used, the value of  $N = 0.85$  meaning that the coefficients of the  $\phi$  terms in equation 3.24 are  $\frac{3}{5}$ . In this way the mixing ratio and the volume may be controlled independently.

The only other experimental consideration in an experimental set-up like this is how to set-up the initial material distribution. Since the particles size segregate, it can be difficult to ensure the mixture is initially homogeneous. One solution is to rotate the drum fast enough to enter the “cascading regime” whereby the particles take flight at the surface, then slow rapidly resulting in a chaotic mixing. Even in this case the mixture is not perfectly homogeneous but as we shall see this is not a major issue. Another more subtle trick is to force the material to segregate in the plane perpendicular to the plane of the drum. Specifically to cause the large particles to come to the side of the drum that will be observed and the small particles to the side which affixes to the apparatus. This is simply done by shaking the drum in a horizontal plane. Since we are concerned with the bulk composition of the material in the avalanche this has no impact on  $\phi$ . When the number of small particles is very small the first method works better, but for more even mixtures this second method works well. The second method has an additional advantage in that the core of revolution will appear as mostly large particles which will contrast better with the surrounding, predominantly, small particles.

In order to help make the experimental set-up and frame of reference clearer a series of images of the experimental set-up for a typical experiment will be illustrated. Figure 3.22 illustrates a set of unrotated and unprocessed images in the laboratory’s frame of reference.

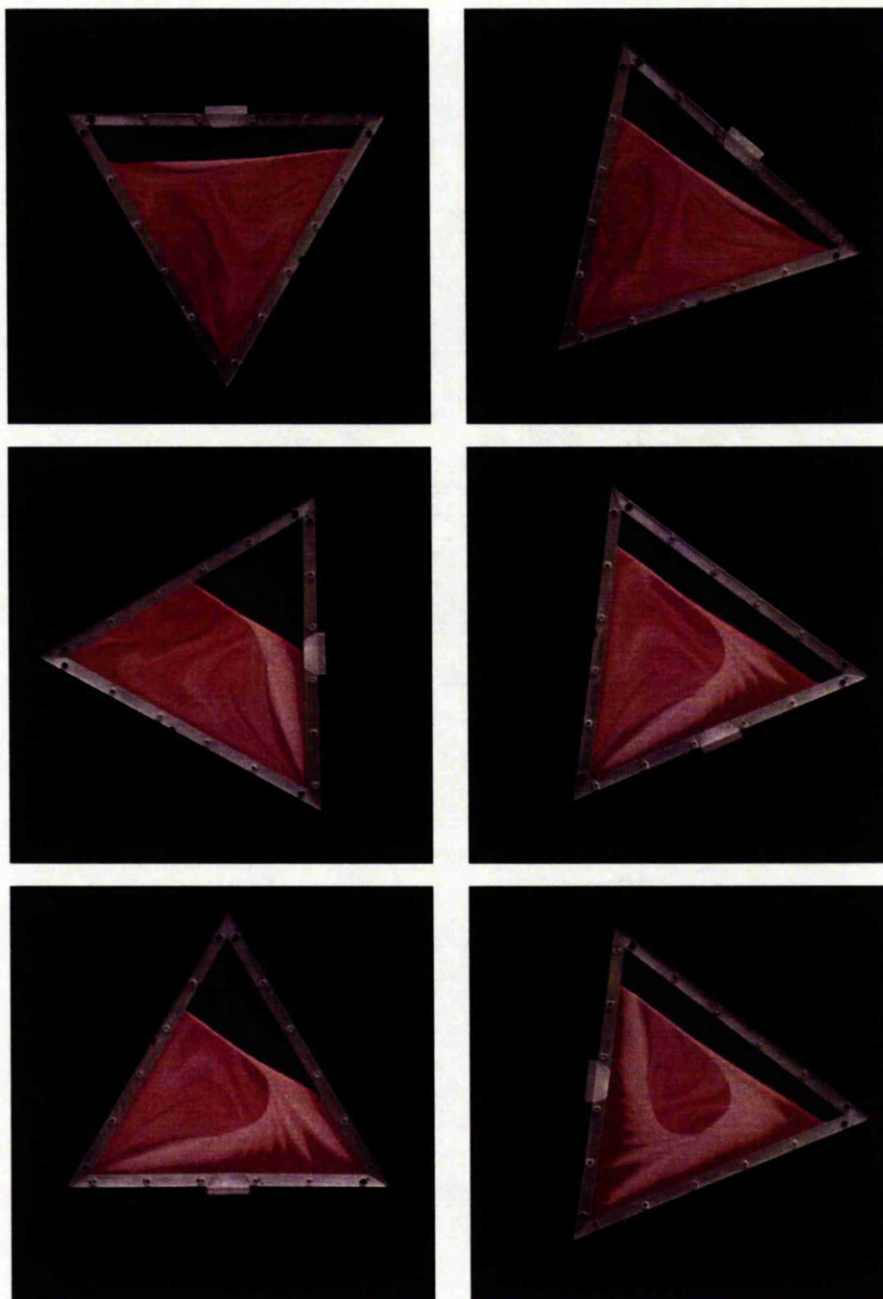


Figure 3.22: A series of photos of the triangular drum in the laboratory's frame of reference of the development of the segregation interface.

The effectiveness of the volumetric method will now be illustrated by comparison to experiment. The generality of the approach allows simulations to be carried out on arbitrary convex drums. Attention will initially be restricted to simple, regular geometric figures.



## Chapter 4

# Comparison of results in a triangular drum

The triangle represents the simplest regular polygon. Due to the minimal nature of the figure one may more easily relate the resulting patterns in the segregation interfaces to the geometric properties of the figure. As was shown in section 1.2, however, even without considering segregation non-trivial patterns may be produced.

### 4.1 Initial comparison to experiment

To illustrate the agreement between experiment and simulation a fill level must be chosen, which is considered representative and illustrative. At this point analysis will be restricted to fill levels above 50%. The reason behind this will be explained in more detail in section 4.5.1. The 75% fill level will thus be analysed initially. In practice there is some degree of experimental error in the exact fill level and particle ratio chosen. Note that despite the simulation parameters representing the

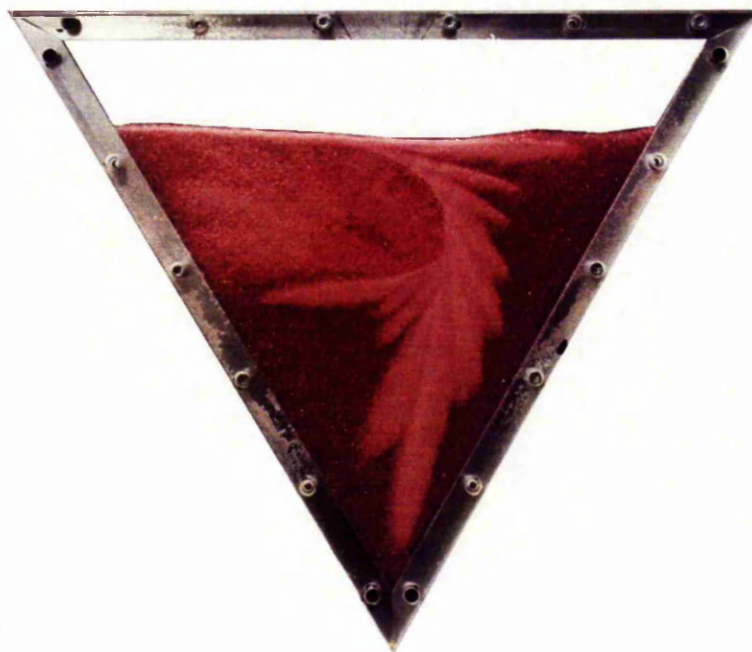


Figure 4.1: A photograph of a triangular drum with a 75% fill and 45% particle ratio at  $\theta = \pi$ .

required parameters, as opposed to being matched with the true values present in the experiment, agreement is good.

The early development of the segregation interface will be compared first. Figure 4.1 illustrates the experiment. Note that  $\theta = 0$  is the angle at which the material has reached its angle of repose and begun to flow. One can see that as soon as this occurred the segregation began to take place, yielding a sharp segregation interface. Comparing this image to the simulation results shown in figure 4.2 the experimental interface is a lot less clean. This early time discrepancy can be put down to, in part, the initial distribution of material in the drum. Since the material size-segregates, it is difficult to make sure that the drum contains a homogeneous mixture of large and small particles. The simulation, in contrast, may contain any distribution of material prescribed, specifically a perfectly mixed

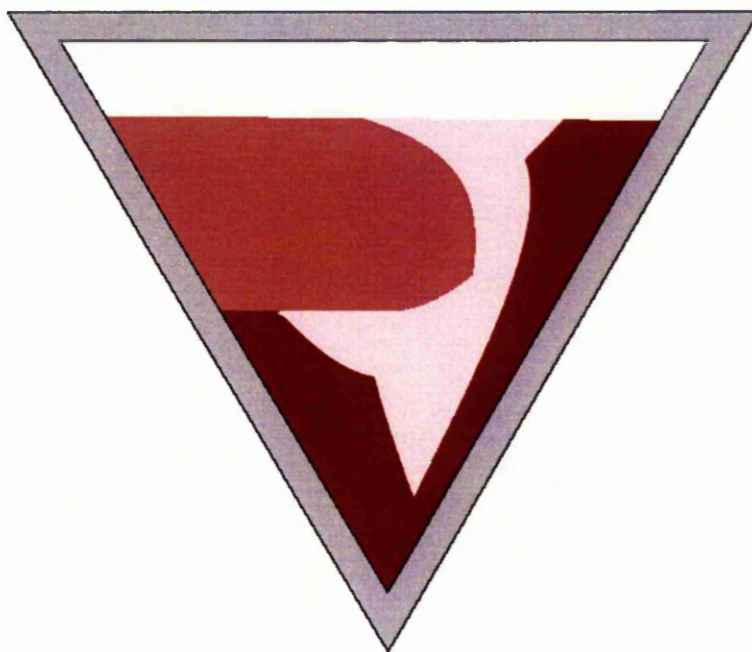


Figure 4.2: A simulation of a triangular drum with a 75% fill and 45% particle ratio at  $\theta = \pi$ .

state. Despite this one can see a good qualitative agreement.

Figure 4.3 shows the same comparison carried out at  $\theta = 3\pi$ . One can see that the material that passed through the interface on the first revolution has now passed through a second time. This second segregation stage has caused the variations in the segregation interface to be reduced. The core of revolution at the centre is already clearly defined. When compared to the equivalent simulation in figure 4.4, the division between the once and twice segregated material appears less well defined. The core of revolution in the simulation also appears somewhat smaller. This comes down to one of the approximations chosen in setting up the mapping method. The core of revolution represents material that has never passed through the avalanching region. In this simulation the avalanching region is completely flat and has no thickness. The interface observed in the experiment,

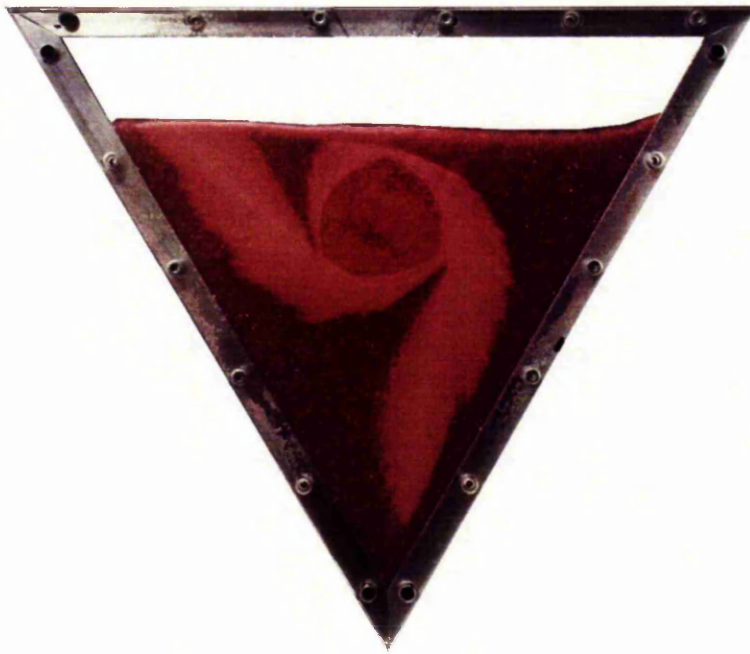


Figure 4.3: A photograph of a triangular drum with a 75% fill and 45% particle ratio at  $\theta = 3\pi$ .

however, has a slight 'S' shape and a small thickness. This causes the avalanche to transport material from somewhat below the apparent surface and the core of revolution is eroded when compared to the simulation. One can still however see a strong qualitative agreement between the simulation and experiment even at this early stage.

Due to the initial discrepancies in the distribution of the material in the experiment and the stable long term behaviour of the segregation interfaces formed, the long term behaviour will now be examined. Figure 4.5 shows the experiment after ten revolutions have been completed. Once again the interface itself is somewhat smoother and the rotational symmetry of the drum is strongly reflected in the pattern. The core of revolution is still clearly visible and well defined. When compared to the simulation the overall form of the segregation interface agrees well.



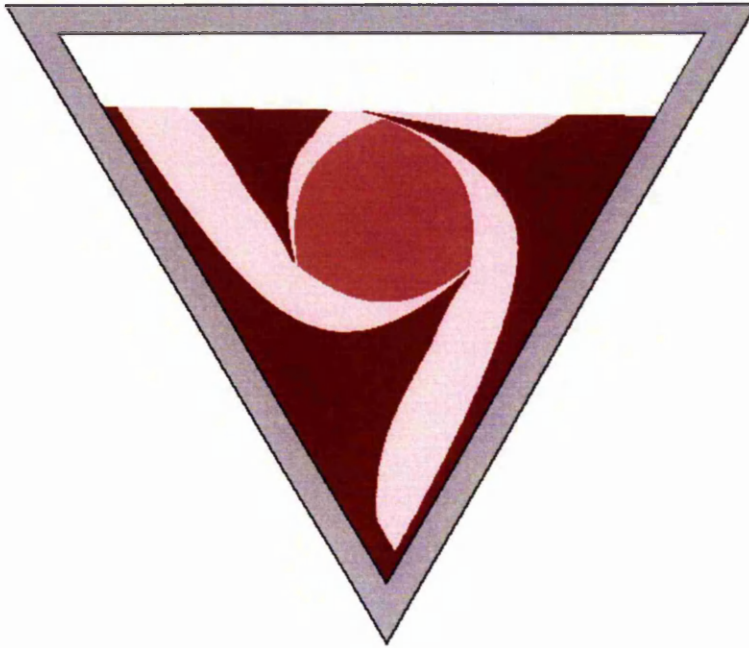


Figure 4.4: A simulation of a triangular drum with a 75% fill and 45% particle ratio at  $\theta = 3\pi$ .

As before the core of revolution is too small in the simulation. The variations in the interface are still visible suggesting they may not stem from initial inhomogeneities. In fact at slower revolution rates the avalanching becomes somewhat intermittent. Even at higher rotation rates there is a degree of intermittency in the avalanche. Apart from this intermittency and the enlarged core of revolution the simulation shown in figure 4.6, agrees well with the experiment.

Having compared the results, the early time behaviour cannot be predicted due to initial inhomogeneities in the mixture, though the agreement is still good. The bulk of the analysis will therefore be restricted to the long term behaviour of the system. Having observed the system for some time in numerous cases, it is concluded that the segregation interface appears to have approximately taken its final form within three revolutions of the drum. The state after ten revolutions is

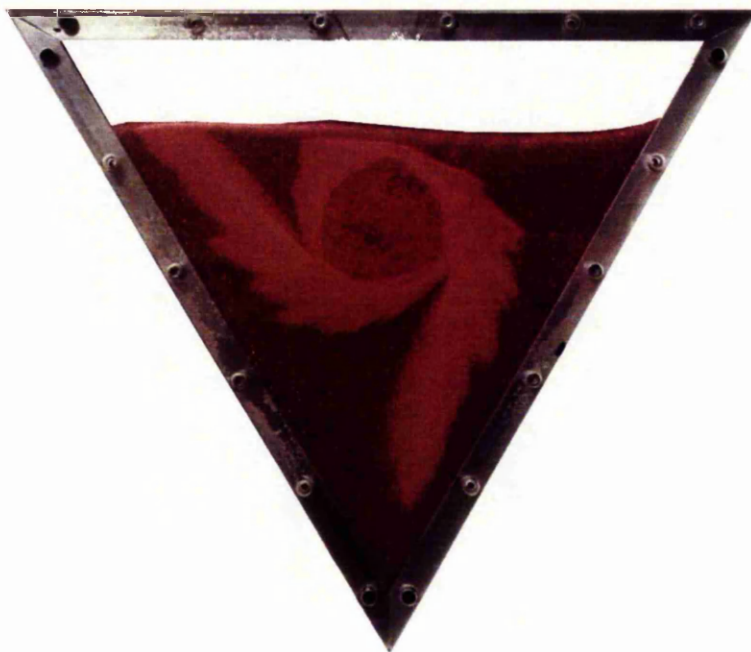


Figure 4.5: A photograph of a triangular drum with a 75% fill and 45% particle ratio at  $\theta = 19\pi$ .

thus considered to reflect the long time state of the system.

## 4.2 Varying ratios

Having determined that the simulation agrees sufficiently well with the experiment, a more detailed analysis of the dynamics of the system is undertaken. For a fixed fill level the effect of varying the average partial volume of small particles,  $\Phi \in (0, 1)$ , will be investigated. Images of the simulation after ten revolutions shall be compared to the experiment. Figure 4.7 contains images of the simulation at six different particle ratios; the corresponding experimental images are shown in figure 4.8.

Two interesting changes in structure are noted throughout this progression.

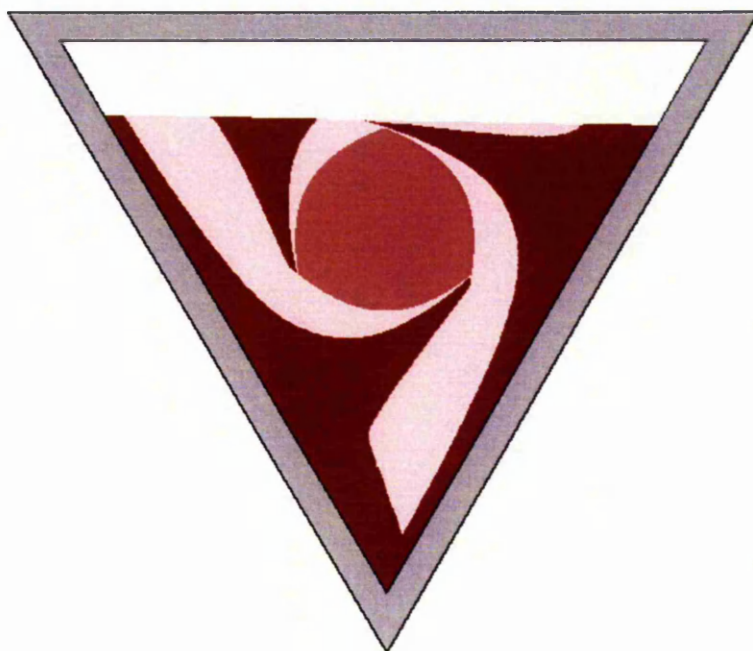


Figure 4.6: A simulation of a triangular drum with a 75% fill and 45% particle ratio at  $\theta = 19\pi$ .

Between 35% and 50% the lobes gain an extra corner causing their tip to appear square rather than sharply pointed. At around 90% the lobes appear to have retained their flat tip, but the sharp corner that appeared previously seems to be becoming curved once again. Despite this qualitative change the structure remains globally similar and varies continuously. There are three single lobes extending into the corners of the drum.

### 4.3 Varying fill levels

Given the variation in structure present when the ratio of large to small particles is varied,  $\Phi$  is similarly fixed and the fill fraction  $A$  is varied. The results are shown in figure 4.9. Similarly, figure 4.10 shows comparable images of the experiment.



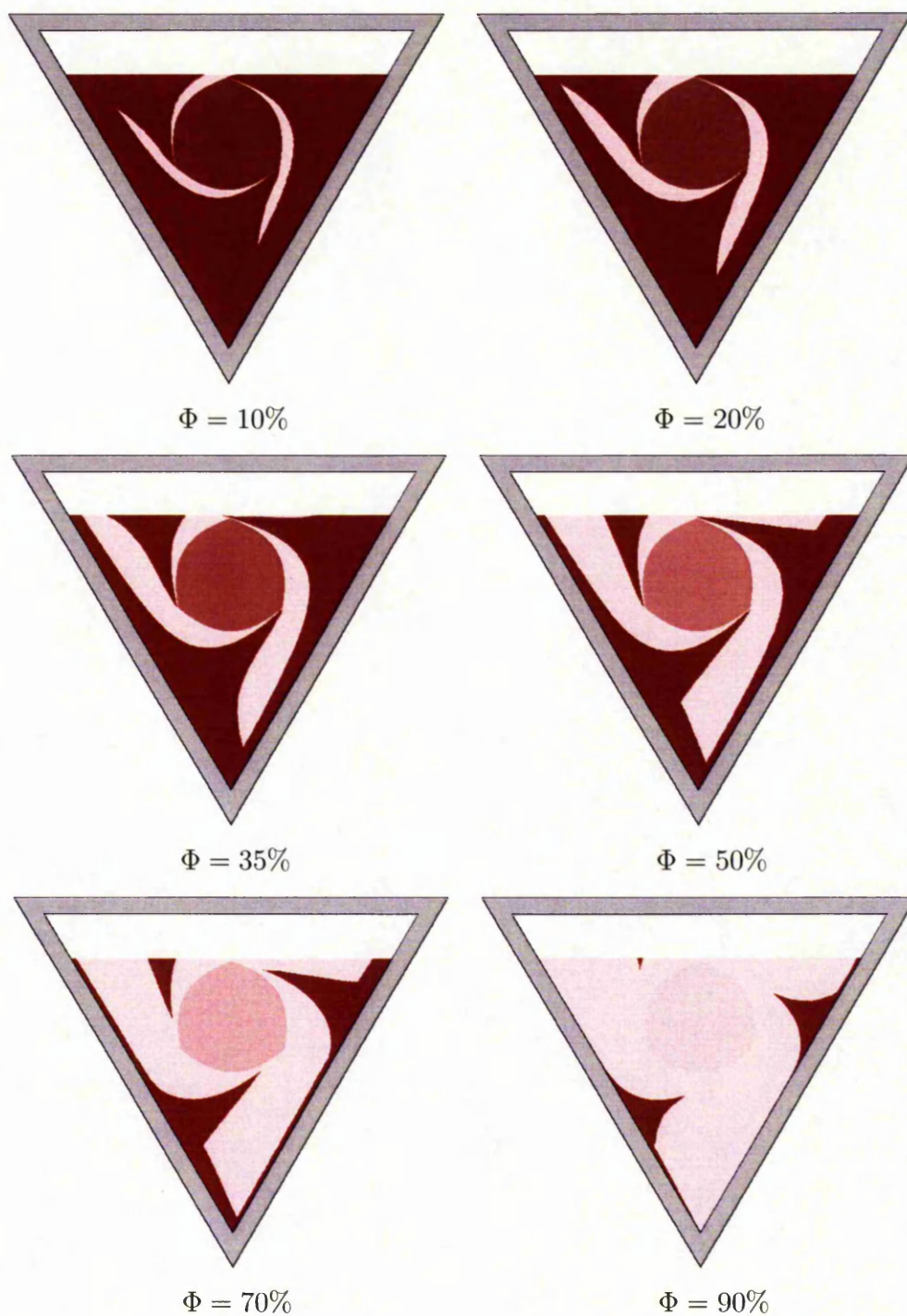


Figure 4.7: Simulations of the triangular drum after ten revolutions with various initial particle ratios at a fixed fill level of 75%.



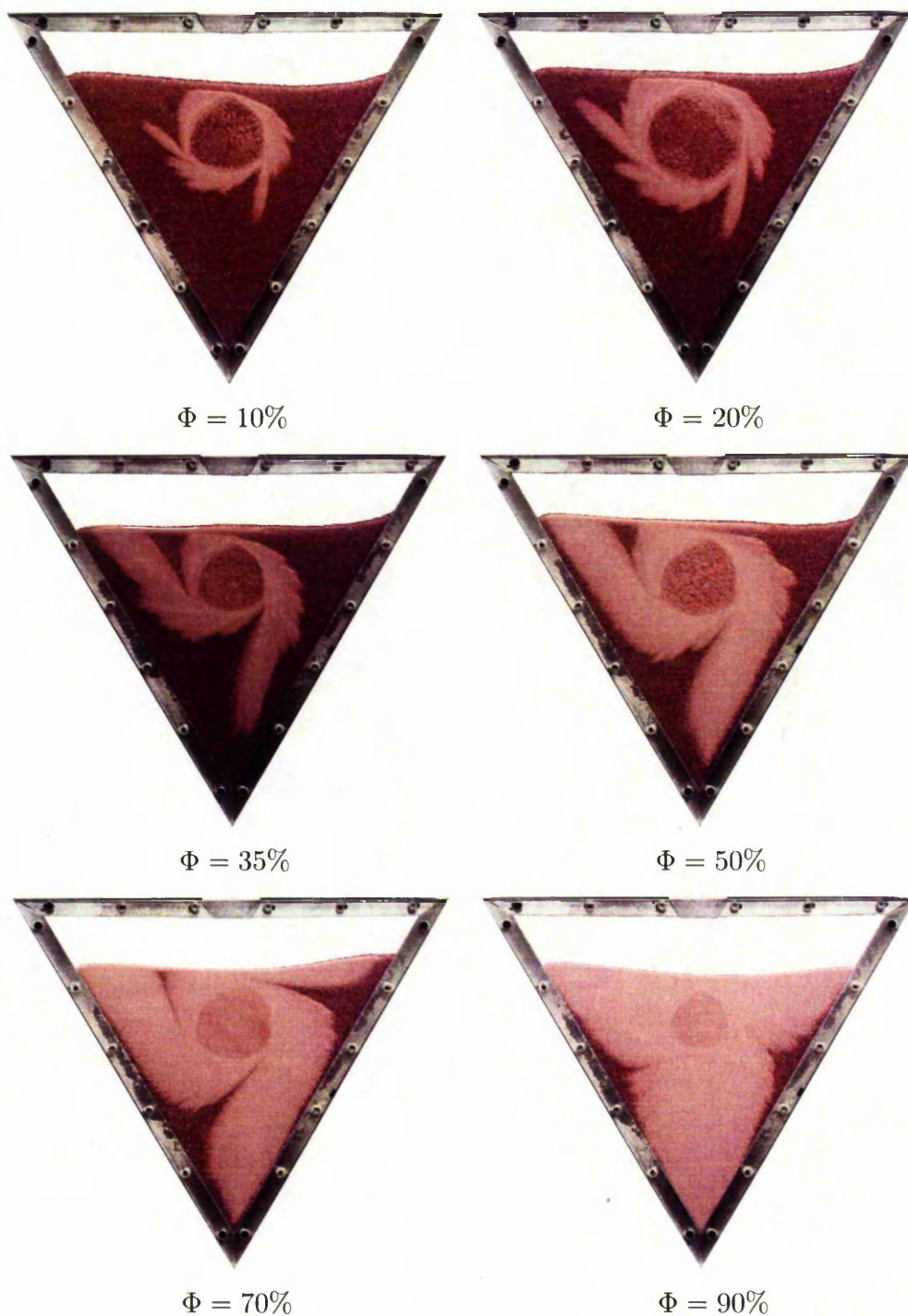


Figure 4.8: Experimental images of the triangular drum after ten revolutions with various initial particle ratios at a fixed fill level of 75%.

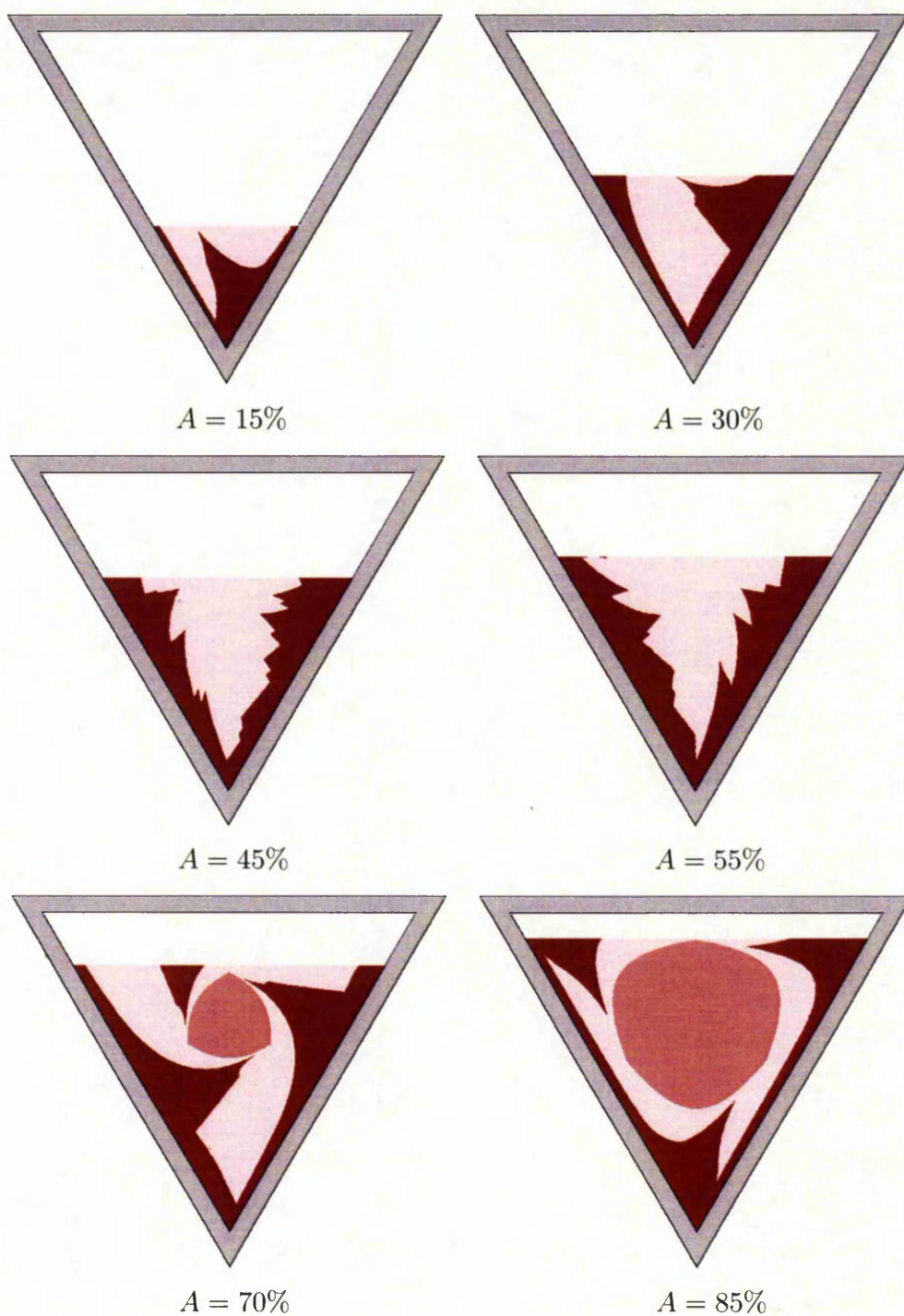


Figure 4.9: Simulations of the triangular drum after ten revolutions with a fixed initial particle ratio of 50% at various fill levels.



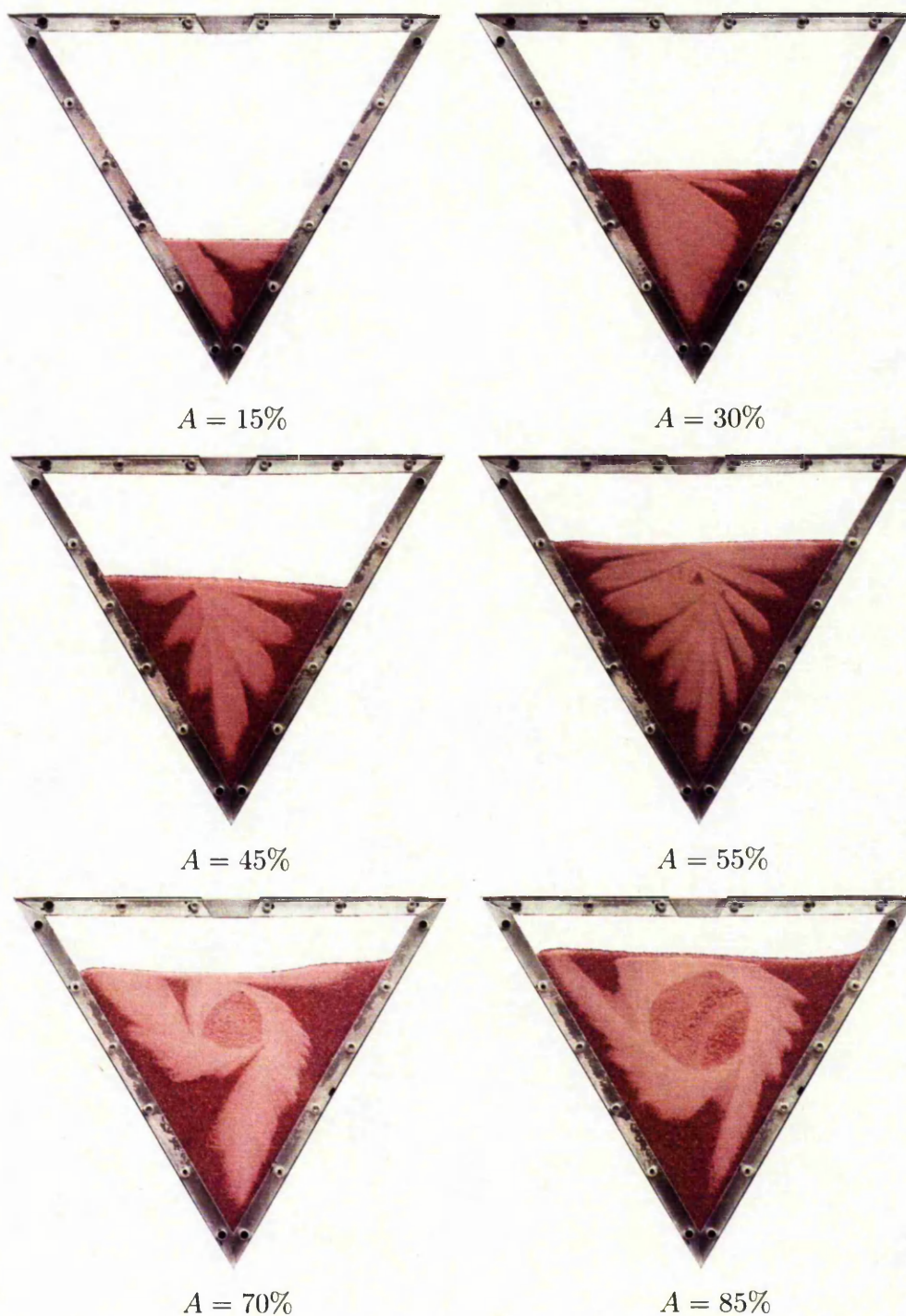


Figure 4.10: Experimental images of the triangular drum after ten revolutions with a fixed initial particle ratio of 50% at various fill levels.

Compared to the particle ratio variation the structure changes within this progression appear far richer. Between 15% and 30% there is a similar change from a pointed tip to a squared tip. At the 30% fill level, and comparably the 70% secondary lobes are beginning to form.

As with the particle ratio variation, the structure for low and high fill levels remains constant. Drawing attention to the 45% and 55% figures there is a marked change in behaviour. There appear to be numerous unstructured lobes superimposed onto a roughly triangular base shape. This behaviour may be indicative of some dynamically different behaviour at these fill levels near 50%.

## 4.4 Long time behaviour

Examining the simulation's long term behaviour, and that of the experiment, it appears that they settle to a steady state. To confirm this supposition a measure of how much the segregation varies from one revolution to the next is introduced. First introduce a new parameter  $T(\theta)$  as defined in equation 3.15. It is simply a normalised measure of the distance from the centre of the surface to the segregation interface at any given angle. The goal is to compare the value of this function over two subsequent revolutions. In this case, a running average will be used,

$$C'(\theta) = \int_{\theta-2\pi}^{\theta} |T(t) - T(t + 2\pi)| dt \quad (4.1)$$

Since the maximum change in  $T$  between any two points is 1 and the absolute value enforces  $C' > 0$  and  $C' \in [0, 2\pi]$ . The value is thus normalised,

$$C(\theta) = \frac{1}{2\pi} \int_{\theta-2\pi}^{\theta} |T(t) - T(t + 2\pi)| dt \quad (4.2)$$

Having determined such a measure an implementation for use on these simulation results must be constructed. Values at discrete  $\theta$  may be obtained but one

must assume a functional relationship to interpolate between these values. Equally the values of  $\theta$  for which  $T$  is obtained, may vary with subsequent revolutions due to the means by which an optimal  $\delta\theta$  was chosen. This value could be decreased to the next integer division of  $2\pi$  thereby ensuring that precisely corresponding values for  $T$  are obtained, i.e. if  $T(a)$  was obtained, so would  $T(a + 2\pi)$ . Decreasing the time step would, however, proportionally increase the run time. To avoid this situation a more generic approach is developed.

$T$  has a non-linear form and there is no way to determine which interpolant might yield the best results. Functions of the type we are dealing with may be arbitrarily well approximated by a polynomial. This means that for sufficiently dense sampling even a piecewise constant function can be sufficient. Due to the relatively high sampling density in the function  $T(\theta)$  and the desire for accuracy a higher order interpolation is chosen.  $T(\theta)$  is thus considered to be the piecewise linear function formed by connecting the discrete points for which we know its value. Figure 4.11 shows  $C$  for a range of fill levels. The lowest curve is produced at the 50% fill level. One can see that this curve is simply oscillating slightly around a small finite value for the entire length of the curve. This reflects the observed behaviour in that there appear to be extremely small perturbations proceeding around the segregation interface, hence the low frequency constant oscillation. This matches the expected behaviour and helps confirm that the measure used is appropriate. The remaining curves appear to exhibit two other behaviours. All remaining curves appear to start at some larger value and fall to a smaller fixed value about which they oscillate. The curves themselves appear to fall into two groups, those whose limit appears to fall to nearly 0.06 and those whose limit falls to about 0.02.

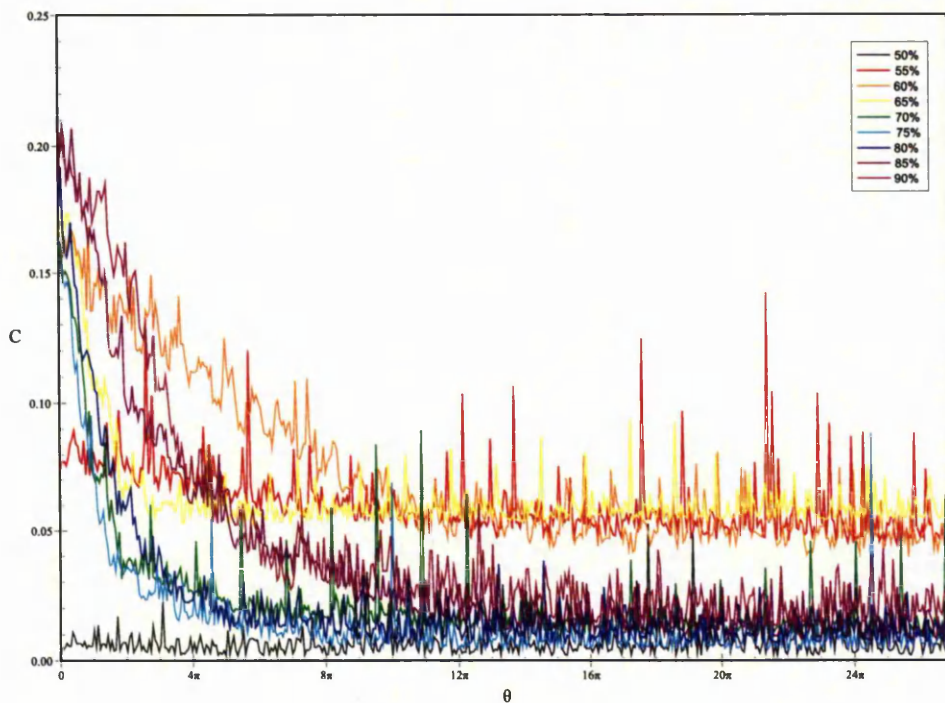


Figure 4.11: A graph of the difference in interface between subsequent revolutions,  $C$ , covering approximately 15 revolutions of the drum.

The apparent limiting values will be investigated. The average value in some tail region of specified length chosen so as all curves appear to have reached their limiting value is used. Given the apparent change in behaviour between 65% and 70%, data from several new points will be added to try and determine more thoroughly the nature of the functional relationship. Given the special nature of the 50% fill it will be omitted. These limiting values are plotted against the respective fill levels in figure 4.12. There is a smooth but rapid change in the 65% to 70% interval. To gain some further understanding of what this means the graph's meaning needs reconsidering. If the difference between subsequent revolutions is staying around some fixed value, and visual inspection shows that the figure is not changing, the values plotted in figure 4.12 represent the size of the



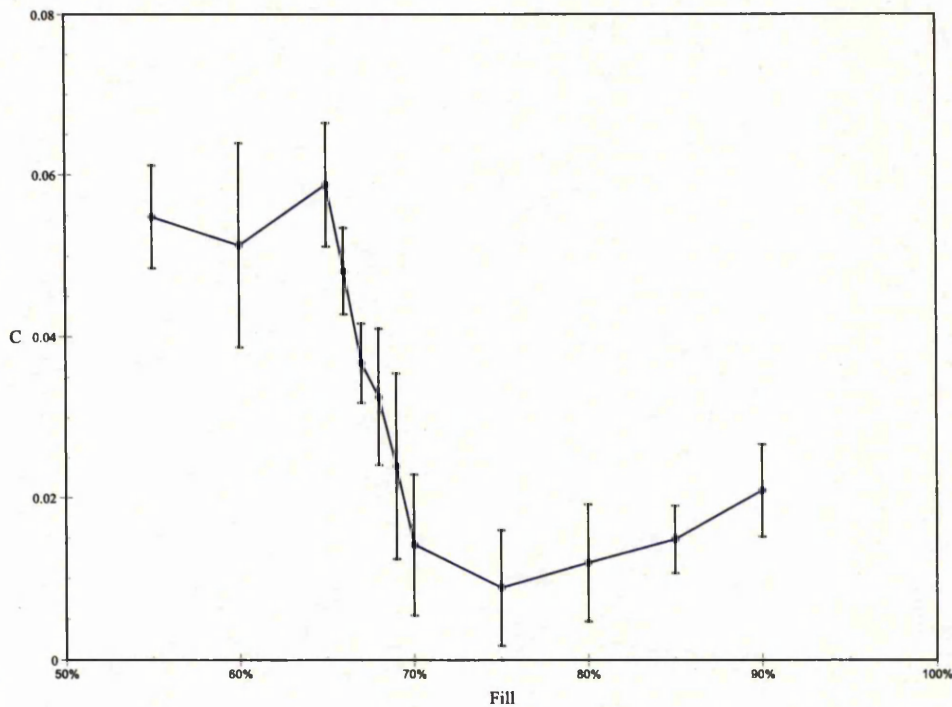


Figure 4.12: A graph of the approximate final value of  $C$  for varying fill levels. The error bars represent the standard deviation over the averaged interval.

oscillation around a fixed state that the segregation interface is undergoing. The two, roughly, constant regions on the graph, below 65% and above 75%, reflect two different magnitudes of orbit. The higher values orbit more tightly. This agrees with what is observed, in that nearer 50% there appear to be small features which proceed around the interface between revolutions whereas these are not visible at higher fills. This change of behaviour therefore takes place over a narrow parameter range which divides out behaviour into three distinct regions:

1. 50% to 65% is a region which settles to a wide orbit around some state.
2. 65% to 70% is a region in which the orbit width drops linearly.
3. 70% to 100% is a region which settles to a narrow orbit around some state.

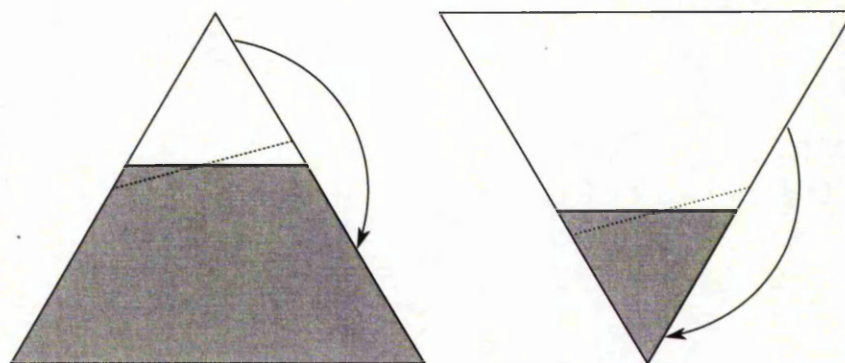


Figure 4.13: A triangular drum shown with a high and matching low fill level with the direction of rotation marked with an arrow and the surface at the next time step marked with a dotted line.

## 4.5 Special considerations

There are several important points and issues to note regarding this system. Special cases and behaviours may be observable in certain regimes and noise may have an impact on the system.

### 4.5.1 Symmetry of the flow

In order to investigate parallels between behaviour at high and low fill levels, for example 40% compared to 60%, two such cases will initially be examined in figure 4.13. Given the direction of rotation, the surface orientation at the next time step was overlaid. If one rotates the two drums so as the material region of one coincides with the non material region of the other however, one can see that the direction of material transport in the two cases oppose one another. In order to match the material transport consider reversing the sense of rotation of the drum with the lower fill level. To allow for more easy comparison, the current orientation of the low fill drum will match the 'next time step' orientation of the high fill drum. The



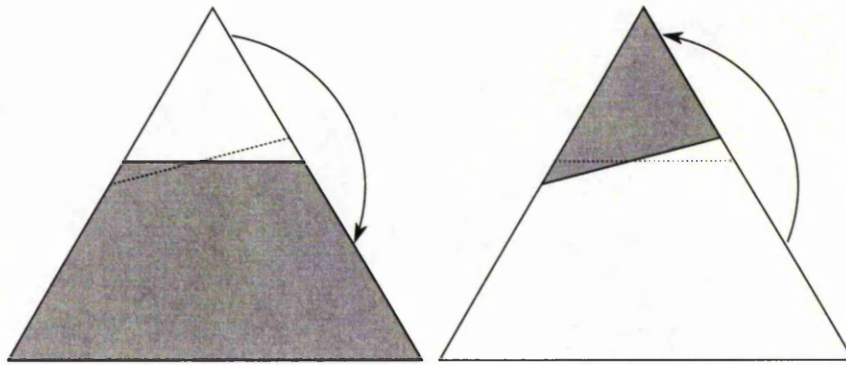


Figure 4.14: A triangular drum shown with a high and matching low fill level with the direction of rotations marked with an arrow and the surface at the next time step marked with a dotted line. The low fill level image has been inverted for ease of comparison.

low fill drum will also be shown inverted to allow more easy comparison to be drawn. Figure 4.14 shows the resulting diagrams.

As can be seen, the domain and co-domain of the mapping in the high and low fill drums match. Since the fill and angular increment were chosen arbitrarily the same can be said for any comparable mapping regions in the drum. The matching fill levels imply the surfaces will coincide for any given angle and this is all that is required for the mapping regions to match except that the low fill is rotating in the opposite sense to the high fill. Reversing the sense of rotation corresponds to a reflection of the segregation interface as the left to right transport is simply replaced with it's right to left counterpart. Hence at equivalent high and low fill levels of the drum, the resulting patterns produced by segregating or non segregating methods will be equivalent under reflection.

For this reason fill levels greater than or equal to or less than or equal to 50% are all that need consideration. In order for the core of revolution to be visible in the experiments fill levels greater than or equal to 50% are required, hence their use in this and subsequent chapters.

### 4.5.2 The effect of noise

The initial inhomogeneity in the experimental system becomes decreasingly apparent as the drum proceeds through several revolutions. A comparison of the simulation with two different initial states is produced. Two states have had noise added to the initial  $\phi$  field. The noise added is uniformly distribution in  $[-0.15, 0.15]$ . Figure 4.15 illustrates both noisy images and the difference between  $\phi$  in the two results, with the contrast increased for ease of viewing.

The primary differences between the figures are limited to the core, in which the differences between distributions in undisturbed material remain clear, and the very edge of the segregation interface. In practice, though the noise is uniformly distributed, the average ratio throughout the drum will almost certainly be perturbed slightly. Since a small change in ratio results in a small change in the resulting segregation interface, this change will yield a small perturbation of the segregation interface. At no point does this margin within which the error lies span more than two grid cells. Since the interface is not grid cell aligned, this suggests an error of order of one grid cell. The segregation interface is therefore robust with respect to perturbation of the initial conditions. This same behaviour can be observed for all particle ratios at the majority of fill levels. Figure 4.16 illustrates the same type of difference image with various parameters. The fill levels for which this perturbation may have a greater impact will be discussed in greater detail further ahead.

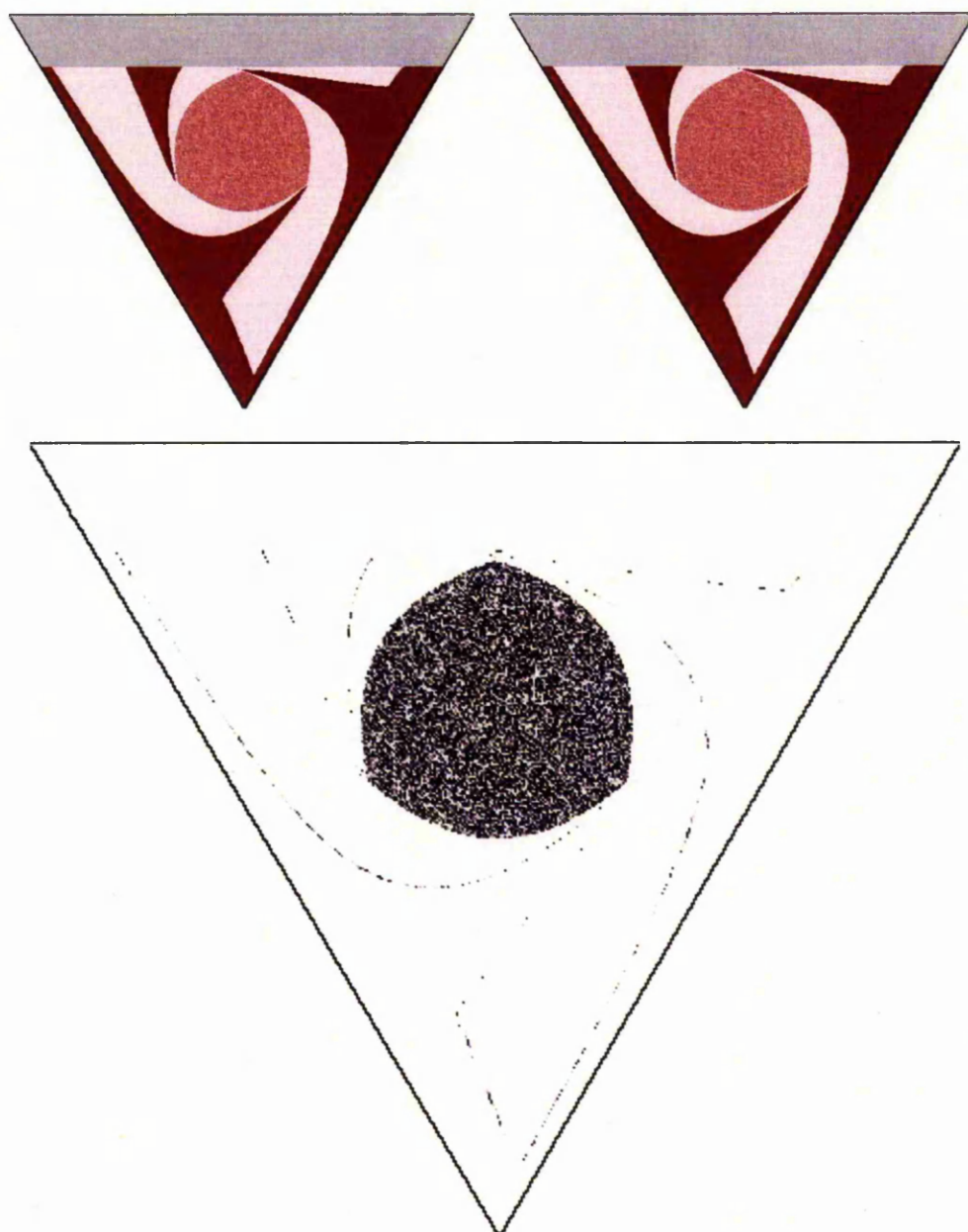


Figure 4.15: Upper: Two simulations of the triangular drum after ten revolutions at a fill level of 75% with 50% initial particle ratios subject to uniform noise in  $[-0.15, 0.15]$ . Lower: An enhanced image of the difference between the two upper figures.



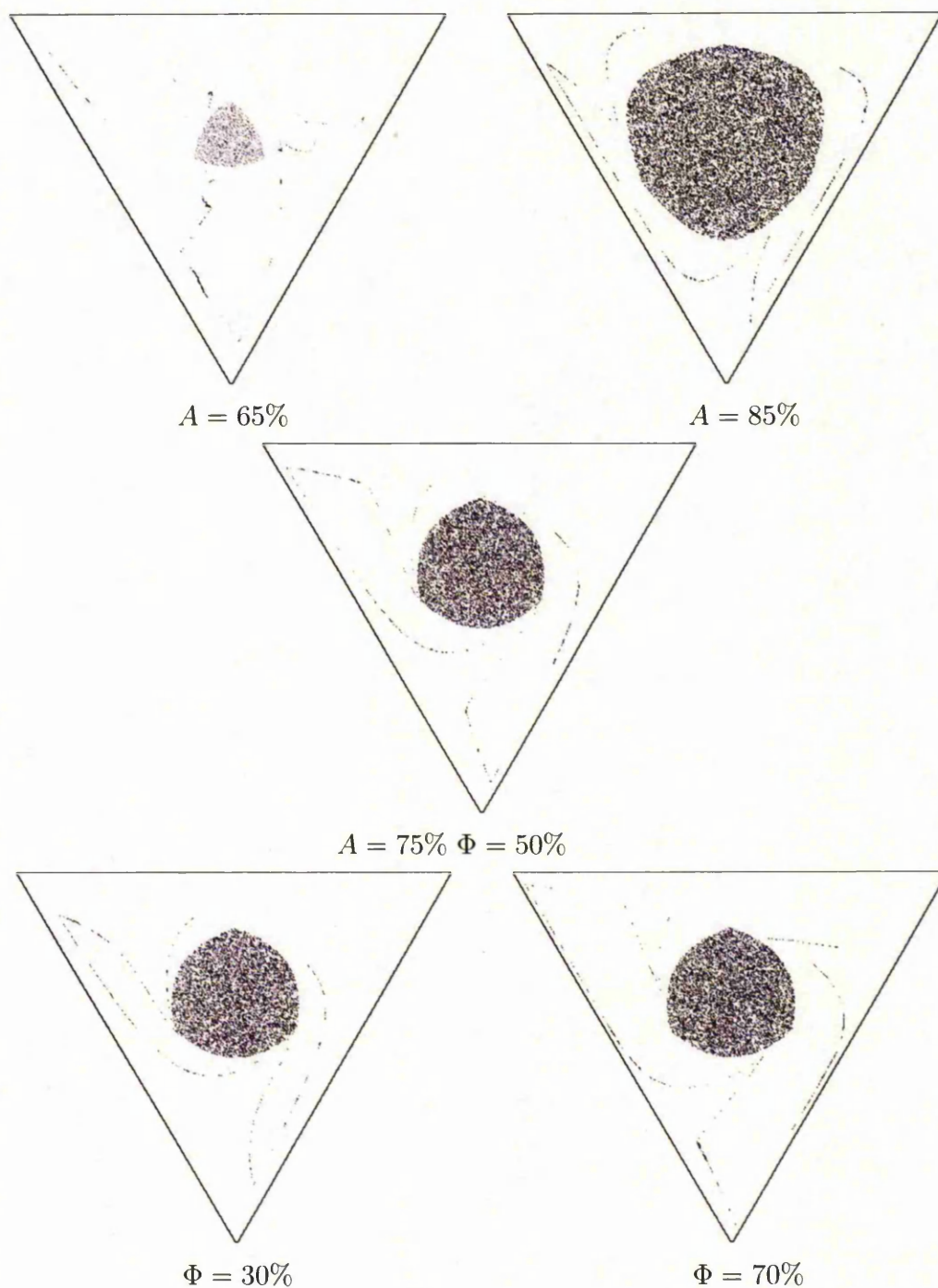


Figure 4.16: Simulations of the triangular drum after ten revolutions differenced as in figure 4.15. The middle image represents a reference state with a 75% fill level and a 50% particle ratio. The upper two images represent a higher and a lower fill level than the reference for the same particle ratio. The lower two images represent a higher and a lower particle ratio for a fixed fill level.

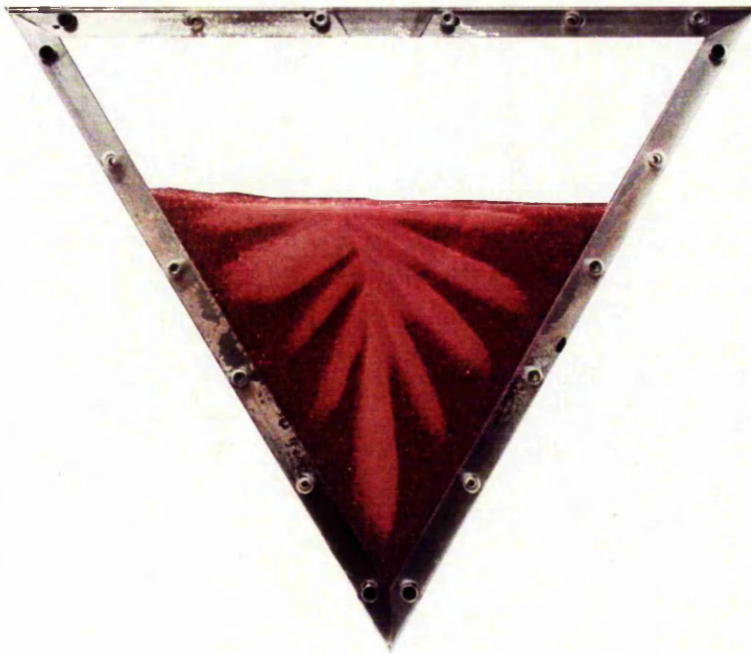


Figure 4.17: A photograph of a triangular drum with a 50% fill and 50% particle ratio at  $\theta = 20\pi$ .

#### 4.5.3 fifty percent fill

Clearly there is some interesting behaviour present in the simulation near the 50% fill level. To determine the relevance of this to the experiment a typical experimental image at near 50% fill will be examined in figure 4.17. Compare this to the comparable image of the simulation with the same parameters shown in figure 4.18. The resulting images are completely different in these two cases and apparently the simulation and approach have failed completely. This is not however completely true. In the very early time of the experiment the simulation's agreement is comparable to other simulations presented. Unlike the other experiments which immediately begin to present their final form, this experiment develops a different, petal like, structure over the course of several revolutions. The number of petals present also depends on the rotation rate of the drum, within the given regime.



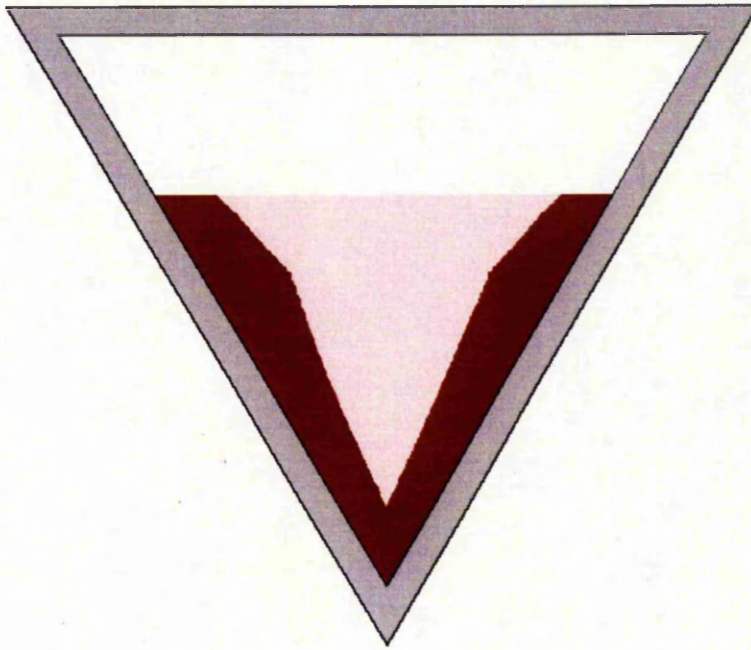


Figure 4.18: A simulation of a triangular drum with a 50% fill and 50% particle ratio at  $\theta = 20\pi$ .

This implies that the observed behaviour is in fact a separate mechanism to what is observed in other situations. [ZGPM06] This behaviour is explained to some extent in [ZGPM06]. In particular it is thought that avalanche dynamics, which are lost in the mapping approach, are critical to the phenomenon. In fact in the correct rotation rate regime, the number of petals present is inversely proportional to the rotation rate.

# Chapter 5

## General convex drums

Due to the generality of this method a similar analysis can be carried out for any convex figure. To this end the next most complex regular polygon is considered.

### 5.1 The square drum

The square represents the first regular polygon with an even number of sides. This adds an extra level of complexity over the triangular drum, while remaining similar enough to allow for some comparisons to be drawn.

#### 5.1.1 Comparison to experiment

As with the triangular drum, we examine the experiment's progression. Figure 5.1 illustrates a set of unrotated and unprocessed images in the laboratory's frame of reference.

Rather than including the complete comparison of section 4.1, a comparison of long term behaviour will be immediately undertaken. The behaviour of the experiment will first be examined in figure 5.2. As before a 75% fill level and 50%

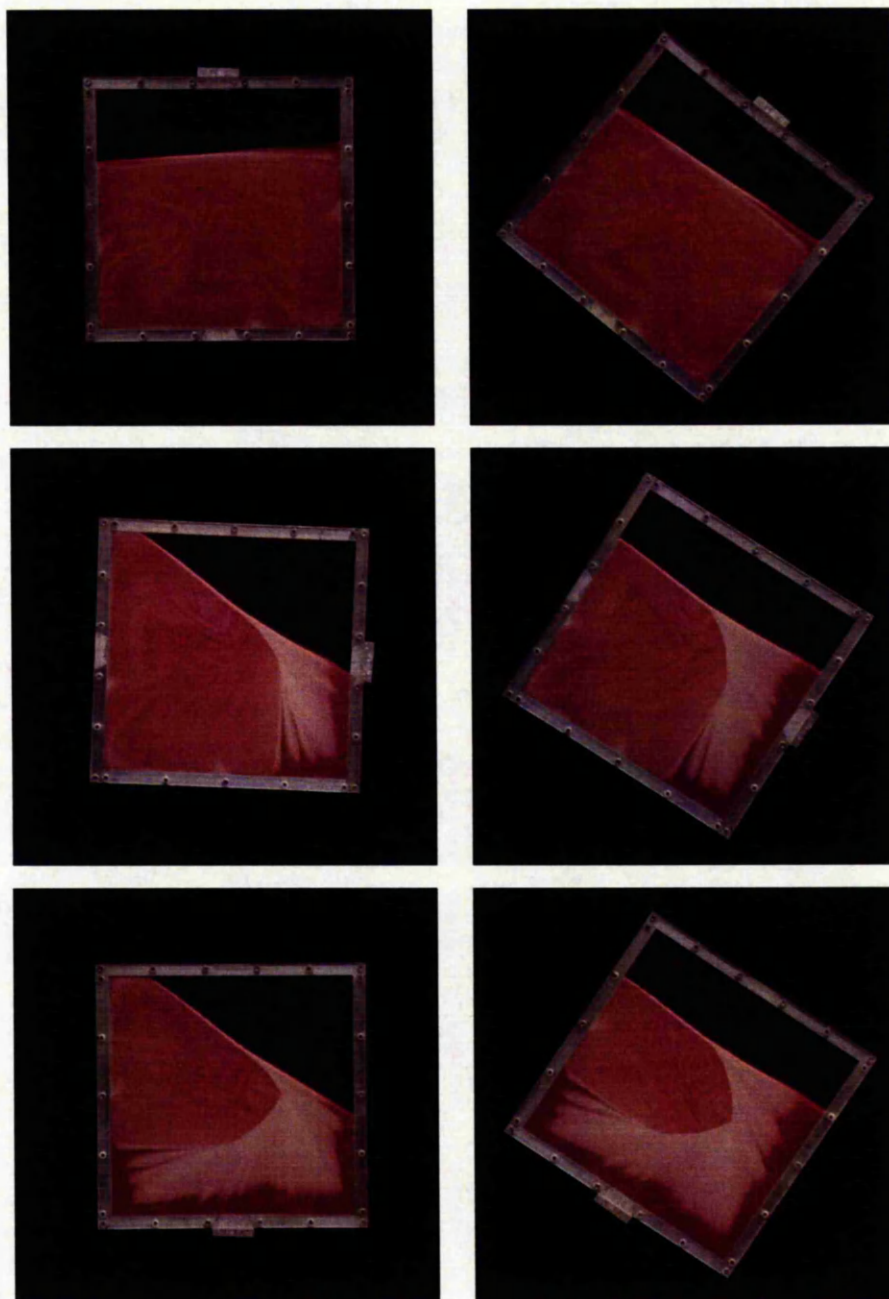


Figure 5.1: A series of photos of the square drum in the laboratory's frame of reference of the development of the segregation interface.



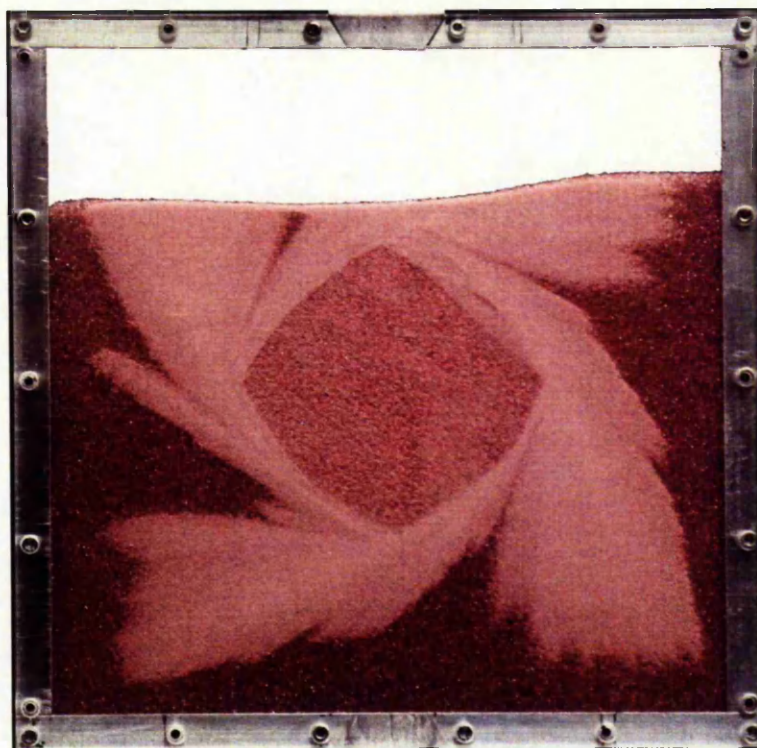


Figure 5.2: A photograph of a square drum with a 75% fill level and 50% particle ratio after 10 revolutions.

particle ratio are chosen. Figure 5.3 shows a comparable simulation. The square clearly shows strong similarities to the triangle. The level of agreement between the simulation and the experiment is similarly good. There is a lobed structure with a lobe pointing into each corner and sharp segregation at the segregation interface. The core of revolution is clearly visible and in the same way the structure appears to have settled to a relatively static configuration. As before comparison of results at various parameter combinations will be carried out.

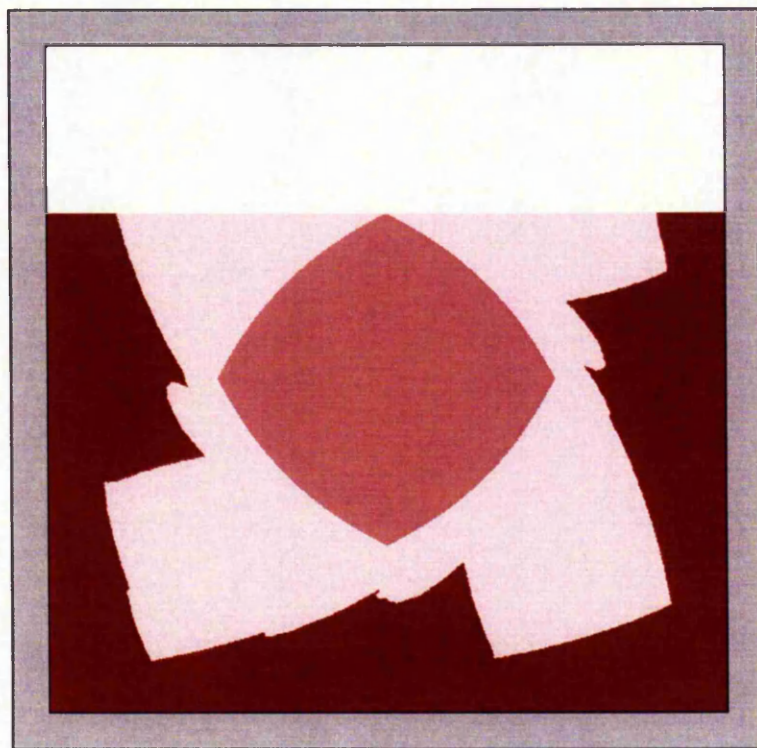


Figure 5.3: A simulation of a square drum with a 75% fill level and 50% particle ratio after 10 revolutions.

### 5.1.2 Varying ratios

Due to the similarity between the square and the triangle comparison with simulation will proceed immediately. As before a more thorough analysis of the dynamics of the system is undertaken. For a fixed fill level, the effect of varying the ratio of large to small particles,  $\Phi \in (0, 1)$ , is investigated. Images of the simulation after ten revolutions shall be compared to experiment. Figure 5.4 contains images of the simulation at six different particle ratios; the corresponding experimental images are shown in figure 5.5.

As with the triangle a straightforward lobed structure, with one lobe per corner, is formed. Unlike the triangle however, the number of secondary structures that



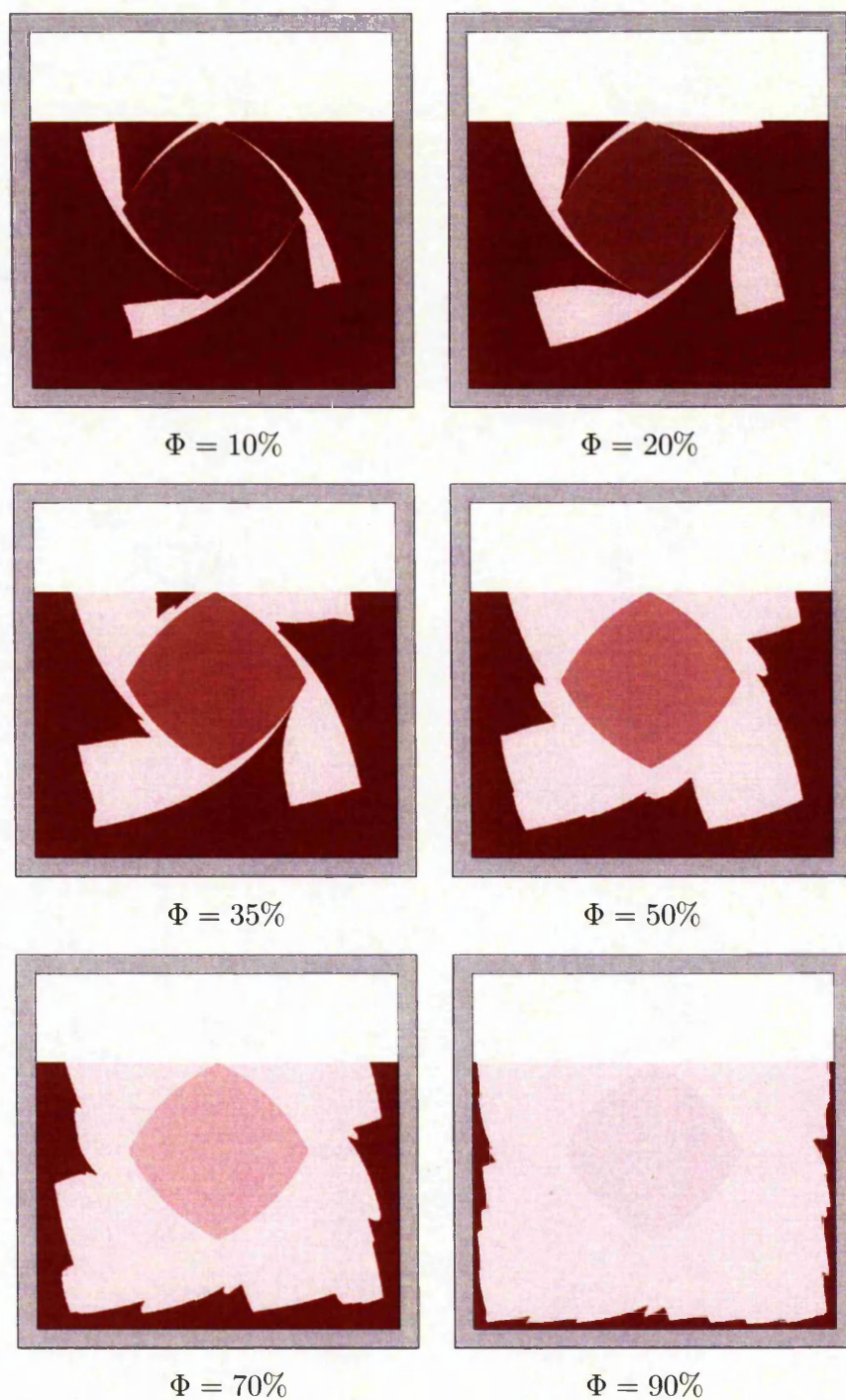


Figure 5.4: Simulations of the square drum after ten revolutions with various initial particle ratios at a fixed fill level of 75%.

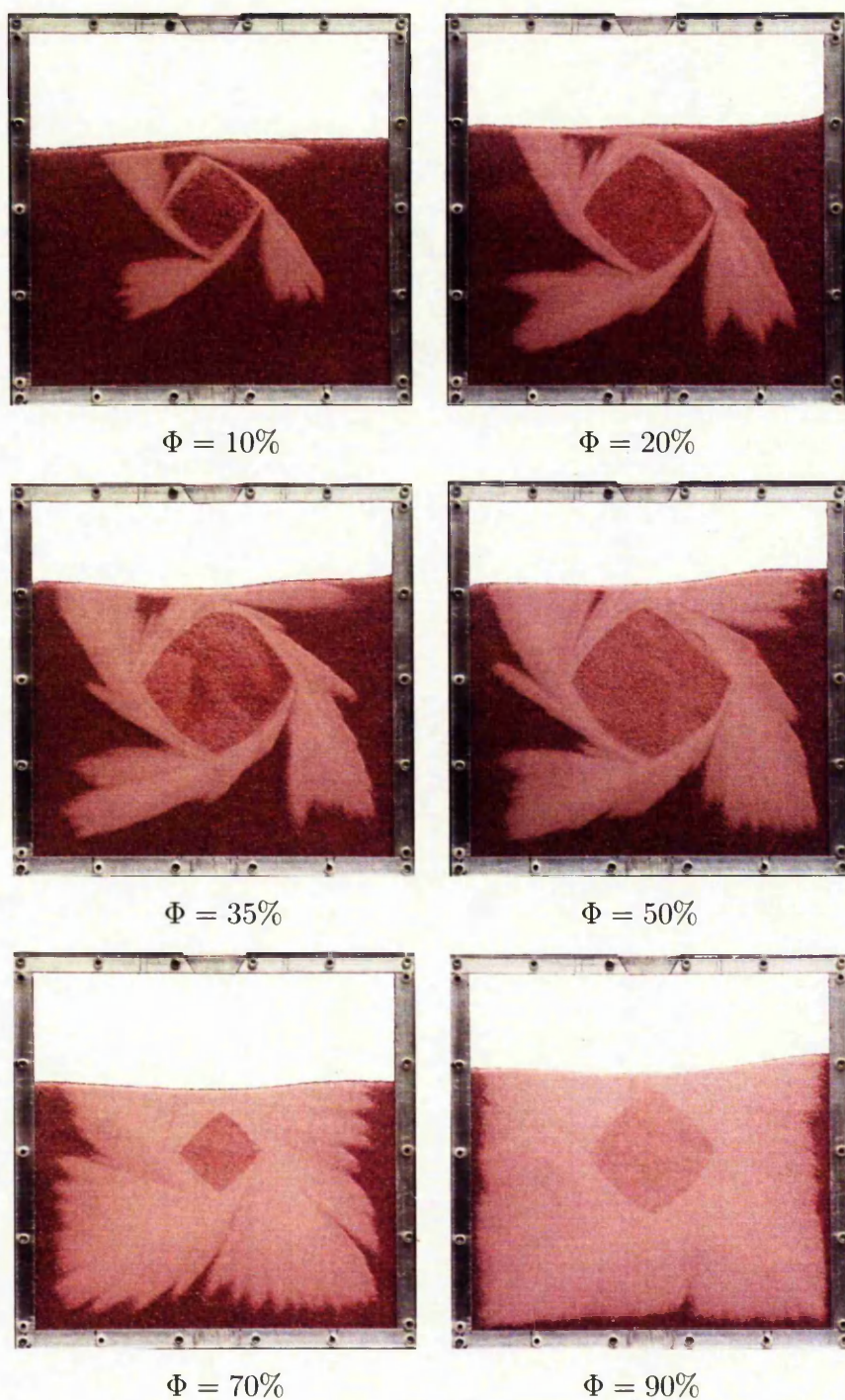


Figure 5.5: Experimental images of the square drum after ten revolutions with various initial particle ratios at a fixed fill level of 75%.



appear on these lobes grows rapidly with the particle ratio though their amplitude is initially relatively small compared to the lobes themselves. Rather than a smooth transition from large well defined lobes to a filled drum, the square appears to gain additional structure as the ratio shifts towards a large concentration of small particles. Further insight is gained by analysing the behaviour as the fill level varies.

### 5.1.3 Varying fill levels

Given the variation in structure present when the ratio of large to small particles is varied, the ratio is similarly fixed and the fill level varied. The figures resulting from this variation are shown in figure 5.6. Their experimental counterparts are illustrated in 5.7.

In the same way as for the triangle the high and low fill levels demonstrate straightforward primary lobe structures pointing into the corners. Similarly one can see the similarities in the patterns at fill levels above and below 50% due to the symmetry discussed in subsection 4.5.1. In contrast to the triangle however, the formation of jagged secondary structures appears at a much wider range of fill levels all the way from 30% to 70%. Given that the chosen fill level for the particle ratio varying was just outside of this range, the secondary structure formation may be due to this effect seen in an interval around 50% fill.

### 5.1.4 Long time behaviour

The same method is employed as was used for the triangle. Figure 5.8 shows a plot of equation 4.2. Similarly 50% is also a special case that oscillates around a small value. There are two other broad groups, those who limit to around 0.04

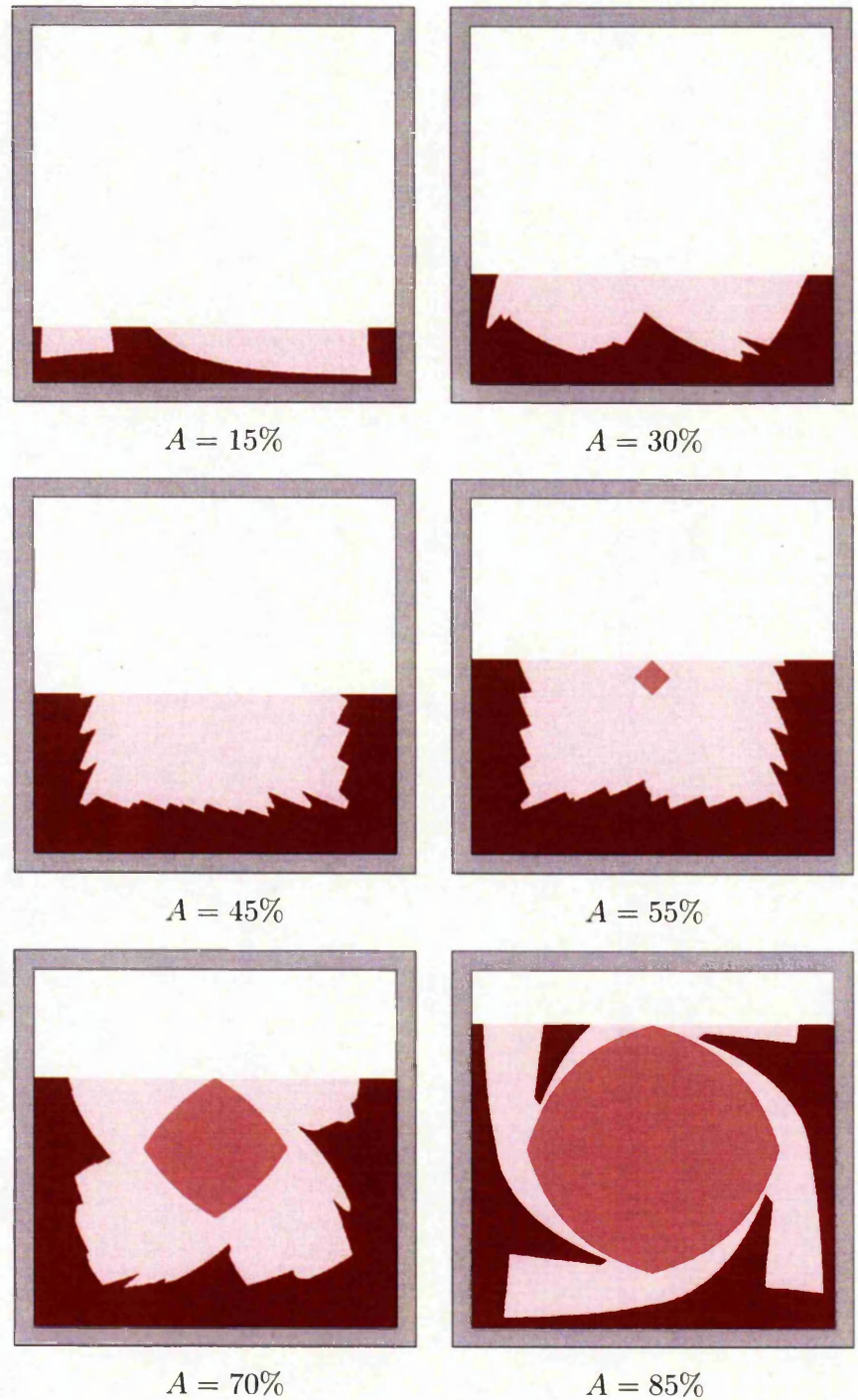


Figure 5.6: Simulations of the square drum after ten revolutions with a fixed initial particle ratio of 50% at various fill levels.

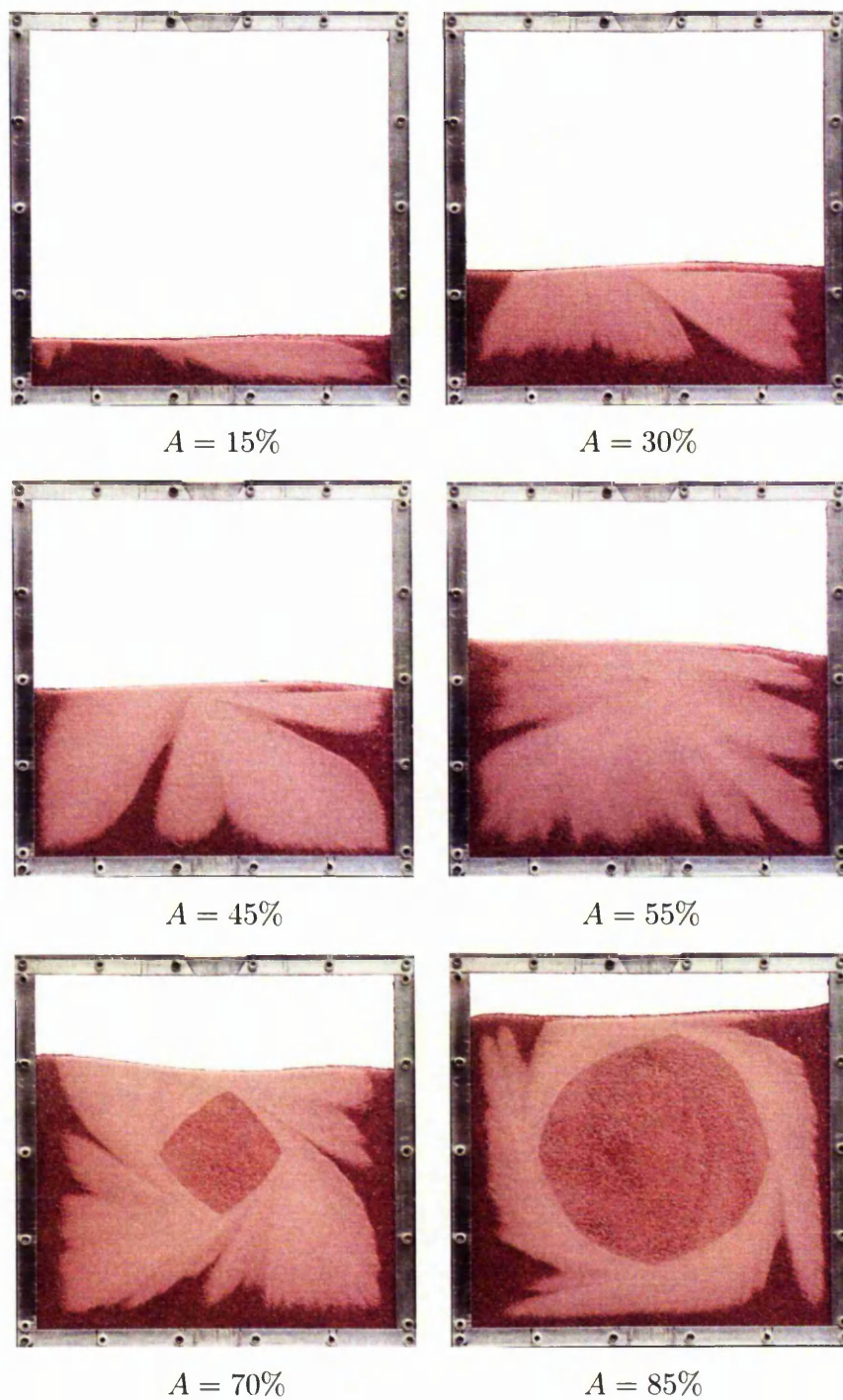


Figure 5.7: Experimental images of the square drum after ten revolutions with a fixed initial particle ratio of 50% at various fill levels.



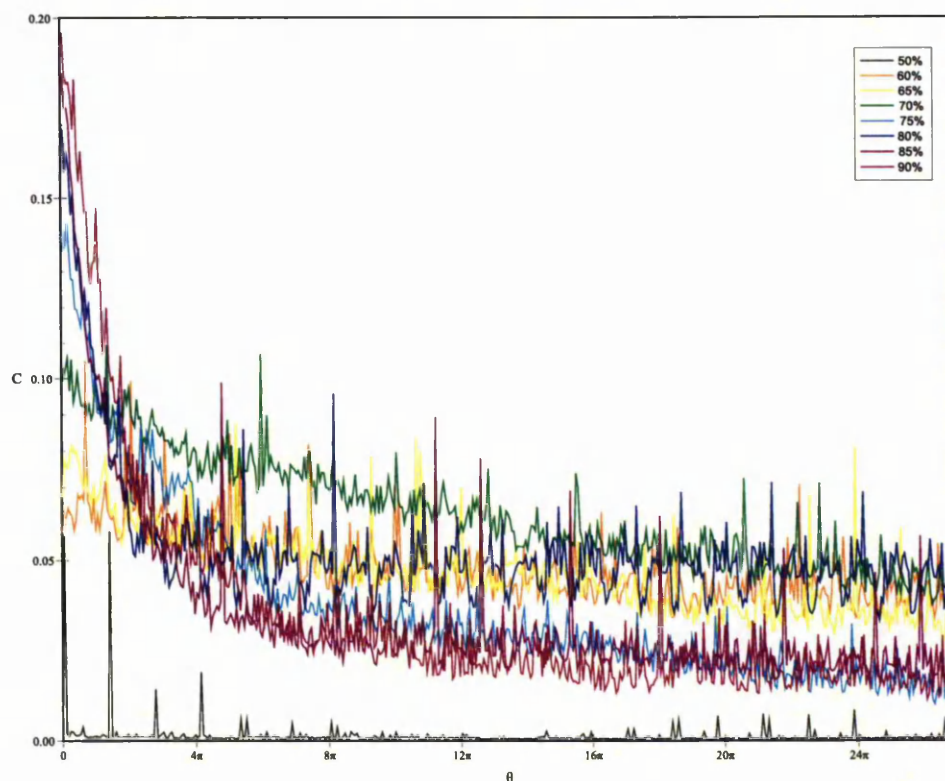


Figure 5.8: A graph of the difference in interface between subsequent revolutions,  $C$ , covering approximately 15 revolutions of the drum.

and those whose limit is around 0.02.

The average value in some tail region of specified length chosen so as all curves appear to have reached their limiting value. Given the apparent change in behaviour between 70% and 85%, data from several new points will be added to try and determine more thoroughly the nature of the functional relationship. The most important curves are rendered in black. Given the special nature of the 50% fill it will be omitted. These limiting values are plotted against the respective fill levels in figure 5.9. Clearly there is some strange behaviour in the 70% to 75% range, followed by, as before, a smooth, rapid change between 76% and 80%. In this case there appear to be two transitional regions as well as two other regions.



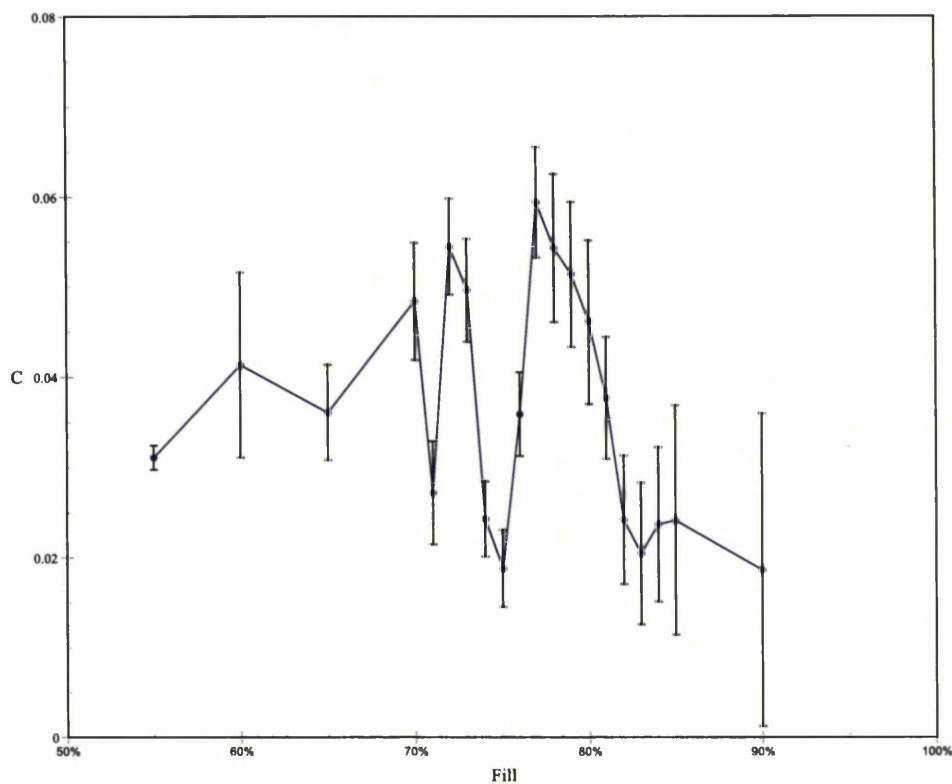


Figure 5.9: A graph of the approximate final value of  $C$  for varying fill levels. The error bars represent the standard deviation over the averaged interval.

1. 50% to 70% is a region which settles to a wide orbit around some state.
2. 70% to 75% is a region in which the orbit width varies erratically.
3. 75% to 80% is a region in which the orbit width drops linearly.
4. 80% to 100% is a region which settles to a narrow orbit around some state.

### 5.1.5 fifty percent fill

Figure 5.10 illustrates the experimental results for the square with a 50% fill level. There is a strong similarity between this behaviour and that illustrated in section 4.5.3. A simulation under a comparable parameter set is illustrated in figure

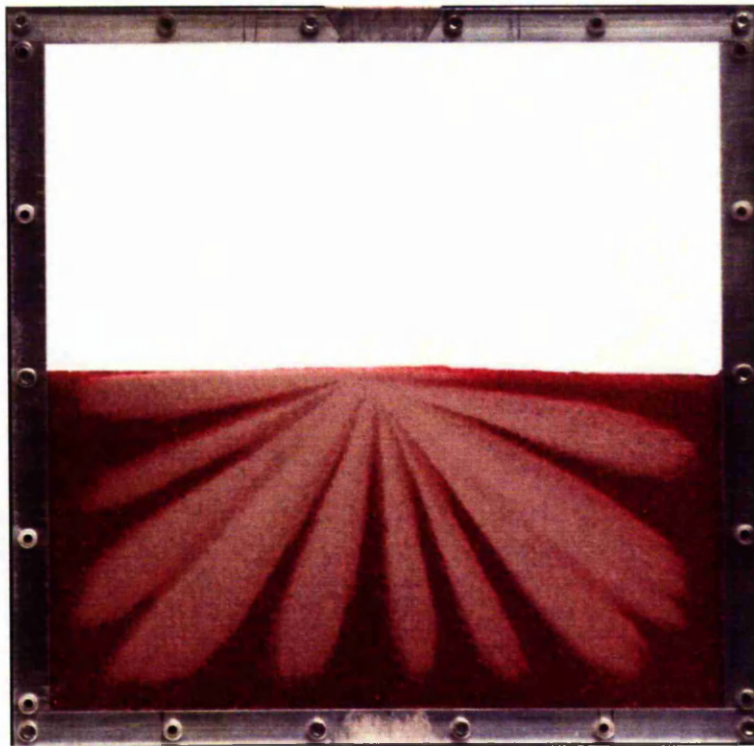


Figure 5.10: A photograph of a square drum with a 50% fill at  $\theta = 20\pi$ .

5.11. The resulting pattern bears a striking similarity to that observed for the triangular drum. In particular the same petal like structure is observed. The same relationship between the rotational frequency and the number and width of petals is observed as was the case with the triangle. The same relationship between the fill level and this dynamically variable behaviour is observed near the 50% fill level in all drum shapes examined.

## 5.2 Other drums

The volumetric method and its computational implementation are sufficiently general that any convex figure may be considered. Despite the drums considered thus

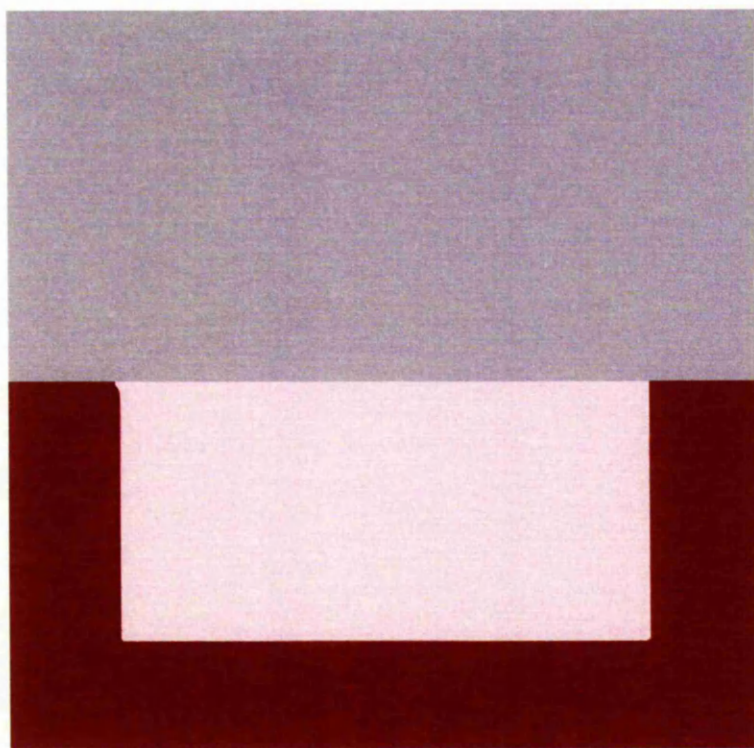


Figure 5.11: A simulation of a square drum with a 50% fill and 50% particle ratio at  $\theta = 20\pi$ .

far, there is no requirement that they be regular.

### 5.2.1 Curved drums

Due to the efficient nature of the numerical scheme figures with large, i.e. greater than one hundred, numbers of sides may be simulated with little performance penalty. This allows for approximations of general convex figures, including those with curved sides, to be simulated such as figure 5.12.

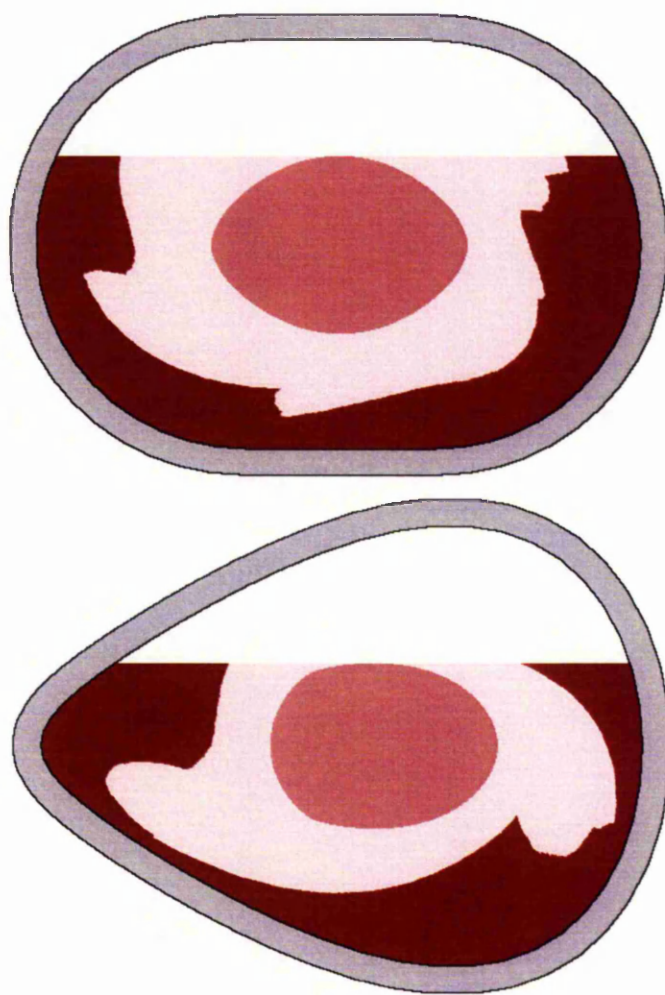


Figure 5.12: Simulations of stadium and egg shaped drums.

### 5.2.2 Affine mapping

An affine mapping is a linear mapping which preserves relative areas and colinearity of points. In any dimension affine mappings and invertible matrices with real coefficients composed with translations are equivalent.

Given that the volumetric method described hinges on integrals 3.11 and 3.14 and that the results for any given line are invariant under affine transformation, it is proposed that the system is invariant under affine transformation. An affine mapping by definition preserves colinearity of points and relative distances between them. This implies that the incoming material region, i.e. the up-slope half of the surface, corresponds precisely to the incoming region in the mapped domain. Since no  $\Omega$  appears in these equations, no account of the non-linear scaling of angles need be taken. All parameters are normalised by the half length of the surface, meaning that precisely the same integrations are carried out in each domain hence  $F$ , the normalised small particle flux, will be unaffected by mapping. In the same way the outgoing material's interfacial parameter  $T$  will be equally unaffected by the mapping meaning equation 3.15 will yield the same result. When the outgoing material is placed by mapping  $T$  from normalised coordinates onto real coordinates on the line, two intervals are obtained. Since affine mappings preserve relative distance these will correspond directly to the intervals in the unmapped domain. For this reason, the results obtained from a drum which has undergone an affine mapping, correspond to the results for the unmapped drum with the corresponding affine map applied.

One interesting consequence is that rather than a simulation representing a specific drum they may be considered to represent the whole equivalence class of drums that are affine transformations of the given configuration. In the case of the



triangle, a single simulation is sufficient to describe all possible triangular drums. Figure 5.13 compares the simulations of an equilateral and a scalene triangle; to allow for more ready comparison an affine mapped image of the scalene triangle to match the equilateral triangle is provided. It is worth remembering that a general affine transformation results in a non-linear relationship between angles, hence the angle to which the drum is rotated in each case is different. Under the same equivalence relation, a circle corresponds to all ellipses as exemplified by figure 5.14 and a square corresponds to all parallelograms and rectangles as shown in figure 5.15. Unlike the triangle however a square is not equivalent to general quadrilaterals as mapping, for example, a trapezium to a square requires a non-linear mapping. Finally a comparable result using an experimental image of a square and parallelogram drum are shown in figure 5.16.

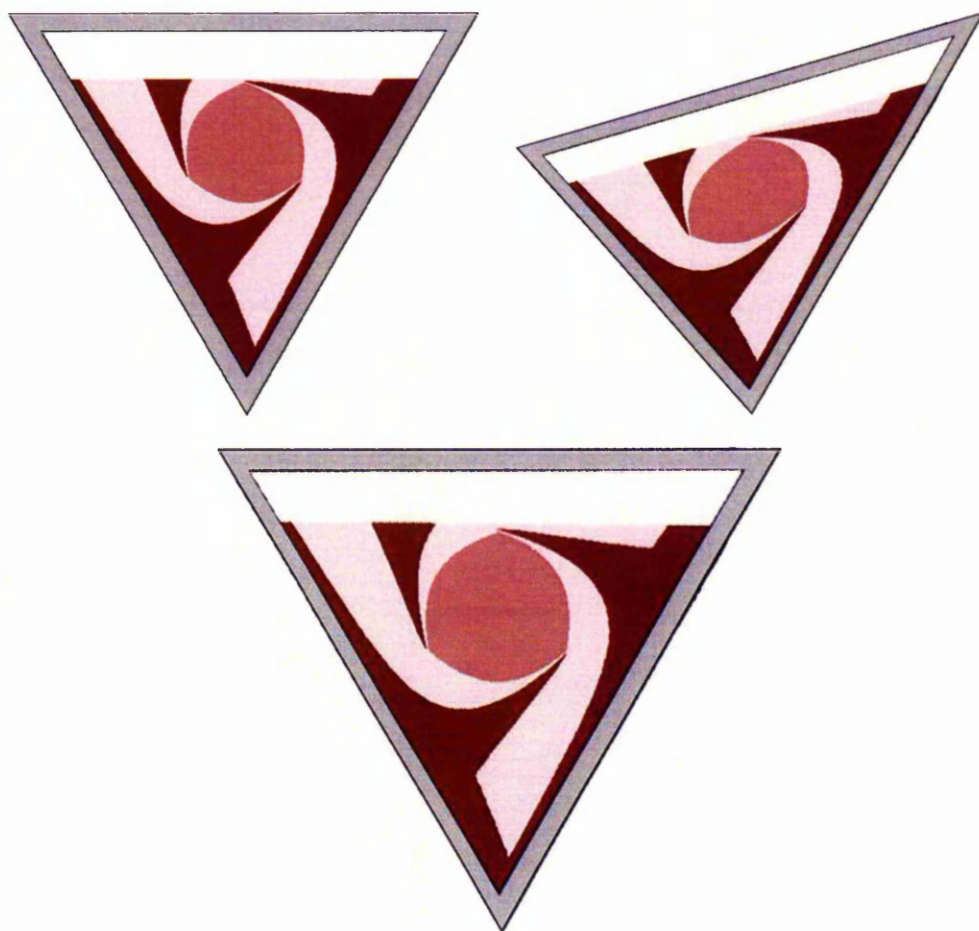


Figure 5.13: Upper: Simulations of equilateral and scalene triangular drums after ten revolutions at a fill level of 75% with 50% initial particle ratios. Lower: The figure of the scalene triangle affine mapped to match the shape of the equilateral triangle.

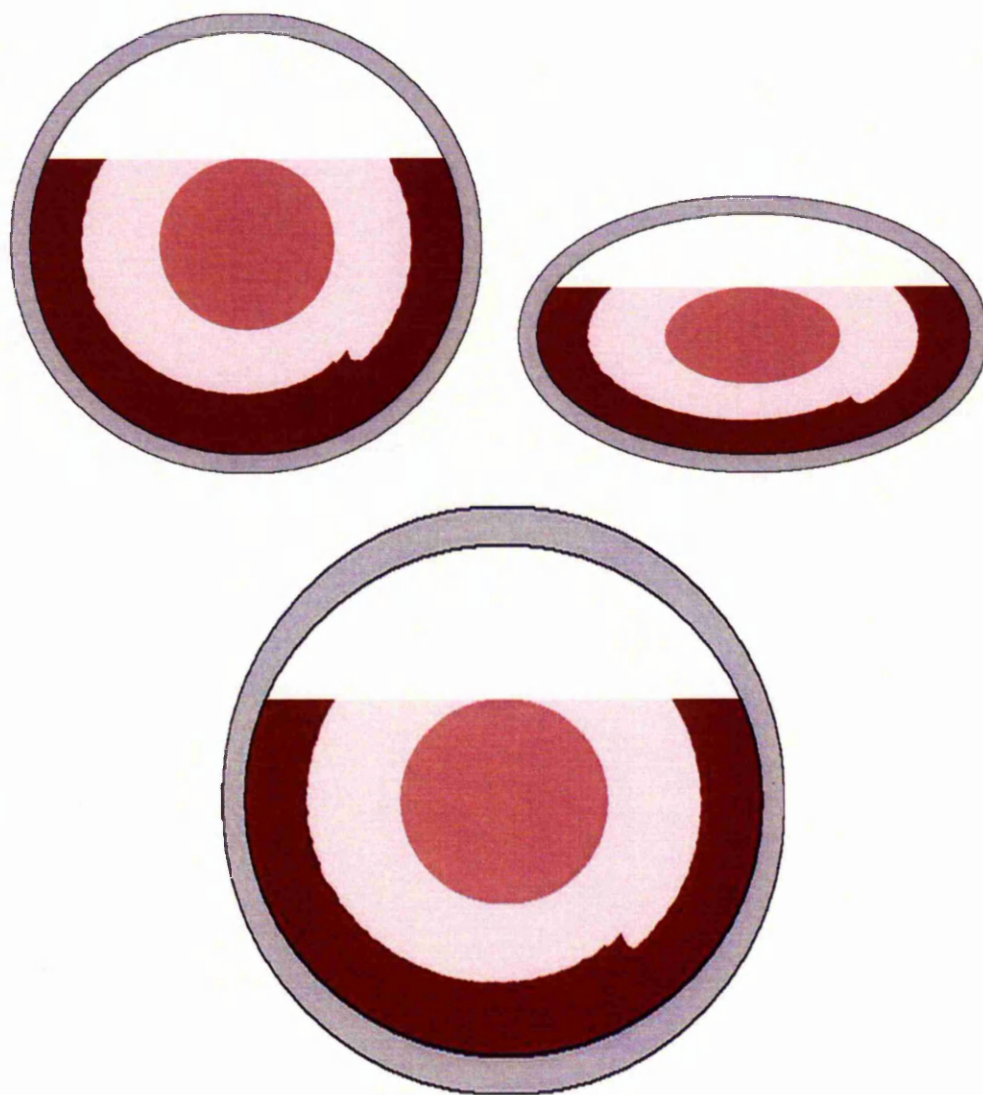


Figure 5.14: Upper: Simulations of circular and elliptical drums after ten revolutions at a fill level of 75% with 50% initial particle ratios. Lower: The figure of the ellipse affine mapped to match the circle.



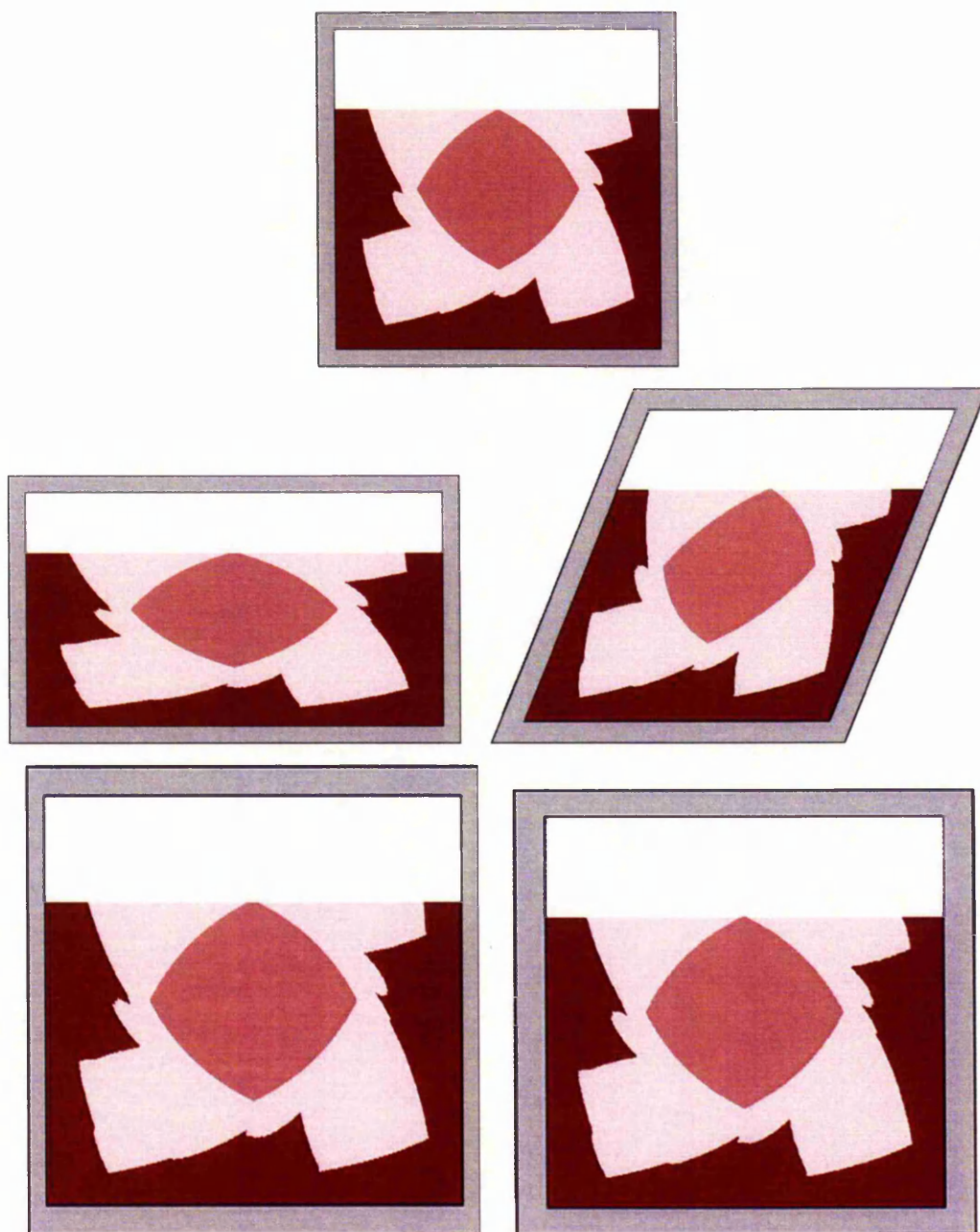


Figure 5.15: Upper: Simulations of square, parallelogram and rectangular drums after ten revolutions at a fill level of 75% with 50% initial particle ratios. Lower: The figures of the parallelogram and rectangle affine mapped to match the square.

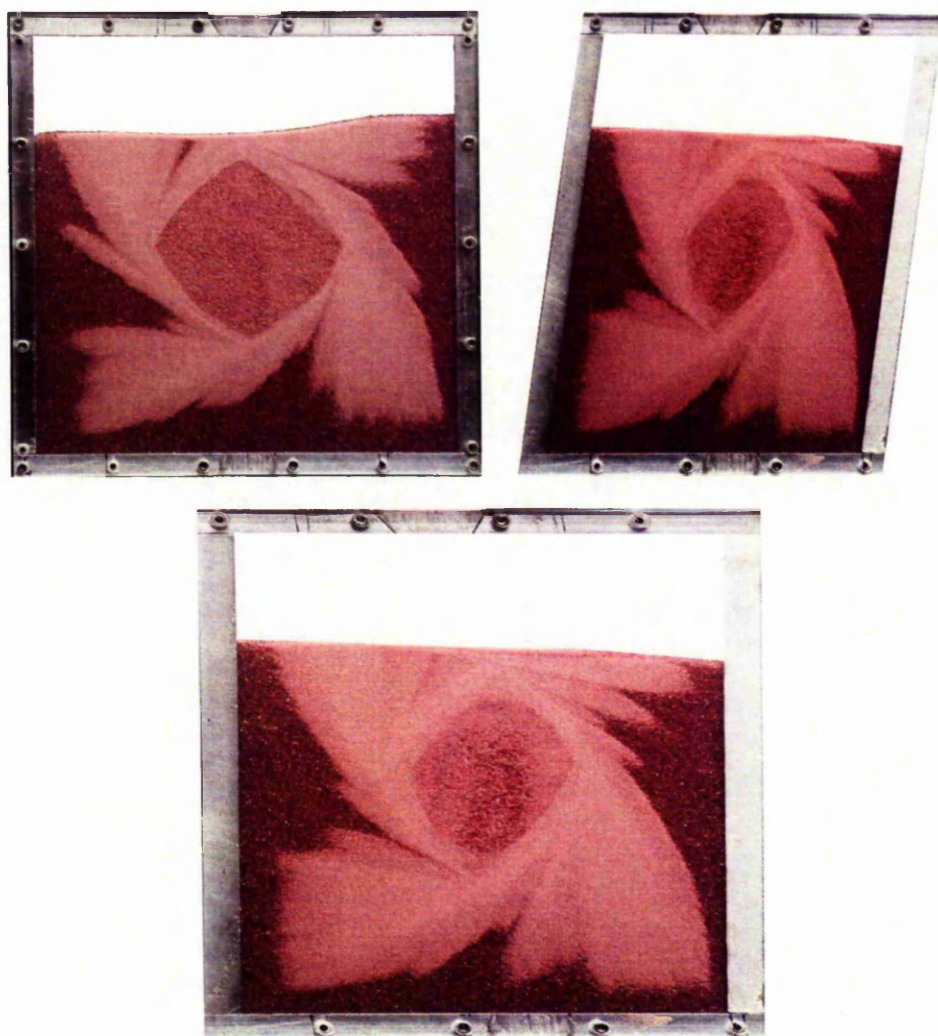


Figure 5.16: Upper: Experimental images of square and parallelogram drums after ten revolutions at a fill level of 75% with 50% initial particle ratios. Lower: The figure of the parallelogram affine mapped to match the shape of the square.

# Chapter 6

## Discussion

The goal to predict the behaviour in the short and long term of segregating material in a rotating drum has been reached. A simple mathematical approach has been implemented into an efficient computational method that is capable of general and accurate simulation. This approach can predict the form and position of the segregation interface in a generic manner. The affine mapping idea is equally unique and allows for a useful generalisation of any results to a family of drums.

The numerical implementation was developed in a unique manner and does not rely on typical methods associated with such systems. The result is a fast and accurate computational method that may be generalised and applied to other situations. As well as being applicable to the segregating case considered, the same methodology can be applied to further studying the chaos and dynamics in the non segregating variant.

The fill levels that are accurately predicted, primarily below forty percent, correspond to fill levels typically used in industry. These are chosen so as to avoid the formation of an unmixed core of revolution within the flow. In real industrial systems fins and blades are present introducing concavities to the tumblers.

This work provides a foundation for a simulation scheme that could be directly applicable in industrial settings.

The next step for such a simulation might include the consideration and treatment of concavities within the drum. Another option might be to provide a full coupling of the behaviour within the avalanche to the rotating bulk. This may potentially allow for even more accurate predictions at all fill levels. If both additions were made the volumetric method should be able to predict with complete generality.

# Bibliography

- [Bar94] G.C. Barker. *Granular Matter: An Interdisciplinary Approach*. SpringerVerlag, 1994.
- [Bri76] J. Bridgwater. Fundamental powder mixing mechanisms. *Powder Technology*, 15:215–236, December 1976.
- [EJK<sup>+</sup>95] E.E. Ehrichs, H.M. Jaegar, G.S. Karczmar, J.B. Knight, V.Y. Kuiperman, and S.R. Nagel. Granular convection observed by magnetic-resonance-imaging. *Science*, 17:1632–1634, March 1995.
- [EV99] T. Elperin and A. Vikhansky. Chaotic mixing of granular material in slowly rotating containers as a discrete mapping. *Chaos*, 9(4):910–915, December 1999.
- [GC06] J.M.N.T. Gray and V.A. Chuganov. Particle-size segregation and diffusive remixing in shallow granular avalanches. *Journal of Fluid Mechanics*, 569:365–398, 2006.
- [Gla90] A.S. Glassner. *Graphics Gems*. Academic Press, 1990.
- [GM07] J.M.N.T. Gray and T. Mullin. Granular caustics and cores of revolution. Incomplete at present, 2007.



- [Gra01] J.M.N.T. Gray. Granular flow in partially filled slowly rotating drums. *Journal of Fluid Mechanics*, 441:1–29, August 2001.
- [GT05] J.M.N.T. Gray and A.R. Thornton. A theory of particle size segregation in shallow granular free-surface flows. *Proceedings of the Royal Society A*, 461:1447–1473, April 2005.
- [HKG<sup>+</sup>99] K.M. Hill, D.V. Khakhar, J.F. Gilchrist, J.J. McCarthy, and J.M. Ottino. Segregation-driven organization in chaotic granular flows. *Proceedings of the National Academy of Sciences*, 96(21):11701–11706, October 1999.
- [JLL<sup>+</sup>98] G.S. Jiang, D. Levy, C.T. Lin, S. Osher, and E. Tadmor. High-resolution nonoscillatory central schemes with nonstaggered grids for hyperbolic conservation laws. *SIAM Journal of Numerical Analysis*, 35(6):2147–2168, December 1998.
- [JS83] J.T. Jenkins and S.B. Savage. A theory for the rapid flow of identical, smooth, nearly elastic, spherical particles. *Journal of Fluid Mechanics*, 130:187–202, 1983.
- [MLJ07] S.W. Meier, R.M. Lueptow, and Ottino J.M. A dynamical systems approach to mixing and segregation of granular materials in tumblers. *Advances in Physics*, 56:757–827, September 2007.
- [MSMJ95] G. Metcalfe, T. Shinbrot, J.J. McCarthy, and Ottino J.M. Avalanche mixing of granular solids. *Nature*, 374:39–41, March 1995.
- [OK00] J.M. Ottino and D.V. Khakhar. Mixing and segregation of granular materials. *Annual Review of Fluid Mechanics*, 32:55–91, January 2000.

- [SC99] M.L. Sawley and P.W. Cleary. A parallel discrete element method for industrial granular flow simulations. *Ecole Polytechnique Fédérale de Lausanne Supercomputing Review*, November 1999.
- [SDS01] R.J. Spurling, J.F. Davidson, and D.M. Scott. The transient response of granular flow in an inclined rotating cylinder. *Transactions of the Institution of Chemical Engineers*, 79(A):51–61, January 2001.
- [SH89] S.B. Savage and K. Hutter. The motion of a finite mass of granular material down a rough incline. *Journal of Fluid Mechanics*, 199:177–215, March 1989.
- [SL88] S.B. Savage and C.K.K. Lun. Particle size segregation in inclined chute flow of dry cohesionless granular solids. *Journal of Fluid Mechanics*, 189:311–335, April 1988.
- [SM00] T. Shinbrot and F.J. Muzzio. Nonequilibrium patterns in granular mixing and segregation. *Physics Today*, 53(3):25–30, March 2000.
- [Tab95] S. Tabachnikov. On the dual billiard problem. *Advances in Mathematics*, 115(2):221–249, October 1995.
- [TGH06] A.R. Thornton, J.M.N.T. Gray, and A.J. Hogg. A three-phase mixture theory for particle size segregation in shallow granular free-surface flows. *Journal of Fluid Mechanics*, 550:1–25, 2006.
- [Wil88] B.A. Wills. *Mineral Processing Technology*. Pergamon Books inc., January 1988.

- [ZGPM06] I. Zuriguel, J.M.N.T. Gray, J. Peixinho, and T. Mullin. Pattern selection by a granular wave in a rotating drum. *Physical Review E*, 73(061302), June 2006.

# Appendix A

## Non oscillatory central differencing source code

### A.1 main.cc

```
#include <ctime>
```

```
#include <cmath>
```

```
#include <iostream>
```

```
#include <SDL.h>
```

```
#include "drawing.h"
```

```
struct point
```

```
{
```

```
    float v;
```

```
    float E;
```

```
    float N;
```

```
};

const bool doubled=false;
const float X=2.0f;
const float Z=1.0f;
const int NZ=250;
const int NX=static_cast<int>(roundf((X/Z)*NZ));
const float dX=X/static_cast<float>(NX);
const float dZ=Z/static_cast<float>(NZ);
    float dT;
const int TX=NX*2+7;
const int TZ=NZ*2+7;
const float left_incoming=0.5f;
const float right_incoming=0.5f;

const float CFL=0.5f;
const float Sr=1.0f;

inline float limit(const float& a, const float& b, const float& c)
{
    float retval=0.0f;
    if (b*c>0.0f) {
        if (fabsf(b)<fabsf(c)) retval=b;
        else retval=c;
    }
    if (a*retval>0.0f) {
        if (fabsf(a)<fabs(retval)) retval=a;
    }
    return retval;
}
```



```

inline void clip(float& f)
{
    if (f<0.0f) f=0.0f;
    if (f>1.0f) f=1.0f;
}

int main(int argc, char* argv[])
{
    srand(time(0));

    SDL_Init(SDL_INIT_VIDEO);
    SDL_Surface *screen;
    if (doubled) screen=SDL_SetVideoMode(NX*2, NZ*2, 0, SDL_SWSURFACE | /
        SDL_DOUBLEBUF);
    else screen=SDL_SetVideoMode(NX, NZ, 0, SDL_SWSURFACE | SDL_DOUBLEBUF);
    SDL_Surface* mask=SDL_CreateRGBSurface(screen->flags, screen->w, screen->h, /
        screen->format->BitsPerPixel, screen->format->Rmask, screen->format->Bmask,/
        screen->format->Gmask, screen->format->Amask);
    SDL_Surface* image=SDL_CreateRGBSurface(screen->flags, screen->w, screen->h, /
        screen->format->BitsPerPixel, screen->format->Rmask, screen->format->Bmask,/
        screen->format->Gmask, screen->format->Amask);

    point v[TX][TZ];

    float u[TZ];

    float umax=0.0f;

    for(int zt=0;zt<TZ;zt++) {
        float z=(zt-3)*dZ*0.5f;
        u[zt]=0.2f+z*1.0f;//z-Z*0.5f;
    }

```

```

//u[zt]=0.8;
if (fabsf(u[zt])>umax) umax=fabsf(u[zt]);
}

// Set the time step based on CFL number
{
    float dXT=0.5f*(dX*CFL)/umax;
    float dZT=0.5f*(dZ*CFL)/(Sr+0.00000000000001f);
    if (dXT<dZT) dT=dXT;
    else dT=dZT;
}

// Construct the initial data
for (int xt=4;xt<TX-4;xt+=2) {
    float x=(xt-3)*dX*0.5f;
    float value=1.0f;
    if (x<X/2) value=0.0f;
    for (int zt=4;zt<TZ-4;zt+=2) {
        float z=(zt-3)*dX*0.5f;
        value=0.5f;// Added
        v[xt][zt].v=value;
        v[xt][zt].N=0.0f;
        v[xt][zt].E=0.0f;
    }
}

// Draw the Mask
{
    SDL_FillRect(mask, NULL, 0x01);
    SDL_SetColorKey(mask, SDL_SRCCOLORKEY|SDL_RLEACCEL, 0x01);
    int width=NX;

```

```
int height=NZ;
if (doubled) {
    width=NX*2;
    height=NZ*2;
}
for (int x=0;x<width;x++) {
    float X=2.0f*static_cast<float>(x)/static_cast<float>(width)-1.0f;
    int limit=height-1-static_cast<int>(X*X*height);
    for (int z=limit;z<height;z++) {
        putpixel(mask, x, z, 0xFF);
    }
}

unsigned int firstTime=SDL_GetTicks();
float t=0.0f;
int count=0;
bool done=false;

while(!done)
{
    SDL_Event event;
    while (SDL_PollEvent(&event)) {
        switch (event.type) {
            case SDL_KEYDOWN:
                switch (event.key.keysym.sym) {
                    case SDLK_q:
                    case SDLK_ESCAPE:
                        done=true;
                        break;
                }
            }
        }
    }
```

```

    break;
case SDL_QUIT:
    done=true;
    break;
}
}

//Parabolic inflow
for (int xt=0;xt<TX;xt++) {
    float x=(xt-3)*0.5f*dX-1.0f;
    for (int zt=0;zt<TZ;zt++) {
        float z=(zt-3)*0.5f*dX;
        if (x<0.0f&&z<x*x) v[xt][zt].v=(sqrtf((x+Sr)*(x+Sr)-4.0f*Sr*x*0.1f)+x+Sr)/(2.0f*/
Sr);
    }
}

// Set even valued boundary data
{
    // Side Boundaries
    for (int zt=4;zt<TZ-4;zt+=2) {
        // Left Boundary
        if (u[zt]>0.0f) {
            v[2][zt].v=left_incoming;
            v[0][zt].v=left_incoming;
        } else {
            v[2][zt].v=2.0f*v[4][zt].v-v[6][zt].v;
            clip(v[2][zt].v);
            v[0][zt].v=3.0f*v[4][zt].v-2.0f*v[6][zt].v;
            clip(v[0][zt].v);
        }
    }
}

```

```

// Right Boundary
if (u[zt]<0.0f) {
    v[TX-3][zt].v=right_incoming;
    v[TX-1][zt].v=right_incoming;
} else {
    v[TX-3][zt].v=2.0f*v[TX-5][zt].v- v[TX-7][zt].v;
    clip(v[TX-3][zt].v);
    v[TX-1][zt].v=3.0f*v[TX-5][zt].v-2.0f*v[TX-7][zt].v;
    clip(v[TX-1][zt].v);
}
}

// Upper and Lower Boundaries
for (int xt=4;xt<TX-4;xt+=2) {
    // Lower Boundary
    v[xt][2].v=0.0f;//2.0f*v[xt][4].v- v[xt][6].v;
    clip(v[xt][2].v);
    v[xt][0].v=0.0f;//3.0f*v[xt][4].v-2.0f*v[xt][6].v;
    clip(v[xt][0].v);
    // Upper Boundary
    v[xt][TZ-3].v=1.0f;//2.0f*v[xt][TZ-5].v- v[xt][TZ-7].v;
    clip(v[xt][TZ-3].v);
    v[xt][TZ-1].v=1.0f;//3.0f*v[xt][TZ-5].v-2.0f*v[xt][TZ-7].v;
    clip(v[xt][TZ-1].v);
}
}

float lambda=dT/dX;
float mu=dT/dZ;

// Timestep the Odds
{

```

```

for (int xt=1;xt<TX-3;xt+=2) {
  for (int zt=1;zt<TZ-3;zt+=2) {
    float bx=limit(2.0f*(v[xt+1][zt+1].v-v[xt-1][zt+1].v),0.5f*(v[xt+3][zt+1].v-v[xt/
-1][zt+1].v),2.0f*(v[xt+3][zt+1].v-v[xt+1][zt+1].v));
    float bz=limit(2.0f*(v[xt+1][zt+1].v-v[xt+1][zt-1].v),0.5f*(v[xt+1][zt+3].v-v[xt/
+1][zt-1].v),2.0f*(v[xt+1][zt+3].v-v[xt+1][zt+1].v));
    float fdb=bx*u[zt+1];
    float gdb=bz*Sr*(1.0f-2.0f*v[xt+1][zt+1].v);
    float bth=v[xt+1][zt+1].v-lambda*0.5f*fdb-mu*0.5f*gdb;
    if (xt>=3) {
      float ax=limit(2.0f*(v[xt-1][zt+1].v-v[xt-3][zt+1].v),0.5f*(v[xt+1][zt+1].v-v[
xt-3][zt+1].v),2.0f*(v[xt+1][zt+1].v-v[xt-1][zt+1].v));
      float az=limit(2.0f*(v[xt-1][zt+1].v-v[xt-1][zt-1].v),0.5f*(v[xt-1][zt+3].v-v[
xt-1][zt-1].v),2.0f*(v[xt-1][zt+3].v-v[xt-1][zt+1].v));
      float fda=ax*u[zt+1];
      float gda=az*Sr*(1.0f-2.0f*v[xt-1][zt+1].v);
      float ath=v[xt-1][zt+1].v-lambda*0.5f*fda-mu*0.5f*gda;
      v[xt][zt].N=0.0625f*(ax-bx)-lambda*0.5f*(bth*u[zt+1]-ath*u[zt+1]);
    }
    if (zt>=3) {
      float cx=limit(2.0f*(v[xt+1][zt-1].v-v[xt-1][zt-1].v),0.5f*(v[xt+3][zt-1].v-v[
xt-1][zt-1].v),2.0f*(v[xt+3][zt-1].v-v[xt+1][zt-1].v));
      float cz=limit(2.0f*(v[xt+1][zt-1].v-v[xt+1][zt-3].v),0.5f*(v[xt+1][zt+1].v-v[xt/
+1][zt-3].v),2.0f*(v[xt+1][zt+1].v-v[xt+1][zt-1].v));
      float fdc=cx*u[zt-1];
      float gdc=cz*Sr*(1.0f-2.0f*v[xt+1][zt-1].v);
      float cth=v[xt+1][zt-1].v-lambda*0.5f*fdc-mu*0.5f*gdc;
      v[xt][zt].E=0.0625f*(cz-bz)-mu*0.5f*(Sr*bth*(1.0f-bth)-Sr*cth*(1.0f-cth));
    }
  }
}
}

```



```

// Update Odds based on Fluxes
for (int xt=3;xt<TX-3;xt+=2) {
    for (int zt=3;zt<TZ-3;zt+=2) {
        v[xt][zt].v=0.25f*(v[xt-1][zt-1].v+v[xt+1][zt-1].v+v[xt+1][zt+1].v+v[xt-1][zt+1].v);
        v[xt][zt].v+=v[xt][zt].N+v[xt][zt].E+v[xt-2][zt].E+v[xt][zt-2].N;
    }
}
}

```

```

// Set odd valued boundary data
{
    // Side Boundaries
    for (int zt=3;zt<TZ-3;zt+=2) {
        // Left Boundary
        if (u[zt]>0.0f) {
            v[3][zt].v=left_incoming;
            v[1][zt].v=left_incoming;
        } else {
            v[3][zt].v=2.0f*v[5][zt].v- v[7][zt].v;
            clip(v[3][zt].v);
            v[1][zt].v=3.0f*v[5][zt].v-2.0f*v[7][zt].v;
            clip(v[1][zt].v);
        }
        // Right Boundary
        if (u[zt]<0.0f) {
            v[TX-4][zt].v=right_incoming;
            v[TX-2][zt].v=right_incoming;
        } else {
            v[TX-4][zt].v=2.0f*v[TX-6][zt].v- v[TX-8][zt].v;

```

```

    clip(v[TX-4][zt].v);
    v[TX-2][zt].v=3.0f*v[TX-6][zt].v-2.0f*v[TX-8][zt].v;
    clip(v[TX-2][zt].v);
}
}
// Upper Boundaries
for (int xt=3;xt<TX-3;xt+=2) {
    // Lower Boundary
    v[xt][3].v=0.0f;//2.0f*v[xt][5].v- v[xt][7].v;
    clip(v[xt][3].v);
    v[xt][1].v=0.0f;//3.0f*v[xt][5].v-2.0f*v[xt][7].v;
    clip(v[xt][1].v);
    // Upper Boundary
    v[xt][TZ-4].v=1.0f;//2.0f*v[xt][TZ-6].v- v[xt][TZ-8].v;
    clip(v[xt][TZ-4].v);
    v[xt][TZ-2].v=1.0f;//3.0f*v[xt][TZ-6].v-2.0f*v[xt][TZ-8].v;
    clip(v[xt][TZ-2].v);
}
}

// Timestep the Evens
{
    for (int xt=2;xt<TX-4;xt+=2) {
        for (int zt=2;zt<TZ-4;zt+=2) {
            float bx=limit(2.0f*(v[xt+1][zt+1].v-v[xt-1][zt+1].v),0.5f*(v[xt+3][zt+1].v-v[xt/
-1][zt+1].v),2.0f*(v[xt+3][zt+1].v-v[xt+1][zt+1].v));
            float bz=limit(2.0f*(v[xt+1][zt+1].v-v[xt+1][zt-1].v),0.5f*(v[xt+1][zt+3].v-v[xt/
+1][zt-1].v),2.0f*(v[xt+1][zt+3].v-v[xt+1][zt+1].v));
            float fdb=bx*u[zt+1];
            float gdb=bz*sr*(1.0f-2.0f*v[xt+1][zt+1].v);
            float bth=v[xt+1][zt+1].v-lambda*0.5f*fdb-mu*0.5f*gdb;

```

```

    if (xt>=3) {
        float ax=limit(2.0f*(v[xt-1][zt+1].v-v[xt-3][zt+1].v),0.5f*(v[xt+1][zt+1].v-v[
xt-3][zt+1].v),2.0f*(v[xt+1][zt+1].v-v[xt-1][zt+1].v));
        float az=limit(2.0f*(v[xt-1][zt+1].v-v[xt-1][zt-1].v),0.5f*(v[xt-1][zt+3].v-v[
xt-1][zt-1].v),2.0f*(v[xt-1][zt+3].v-v[xt-1][zt+1].v));
        float fda=ax*u[zt+1];
        float gda=az*Sr*(1.0f-2.0f*v[xt-1][zt+1].v);
        float ath=v[xt-1][zt+1].v-lambda*0.5f*fda-mu*0.5f*gda;
        v[xt][zt].N=0.0625f*(ax-bx)-lambda*0.5f*(bth*u[zt+1]-ath*u[zt+1]);
    }
    if (zt>=3) {
        float cx=limit(2.0f*(v[xt+1][zt-1].v-v[xt-1][zt-1].v),0.5f*(v[xt+3][zt-1].v-v[
xt-1][zt-1].v),2.0f*(v[xt+3][zt-1].v-v[xt+1][zt-1].v));
        float cz=limit(2.0f*(v[xt+1][zt-1].v-v[xt+1][zt-3].v),0.5f*(v[xt+1][zt+1].v-v[xt/
+1][zt-3].v),2.0f*(v[xt+1][zt+1].v-v[xt+1][zt-1].v));
        float fdc=cx*u[zt-1];
        float gdc=cz*Sr*(1.0f-2.0f*v[xt+1][zt-1].v);
        float cth=v[xt+1][zt-1].v-lambda*0.5f*fdc-mu*0.5f*gdc;
        v[xt][zt].E=0.0625f*(cz-bz)-mu*0.5f*(Sr*bth*(1.0f-bth)-Sr*cth*(1.0f-cth));
    }
}
}
}
// Update Evens based on Fluxes
for (int xt=4;xt<TX-4;xt+=2) {
    for (int zt=4;zt<TZ-4;zt+=2) {
        v[xt][zt].v=0.25f*(v[xt-1][zt-1].v+v[xt+1][zt-1].v+v[xt+1][zt+1].v+v[xt-1][zt+1]./
v);
        v[xt][zt].v+=v[xt][zt].N+v[xt][zt].E+v[xt-2][zt].E+v[xt][zt-2].N;
    }
}
}
}

```

```

if (!(count%15)) {
    //SDL_FillRect(screen, NULL, 0xFF);
    for (int xt=4;xt<TX-4;xt+=2) {
        int x=(xt/2)-2;
        for (int zt=4;zt<TZ-4;zt+=2) {
            int z=(zt/2)-2;
            unsigned char val=static_cast<unsigned char>(255.0f*(1.0f-(v[xt][zt].v<0.0f/
?0.0f:(v[xt][zt].v>1.0f?1.0f:v[xt][zt].v)))));
            //val=((val*16)+128)%256;
            unsigned int colour=SDL_MapRGB(image->format, val, val, val);
            if (doubled) {
                putpixel(image, x*2, (NZ-1-z)*2, colour);
                putpixel(image, x*2+1, (NZ-1-z)*2, colour);
                putpixel(image, x*2, (NZ-1-z)*2+1, colour);
                putpixel(image, x*2+1, (NZ-1-z)*2+1, colour);

            } else putpixel(image, x, NZ-1-z, colour);
        }
    }
    if (!(count%20)) {
        char title[40];
        float ratio=t/(0.00001f*(SDL_GetTicks()-firstTime));
        sprintf(title, "Time:_%5.1fs_(%4.1f%%_realtime)", t, ratio);
        SDL_WM_SetCaption(title, NULL);
    }
    SDL_BlitSurface(image, NULL, screen, NULL);
    SDL_BlitSurface(mask, NULL, screen, NULL);
    SDL_Flip(screen);
}
count++;

```

```
    t+=2.0f*dT;
}

SDL_SaveBMP(screen, "image.bmp");

SDL_FreeSurface(image);
SDL_FreeSurface(mask);


// Exit SDL
SDL_Quit();
return 0;
}
```

# Appendix B

## Particle on a string source code

### B.1 main.cc

```
#include <sstream>

#include <SDL.h>

#include <GL/gl.h>
#include <GL/glu.h>

float fillfraction=0.6f;

extern void triangle(float);

int main(int argc, char* argv[])
{
    if (SDL_Init (SDL_INIT_VIDEO) < 0) exit(1);

    if (argc==2) fillfraction=atof(argv[1]);
```



```
const int width=512;
const int height=512;

SDL_GL_SetAttribute( SDL_GL_RED_SIZE, 5 );
SDL_GL_SetAttribute( SDL_GL_GREEN_SIZE, 5 );
SDL_GL_SetAttribute( SDL_GL_BLUE_SIZE, 5 );
SDL_GL_SetAttribute( SDL_GL_DEPTH_SIZE, 16 );
SDL_GL_SetAttribute( SDL_GL_DOUBLEBUFFER, 1 );

SDL_SetVideoMode(width, height, 0, SDL_OPENGL | SDL_HWSURFACE );
glViewport(0, 0, width, height);

glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClearDepth(1.0f);

glDepthFunc(GL_LEQUAL);
glEnable(GL_DEPTH_TEST);

glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glOrtho(-1.0f, 1.0f, -1.0f, 1.0f, -1.0f, 1.0f);
glMatrixMode(GL_MODELVIEW);

double angle=0.0f;

bool done=false;
do {
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    SDL_Event event;
```

```
while (SDL_PollEvent (&event))
{
    switch (event.type)
    {
        case SDL_KEYDOWN:
            if(event.key.keysym.sym==SDLK_ESCAPE)
                done = true;
            break;
        case SDL_QUIT:
            done = true;
            break;
        default:
            break;
    }
}
angle+=0.01f;
triangle(angle);
SDL_GL_SwapBuffers();
} while(!done);
return 0;
}
```

## B.2 triangle.cc

```
#include <cmath>

#include <iostream>

#include <vector>
#include <list>
```

```

#include <GL/gl.h>

#define F_PI 3.1415926535897932384626433832795f

extern float fillfraction;

struct cPoint
{
    float x,y;
    cPoint(float tX, float tY) : x(tX), y(tY) {}
    float normalize() {
        float length=sqrtf(x*x+y*y);
        x/=length;
        y/=length;
        return length;
    }
};

std::list<cPoint> mInterface;

void triangle(float angle)
{
    std::cout<<fillfraction<<std::endl;
    std::list<float> mIntersections;
    const float L = 1.5f; // Side length

    // Plot the border
    float thetaF= F_PI/2.0f;
    cPoint p1((L/sqrtf(3.0f))*cosf(thetaF), (L/sqrtf(3.0f))*sinf(thetaF));
    cPoint p2((L/sqrtf(3.0f))*cosf(thetaF+2.0f*F_PI/3.0f), (L/sqrtf(3.0f))*sinf(thetaF+2.0f*/
    F_PI/3.0f));

```

```
cPoint p3((L/sqrtf(3.0f))*cosf(thetaF+4.0f*F_PI/3.0f), (L/sqrtf(3.0f))*sinf(thetaF+4.0f*F_PI/3.0f));
```

```
glBegin(GL_LINE_LOOP);
    glColor3f(0.0f, 0.0f, 0.0f);
    glVertex2f(p1.x, p1.y);
    glVertex2f(p2.x, p2.y);
    glVertex2f(p3.x, p3.y);
glEnd();
```

```
float A_triangle = sqrtf(3.0f)*L*L/4.0f;
float A_grain = fillfraction*A_triangle;
float A_air = (1.0f-fillfraction)*A_triangle;
```

```
float beta0 = atanf(sqrtf(3.0f)*(1.0f-fillfraction)/(1.0f+fillfraction));
float theta0 = F_PI/6.0f-beta0;
float beta1 = atanf(sqrtf(3.0f)*fillfraction/(2.0f-fillfraction));
float theta1 = F_PI/2.0f-beta1;
float theta2 = 5.0f*F_PI/6.0f-beta0;
```

```
float beta, alpha, a, b, c, d=0.0f, rA=0.0f, rB=0.0f;
```

```
float thetaD=fmodf(angle,(2.0f*F_PI/3.0f));
if (0.0f<=thetaD && thetaD<=theta0) {//low angle air triangle
    beta = F_PI/6.0f-thetaD;
    alpha = 2.0f*F_PI/3.0f-beta;
    c = sqrtf(sqrtf(3.0f)*A_air/(sinf(alpha)*sinf(beta)));
    a = 2.0f/sqrtf(3.0f)*c*sinf(alpha);
    b = 2.0f/sqrtf(3.0f)*c*sinf(beta);
    rA = L/sqrtf(3.0f)*cosf(thetaD+2.0f*F_PI/3.0f)-b*cosf(alpha);
```

```

rB = L/sqrtf(3.0f)*cosf(thetaD+2.0f*F_PI/3.0f)+a*cosf(beta);
d = L/sqrtf(3.0f)*sinf(thetaD+2.0f*F_PI/3.0f)-a*sinf(beta);
}
else {
    if (thetaD <= theta1) {// low angle grain triangle
        beta = F_PI/2.0f-thetaD;
        alpha = 2.0f*F_PI/3.0f-beta;
        c = sqrtf(sqrtf(3.0f)*A_grain/(sinf(alpha)*sinf(beta)));
        a = 2.0f/sqrtf(3.0f)*c*sinf(alpha);
        b = 2.0f/sqrtf(3.0f)*c*sinf(beta);
        rA = L/sqrtf(3.0f)*cosf(thetaD+4.0f*F_PI/3.0f)-a*cosf(beta);
        rB = L/sqrtf(3.0f)*cosf(thetaD+4.0f*F_PI/3.0f)+b*cosf(alpha);
        d = L/sqrtf(3.0f)*sinf(thetaD+4.0f*F_PI/3.0f)+a*sinf(beta);
    }
    else {
        if (thetaD <= theta2) {// high angle air triangle
            beta = 5.0f*F_PI/6.0f-thetaD;
            alpha = 2.0f*F_PI/3.0f-beta;
            c = sqrtf(sqrtf(3.0f)*A_air/(sinf(alpha)*sinf(beta)));
            a = 2.0f/sqrtf(3.0f)*c*sinf(alpha);
            b = 2.0f/sqrtf(3.0f)*c*sinf(beta);
            rA = L/sqrtf(3.0f)*cosf(thetaD)-b*cosf(alpha);
            rB = L/sqrtf(3.0f)*cosf(thetaD)+a*cosf(beta);
            d = L/sqrtf(3.0f)*sinf(thetaD)-a*sinf(beta);
        }
        else {
            if (thetaD <= 2.0f*F_PI/3.0f) {// high angle grain triangle
                beta = 7.0f*F_PI/6.0f-thetaD;
                alpha = 2.0f*F_PI/3.0f-beta;
                c = sqrtf(sqrtf(3.0f)*A_grain/(sinf(alpha)*sinf(beta)));
                a = 2.0f/sqrtf(3.0f)*c*sinf(alpha);

```

```

        b = 2.0f/sqrtf(3.0f)*c*sinf(beta);
        rA = L/sqrtf(3.0f)*cosf(thetaD+2.0f*F_PI/3.0f)-a*cosf(beta);
        rB = L/sqrtf(3.0f)*cosf(thetaD+2.0f*F_PI/3.0f)+b*cosf(alpha);
        d = L/sqrtf(3.0f)*sinf(thetaD+2.0f*F_PI/3.0f)+a*sinf(beta);
    }
}
}

//Draw rotated surface
angle=angle-F_PI/2.0f;

cPoint end1(rA*cosf(angle)+d*sinf(angle), -rA*sinf(angle)+d*cosf(angle));
cPoint end2(rB*cosf(angle)+d*sinf(angle), -rB*sinf(angle)+d*cosf(angle));
cPoint normal(end1.y-end2.y, end2.x-end1.x);
cPoint middle((end1.x+end2.x)*0.5f, (end1.y+end2.y)*0.5f);
cPoint direction(end2.x-end1.x, end2.y-end1.y);
float length=direction.normalize();

// Generate the interface
if (mInterface.size()==0) {
    const float pointCount=200;
    for (int i=0;i<pointCount;i++) mInterface.push_back(cPoint(L*0.25f*cosf(-F_PI/
*2.0f*static_cast<float>(i)/(static_cast<float>(pointCount))-F_PI*0.5f), L*0.25f*
sinf(-F_PI*2.0f*static_cast<float>(i)/(static_cast<float>(pointCount))-F_PI*0.5f))/
);
    while (true)
    {
        if ((mInterface.front().x-end1.x)*normal.x+(mInterface.front().y-end1.y)*
normal.y>=0.0f) {
            if ((mInterface.back().x-end1.x)*normal.x+(mInterface.back().y-end1.y)*
normal.y<0.0f) break;

```



```

    }
    mInterface.push_back(mInterface.front());
    mInterface.pop_front();
}
mInterface.push_back(mInterface.front());
mInterface.pop_front();
}

// Plot the Surface
glColor3f(0.7f, 0.7f, 0.7f);
glBegin(GL_LINES);
    glVertex2f(end1.x, end1.y);
    glVertex2f(end2.x, end2.y);
glEnd();

// Clear points beyond the end
while ((mInterface.front().x-end1.x)*normal.x+(mInterface.front().y-end1.y)*normal.y/
<=0.0f) {
    mInterface.pop_front();
}

// Find the intersection
mIntersections.push_back(0.0f);
for (std::list<cPoint>::iterator iter=mInterface.begin();iter!=mInterface.end();iter++)
{
    if ((iter->x-end1.x)*normal.x+(iter->y-end1.y)*normal.y>0.0f) {
        std::list<cPoint>::iterator iter2=iter;
        iter2++;
        if ((iter2->x-end1.x)*normal.x+(iter2->y-end1.y)*normal.y<=0.0f) {
            float t1 = iter->x*end1.y;
            float t3 = iter2->y*end1.x;

```

```

float t5 = iter->y*end1.x;
float t6 = iter2->x*end1.y;
float t7 = (iter2->y*end2.x-t3-iter->y*end2.x+t5-iter2->x*end2.y+t6/
+iter->x*end2.y-t1);
if (fabsf(t7)>0.00001f) {
    float t = 1.0f+(t1+iter2->x*iter->y+t3-iter2->y*iter->x-t5-t6)//
t7;

    if (t<=0.5f && t>=0.0f) mIntersections.push_back(t);
}
else {
    float dist=(iter->x-end1.x)*normal.x+(iter->y-end1.y)*normal.y;
    if (fabsf(dist)<0.00001f) {
        float t=((iter->x-end1.x)*direction.x+(iter->y-end1.y)*direction/
.y)/length;

        float t2=((iter2->x-end1.x)*direction.x+(iter2->y-end1.y)*
direction.y)/length;

        std::cout<<"Clause_A:_"<<t<<"_"<<t2<<std::endl;
        if (t>0.0f && t2>0.0f) {
            if (t<0.5f && t2<0.5f) {
                if (t>t2) mIntersections.push_back(t);
                else mIntersections.push_back(t2);
            }
            else {
                if (t>0.5f) {
                    if (t2>0.0f && t2<0.5f) mIntersections.push_back(t2);
                    if (t2<0.0f) mIntersections.push_back(0.0f);
                }
                else {
                    if (t>0.0f && t<0.5f) mIntersections.push_back(t);
                    if (t<0.0f) mIntersections.push_back(0.0f);
                }
            }
        }
    }
}

```

```

    }
}
else {
    if (t<0.0f) {
        if (t2>0.0f && t2<0.5f) mIntersections.push_back(t2);
        if (t2>0.5f) mIntersections.push_back(0.5f);
    }
    else {
        if (t>0.0f && t<0.5f) mIntersections.push_back(t);
        if (t>0.5f) mIntersections.push_back(0.5f);
    }
}
}
}
}
}
else {
    std::list<cPoint>::iterator iter2=iter;
    iter2++;
    if ((iter2->x-end1.x)*normal.x+(iter2->y-end1.y)*normal.y>0.0f) {
        float t1 = iter->x*end1.y;
        float t3 = iter2->y*end1.x;
        float t5 = iter->y*end1.x;
        float t6 = iter2->x*end1.y;
        float t7 = (iter2->y*end2.x-t3-iter->y*end2.x+t5-iter2->x*end2.y+t6/
+iter->x*end2.y-t1);
        if (fabsf(t7)>0.00001f) {
            float t = 1.0f+(t1+iter2->x*iter->y+t3-iter2->y*iter->x-t5-t6)//
t7;
            if (t<=0.5f && t>=0.0f) mIntersections.push_back(t);
        }
    }
}

```

```

else {
    float dist=(iter->x-end1.x)*normal.x+(iter->y-end1.y)*normal.y;
    if (fabsf(dist)<0.00001f) {
        float t=((iter->x-end1.x)*direction.x+(iter->y-end1.y)*direction/
.y)/length;

        float t2=((iter2->x-end1.x)*direction.x+(iter2->y-end1.y)*
direction.y)/length;

        std::cout<<"Clause_B:_"<<t<<"_"<<t2<<std::endl;
        if (t>0.0f && t2>0.0f) {
            if (t<0.5f && t2<0.5f) {
                if (t>t2) mIntersections.push_back(t);
                else mIntersections.push_back(t2);
            }
            else {
                if (t>0.5f) {
                    if (t2>0.0f && t2<0.5f) mIntersections.push_back(t2);
                    if (t2<0.0f) mIntersections.push_back(0.0f);
                }
                else {
                    if (t>0.0f && t<0.5f) mIntersections.push_back(t);
                    if (t<0.0f) mIntersections.push_back(0.0f);
                }
            }
        }
        else {
            if (t<0.0f) {
                if (t2>0.0f && t2<0.5f) mIntersections.push_back(t2);
                if (t2>0.5f) mIntersections.push_back(0.5f);
            }
            else {
                if (t>0.0f && t<0.5f) mIntersections.push_back(t);

```

```

        if (t>0.5f) mIntersections.push_back(0.5f);
    }
}
}
}
}
}
}
}
mIntersections.push_back(0.5f);
float sum=0.0f;
int parity=0;
mIntersections.sort();
for (std::list<float>::iterator iter=mIntersections.begin();(*iter)!=0.5f;iter++)
{
    std::cout<<*iter<<" ";
    std::list<float>::iterator iter2=iter;
    ++iter2;
    if (fabsf(*iter-*iter2)>0.00001f) ++parity;
    float& a=*iter;
    float& b=*iter2;
    if (parity%2) sum+=0.5f*b-0.5f*a-0.5f*b*b+0.5f*a*a;
}

std::cout<<std::endl;

if (sum>0.125f) sum=0.125f;
if (sum<0.0f) sum=0.0f;

{
    float t=0.5f-0.5f*sqrtf((1.0f-8.0f*sum));

```

```
        mInterface.push_back(cPoint((end2.x-end1.x)*t+end1.x, (end2.y-end1.y)*t+end1.y)/
    );
}

// Plot the interface
glColor3f(0.0f, 1.0f, 0.0f);
glBegin(GL_LINE_STRIP);
    for (std::list<cPoint>::iterator iter=mInterface.begin();iter!=mInterface.end();iter/
    +++) glVertex2f(iter->x, iter->y);
glEnd();
}
```



# Appendix C

## Volumetric method source code

### C.1 main.cc

```
#include <ctime>
```

```
#include <cstdlib>
```

```
#include <iostream>
```

```
#include <sstream>
```

```
#include <SDL.h>
```

```
#include "../config/config.h"
```

```
#include "cdrom.h"
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    srand(time(0));
```

```
double fillFraction=0.75, particleRatio=0.5;
if (argc>3) {
    std::cout<<"Extra arguments detected and ignored"<<std::endl;
}
if (argc>2) {
    {
        std::stringstream input(argv[1]);
        input>>fillFraction;
    }
    if (argc==3) {
        std::stringstream input(argv[2]);
        input>>particleRatio;
    }
}

if (SDL_Init (SDL_INIT_VIDEO) < 0) exit (1);
atexit (SDL_Quit);

SDL_Surface* screen = SDL_SetVideoMode(725, 725, 32, SDL_DOUBLEBUF | /
SDL_SWSURFACE);
SDL_WM_SetCaption("Rotating Drum Simulation", NULL);
CDrum drum(fillFraction, particleRatio, screen);

bool done=false;
do {
    drum.update();
    if (drum.render(screen)) SDL_Flip(screen);
    SDL_Event event;
    while (SDL_PollEvent (&event))
    {
        switch (event.type)
```

```
{
case SDL_KEYDOWN:
    switch (event.key.keysym.sym) {
        case SDLK_q:
        case SDLK_ESCAPE:
            done = true;
            break;
        default:
            break;
    }
    break;
case SDL_QUIT:
    done = true;
    break;
default:
    break;
}
} while(!done);
return 0;
}
```

## C.2 cdrum.h

```
#ifndef CDRUM_HXX
#define CDRUM_HXX

#include <vector>
#include <string>

#include <SDL.h>
```

```
#include "../config/config.h"
#include "csurface.h"
#include "cvertex.h"

class CDrum
{
public:
    CSurface surface;
    double gamma;
    double theta;
    double ratio;
    SDL_Surface* image;
    SDL_Surface* backbuffer;
    SDL_Surface* mask;

    CDrum(double, double, SDL_Surface*);
    ~CDrum();
    bool render(SDL_Surface*);
    void update();
};

#endif //CDRUM_HXX
```

### C.3 cdrum.cc

```
#include <iostream>
#include <fstream>
#include <sstream>
#include <iomanip>
#include <string>
```

```
#include <vector>

#include <cstdlib>
#include <cmath>

#include <SDL.h>

#include "SDL_rotozoom.h"

#include "../config/config.h"
#include "drawing.h"
#include "cdrum.h"
#include "cvertex.h"

const double pi=acos(-1.0);

double correction=1.0;
bool particle1Found=false;
//#define OUTPUT
//#define MASK

#ifdef OUTPUT
SDL_Surface* hidden;
#endif

extern std::ofstream coordinates;
extern std::ofstream distances;

CDrum::CDrum(double fillFraction, double particleRatio, SDL_Surface* screen) : surface(/
    fillFraction), gamma(pi*0.5), ratio(particleRatio) // REMOVE 2pi
{
```

```

{
    int imageSize=screen->w>screen->h?screen->h:screen->w;
    image=SDL_CreateRGBSurface(screen->flags, (int)(imageSize/
*0.70710678118654752440084436210485), (int)(imageSize/
*0.70710678118654752440084436210485), screen->format->BitsPerPixel, screen->/
format->Rmask, screen->format->Gmask, screen->format->Bmask, screen->/
format->Amask);

    double d=particleRatio>1.0?1.0:particleRatio<0.0?0.0:particleRatio;
    unsigned char colr=(unsigned char) (150.0+d*105.0);
    unsigned char colg=(unsigned char) (55.0+d*155.0);
    unsigned char colb=(unsigned char) (d*255.0);
    colr=colr>254?254:colr<1?1:colr;
    colg=colg>254?254:colg<1?1:colg;
    colb=colb>254?254:colb<1?1:colb;
    SDL_FillRect(image, NULL, SDL_MapRGB(image->format, colr, colg, colb));
    // Add noise to initial material
    for (int y=0;y<image->h;++y) {
        SDL_Rect pixel;
        for (int x=0;x<image->w;++x) {
            pixel.x=x;
            pixel.y=y;
            pixel.w=1;
            pixel.h=1;
            double d_=d+0.3*static_cast<double>(rand())/static_cast<double>(/
RAND_MAX)-0.15;
            if (d_<0.0) d_=0.0;
            if (d_>1.0) d_=1.0;
            unsigned char colr=(unsigned char) (150.0+d_*105.0);
            unsigned char colg=(unsigned char) (55.0+d_*155.0);
            unsigned char colb=(unsigned char) (d_*255.0);

```

```

        colr=colr>254?254:colr<1?1:colr;
        colg=colg>254?254:colg<1?1:colg;
        colb=colb>254?254:colb<1?1:colb;
        SDL_FillRect(image, &pixel, SDL_MapRGB(image->format, colr, colg, colb));
    }
}

#ifdef OUTPUT
hidden=SDL_CreateRGBSurface(screen->flags, screen->w, screen->h, screen->/
format->BitsPerPixel, screen->format->Rmask, screen->format->Gmask, screen-/
->format->Bmask, screen->format->Amask);
#endif

mask=SDL_CreateRGBSurface(image->flags, image->w, image->h, image->format/
->BitsPerPixel, image->format->Rmask, image->format->Gmask, image->format/
->Bmask, image->format->Amask);
backbuffer=SDL_CreateRGBSurface(image->flags, image->w, image->h, image->/
format->BitsPerPixel, image->format->Rmask, image->format->Gmask, image->/
format->Bmask, image->format->Amask);

//std::cout<<"CDrum message:\t\tMask Generated"<<std::endl;
// {
// SDL_Surface* tempmask=SDL_LoadBMP("initial.bmp");
// SDL_BlitSurface(tempmask, NULL, mask, NULL);
// SDL_FreeSurface(tempmask);
// }

SDL_SetColorKey(mask, SDL_SRCCOLORKEY|SDL_RLEACCEL, 0x01);
{
    #ifdef MASK

```



```

    SDL_Surface* tempMask=SDL_LoadBMP("mask.bmp");
    SDL_BlitSurface(tempMask, NULL, mask, NULL);
    SDL_FreeSurface(tempMask);
    #else
    SDL_FillRect(mask, NULL, 0x01);//SDL_MapRGB(mask->format, 240, 240, 240));
    #endif
}

theta=0.99*asin(2.0/static_cast<double>(image->w<image->h?image->w:image/
->h));
theta*=1.0; // USE A LARGER ANGULAR INCREMENT
}

CDrum::~CDrum()
{
    SDL_FreeSurface(backbuffer);
    SDL_FreeSurface(mask);
    SDL_FreeSurface(image);
    #ifdef OUTPUT
    SDL_FreeSurface(hidden);
    #endif
}

bool CDrum::render(SDL_Surface* screen)
{
    bool rendered=false;
    #ifdef OUTPUT
    static int count=0;
    #endif
    if ((!particle1Found)&&(gamma>2.0943951023931954923084289221863)) particle1Found/
    =true;

```

```

surface.render(image, mask);

float surfaceAngle=0.4;
#ifdef OUTPUT
if (fmod(gamma, pi/699.0)<theta) {
#else
if (fmod(gamma, pi*0.1122)<theta) {
#endif
    #ifdef OUTPUT
    SDL_FillRect(hidden, NULL, SDL_MapRGB(hidden->format, 240, 240, 240));
    #else
    SDL_FillRect(screen, NULL, SDL_MapRGB(screen->format, 240, 240, 240));
    #endif
    SDL_BlitSurface(image, NULL, backbuffer, NULL);
    SDL_BlitSurface(mask, NULL, backbuffer, NULL);
    #ifdef OUTPUT
    transformSurfaceRGBA(backbuffer, hidden, screen->w/2, screen->h/2, -(int)(sin(/
gamma-(pi*0.5-surfaceAngle))*65536.0), (int)(cos(gamma-(pi*0.5-surfaceAngle))/
*65536.0), SMOOTHING_ON);
    #else
    transformSurfaceRGBA(backbuffer, screen, screen->w/2, screen->h/2, -(int)(sin(/
gamma-(pi*0.5-surfaceAngle))*65536.0), (int)(cos(gamma-(pi*0.5-surfaceAngle))/
*65536.0), SMOOTHING_ON);
    #endif
    rendered=true;
    #ifdef OUTPUT
    {
        if (!(count%50)) SDL_BlitSurface(hidden, NULL, screen, NULL);
        if (!(count%1)) {
            std::stringstream filename;

```

```

        filename<<"../images/"<<std::setfill('0')<<std::setw(5)<<count<<".bmp";
        SDL_SaveBMP(hidden, filename.str().c_str());

    }
    count++;
}
if (count>7000) exit(0);
#endif
}
if (fmod(gamma-0.5*pi, pi*2.0)<theta) {
    coordinates<<"+"<<std::endl;
}
if (gamma-0.5*pi>10.0*pi) {
    SDL_SaveBMP(backbuffer, "../test.bmp");
    exit(0);
}
if ((gamma>0.4*pi)&&(fmod(gamma, pi*0.0625)<theta)) {
    int count=0;
    int cellcount=0;
    if (SDL_MUSTLOCK(image)) SDL_LockSurface(image);
        if (SDL_MUSTLOCK(mask)) SDL_LockSurface(mask);
            for(int y=0;y<image->h;y++) {
                for (int x=0;x<image->w;x++) {
                    if (getpixel(mask, x, y)==0x01) {
                        count+=getpixel(image, x, y)&0xFF;
                        cellcount++;
                    }
                }
            }
        if (SDL_MUSTLOCK(image)) SDL_UnlockSurface(image);
    if (SDL_MUSTLOCK(mask)) SDL_UnlockSurface(mask);
}

```

```
        correction=correction*0.9+0.1*ratio/(pi*0.00125*((double)count)/((double)cellcount)/
    );
}

    return rendered;
}

void CDrum::update()
{
    surface.update(gamma);
    gamma+=theta;
}
```

## C.4 csurface.h

```
#ifndef CSURFACE_HXX
#define CSURFACE_HXX

#include <vector>
#include <fstream>

#include <SDL.h>

#include "../config/config.h"
#include "cvector.h"
#include "cvertex.h"

class CSurface
{
public:
    CVertex A, B, C, D, E;
```

```

    CVector normal;
    CVertex location;
    const double fillFraction;
    std::vector<std::pair<std::vector<CVertex>::const_iterator, double> > sortedCorners;

    void update(double);
    void render(SDL_Surface*, SDL_Surface*);
    CSurface(const double&);
    ~CSurface();
};

#ifdef CSURFACE_HXX

```

## C.5 csurface.cc

```

#include <iostream>
#include <fstream>
#include <iomanip>
#include <cmath>
#include <cfloat>

#include <SDL.h>

#include "../config/config.h"
#include "csurface.h"
#include "drawing.h"

extern double correction;
extern bool particle1Found;
const double pi=acos(-1.0);
bool particle2Found=false;

```

```
std::ofstream coordinates("../data/coords.txt", std::ios::trunc), distances("../data/dists.txt", /
    std::ios::trunc), particle("../data/part.txt", std::ios::trunc);
```

```
struct vertex
```

```
{
    double x;
    double y;
};
```

```
CSurface::CSurface(const double& tFillFraction) : fillFraction(tFillFraction)
```

```
{
    std::cout<<std::setprecision(18)<<std::fixed;
    coordinates<<std::setprecision(18)<<std::fixed;
    distances<<std::setprecision(18)<<std::fixed;
    particle<<std::setprecision(18)<<std::fixed;
}
```

```
CSurface::~CSurface()
```

```
{
    distances.close();
    coordinates.close();
}
```

```
void CSurface::render(SDL_Surface* image, SDL_Surface* mask)
```

```
{
    static bool trackParticle1=false;
    static bool trackParticle2=false;
    static double radius=image->h<image->w?image->h*0.497:image->w*0.497;
    static double cx=image->w*0.5;
    static double cy=image->h*0.5;//+correctionFactor;
```

```
static vertex part1={DBL_MAX,DBL_MAX};
static vertex part2={DBL_MAX,DBL_MAX};

double x0=B.x*radius+cx;
double x1=A.x*radius+cx;
double x2=C.x*radius+cx;
double x3=D.x*radius+cx;
double x4=E.x*radius+cx;
double y0=cy-B.y*radius;
double y1=cy-A.y*radius;
double y2=cy-C.y*radius;
double y3=cy-D.y*radius;
double y4=cy-E.y*radius;

int* yBufferLeft=new int[image->h];
int* yBufferRight=new int[image->h];

for(int y=0;y<image->h;y++) {
    yBufferLeft[y]=0;
    yBufferRight[y]=image->w-1;
}
int output[2]={0};
double triple[3][2]={{x1, y1}, {x2, y2}, {x0, y0}};

scanConvertTriangle( triple, yBufferLeft, yBufferRight, output );
vertex tri[3]={{x4,y4},{x3,y3},{x0,y0}};

vertex rect[2];
double areaSum=0.0;
```



```

    double trueArea=255.0*0.5*(tri[0].x*(tri[2].y-tri[1].y)+tri[1].x*(tri[0].y-tri[2].y)+tri/
[2].x*(tri[1].y-tri[0].y));
    rect[0].x=DBL_MAX;
    rect[0].y=DBL_MAX;
    rect[1].x=-DBL_MAX;
    rect[1].y=-DBL_MAX;
    for (int t=0;t<3;t++) {
        if (tri[t].x<rect[0].x) rect[0].x=tri[t].x;
        if (tri[t].y<rect[0].y) rect[0].y=tri[t].y;
        if (tri[t].x>rect[1].x) rect[1].x=tri[t].x;
        if (tri[t].y>rect[1].y) rect[1].y=tri[t].y;
    }
    if (SDL_MUSTLOCK(image)) SDL_LockSurface(image);
    for (int y=(int)floorf(rect[0].y);y<(int)ceilf(rect[1].y);y++) {
        for (int x=(int)floorf(rect[0].x);x<(int)ceilf(rect[1].x);x++) {
            bool corners[3]={false, false, false};
            int cornercount=0;
            for (int t=0;t<3;t++) {
                if (tri[t].x>=x&&tri[t].y>=y&&tri[t].x<x+1&&tri[t].y<y+1) {
                    corners[t]=true;
                    cornercount++;
                }
            }
            if (cornercount>0) {
                switch (cornercount) {
                    case 1:
                        {
                            vertex polygon[7];
                            int polyVerts=0;
                            int t=0;
                            for (;t<3;t++) if (corners[t]) break;

```

```

vertex edge1[2]={tri[t], tri[(t+1)%3]};
vertex edge2[2]={tri[t], tri[(t+2)%3]};
polygon[0]=edge1[0]; polyVerts++;
int counter=0;
for (;counter<4;counter++) {
    if (counter==0) {
        double X=x+1;
        double Y=y+1;
        double denom=edge1[1].x-edge1[0].x;
        if (denom!=0.0) {
            double T=(X-edge1[0].x)/denom;
            if (T>=0.0&&T<=1.0) {
                double S=Y-T*(edge1[1].y-edge1[0].y)-/
edge1[0].y;

                if (S>=0.0&&S<=1.0) {
                    polygon[polyVerts].x=X;
                    polygon[polyVerts].y=Y-S;
                    polyVerts++;
                    break;
                }
            }
        }
    }
    if (counter==1) {
        double X=x+1;
        double Y=y;
        double denom=edge1[1].y-edge1[0].y;
        if (denom!=0.0) {
            double T=(Y-edge1[0].y)/denom;
            if (T>=0.0&&T<=1.0) {

```

```

double S=X-T*(edge1[1].x-edge1[0].x)-/
edge1[0].x;

    if (S>=0.0&&S<=1.0) {
        polygon[polyVerts].x=X-S;
        polygon[polyVerts].y=Y;
        polyVerts++;
        break;
    }
}
}
if (counter==2) {
    double X=x;
    double Y=y;
    double denom=edge1[1].x-edge1[0].x;
    if (denom!=0.0) {
        double T=(X-edge1[0].x)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=edge1[0].y-Y+T*(edge1[1].y-/
edge1[0].y);

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y+S;
                polyVerts++;
                break;
            }
        }
    }
}
if (counter==3) {
    double X=x;

```

```

    double Y=y+1;
    double denom=edge1[1].y-edge1[0].y;
    if (denom!=0.0) {
        double T=(Y-edge1[0].y)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=edge1[0].x-X+T*(edge1[1].x-
edge1[0].x);

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X+S;
                polygon[polyVerts].y=Y;
                polyVerts++;
                break;
            }
        }
    }
}
vertex edge[2]={edge1[1],edge2[1]};
for (int tempCount=0;tempCount<4;tempCount++) {
    switch (counter) {
        case 0:
            {
                double X=x+1;
                double Y=y+1;
                vertex edgeDir={edge[1].y-edge[0].y,edge[0].x-
edge[1].x};

                if (edgeDir.x*(X-edge[0].x)+edgeDir.y*(Y-
edge[0].y)>0.0) {

                    edgeDir.x=edge1[1].y-edge1[0].y,
                    edgeDir.y=edge1[0].x-edge1[1].x;

```

```

        if (edgeDir.x*(X-edge1[0].x)+edgeDir.y*(Y/
-edge1[0].y)>0.0) {

            edgeDir.x=edge2[1].y-edge2[0].y,
            edgeDir.y=edge2[0].x-edge2[1].x;
            if (edgeDir.x*(X-edge2[0].x)+edgeDir.y/
*(Y-edge2[0].y)<0.0) {

                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y;
                polyVerts++;
            }
        }
    }
    double denom=edge[1].x-edge[0].x;
    if (denom!=0.0) {
        double T=(X-edge[0].x)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=Y-T*(edge[1].y-edge[0].y)-/
edge[0].y;

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y-S;
                polyVerts++;
            }
        }
    }
    break;
case 1:
    {
        double X=x+1;
        double Y=y;

```

```

vertex edgeDir={edge[1].y-edge[0].y,edge[0].x-/
edge[1].x};

edge[0].y>0.0) {

    edgeDir.x=edge1[1].y-edge1[0].y,
    edgeDir.y=edge1[0].x-edge1[1].x;
    if (edgeDir.x*(X-edge1[0].x)+edgeDir.y*(Y-
-edge1[0].y)>0.0) {

        edgeDir.x=edge2[1].y-edge2[0].y,
        edgeDir.y=edge2[0].x-edge2[1].x;
        if (edgeDir.x*(X-edge2[0].x)+edgeDir.y/
*(Y-edge2[0].y)<0.0) {

            polygon[polyVerts].x=X;
            polygon[polyVerts].y=Y;
            polyVerts++;
        }
    }
}

double denom=edge[1].y-edge[0].y;
if (denom!=0.0) {
    double T=(Y-edge[0].y)/denom;
    if (T>=0.0&&T<=1.0) {
        double S=X-T*(edge[1].x-edge[0].x)-/
edge[0].x;

        if (S>=0.0&&S<=1.0) {
            polygon[polyVerts].x=X-S;
            polygon[polyVerts].y=Y;
            polyVerts++;
        }
    }
}

```

```

    }
    break;
case 2:
    {
        double X=x;
        double Y=y;
        vertex edgeDir={edge[1].y-edge[0].y,edge[0].x-
edge[1].x};

        if (edgeDir.x*(X-edge[0].x)+edgeDir.y*(Y-
edge[0].y)>0.0) {

            edgeDir.x=edge1[1].y-edge1[0].y,
            edgeDir.y=edge1[0].x-edge1[1].x;
            if (edgeDir.x*(X-edge1[0].x)+edgeDir.y*(Y-
-edge1[0].y)>0.0) {

                edgeDir.x=edge2[1].y-edge2[0].y,
                edgeDir.y=edge2[0].x-edge2[1].x;
                if (edgeDir.x*(X-edge2[0].x)+edgeDir.y/
*(Y-edge2[0].y)<0.0) {

                    polygon[polyVerts].x=X;
                    polygon[polyVerts].y=Y;
                    polyVerts++;
                }
            }
        }
        double denom=edge[1].x-edge[0].x;
        if (denom!=0.0) {
            double T=(X-edge[0].x)/denom;
            if (T>=0.0&&T<=1.0) {
                double S=edge[0].y-Y+T*(edge[1].y-
edge[0].y);

                if (S>=0.0&&S<=1.0) {

```



```

        polygon[polyVerts].x=X;
        polygon[polyVerts].y=Y+S;
        polyVerts++;
    }
}
}
}
break;
case 3:
{
    double X=x;
    double Y=y+1;
    vertex edgeDir={edge[1].y-edge[0].y,edge[0].x-/
edge[1].x};

    if (edgeDir.x*(X-edge[0].x)+edgeDir.y*(Y-/
edge[0].y)>0.0) {

        edgeDir.x=edge1[1].y-edge1[0].y,
        edgeDir.y=edge1[0].x-edge1[1].x;
        if (edgeDir.x*(X-edge1[0].x)+edgeDir.y*(Y/
-edge1[0].y)>0.0) {

            edgeDir.x=edge2[1].y-edge2[0].y,
            edgeDir.y=edge2[0].x-edge2[1].x;
            if (edgeDir.x*(X-edge2[0].x)+edgeDir.y/
*(Y-edge2[0].y)<0.0) {

                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y;
                polyVerts++;
            }
        }
    }
}
double denom=edge[1].y-edge[0].y;

```

```

    if (denom!=0.0) {
        double T=(Y-edge[0].y)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=edge[0].x-X+T*(edge[1].x-
edge[0].x);

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X+S;
                polygon[polyVerts].y=Y;
                polyVerts++;
            }
        }
    }
    counter++;
    counter=counter%4;
}
for (counter=0;counter<4;counter++) {
    if (counter==0) {
        double X=x+1;
        double Y=y+1;
        double denom=edge2[1].x-edge2[0].x;
        if (denom!=0.0) {
            double T=(X-edge2[0].x)/denom;
            if (T>=0.0&&T<=1.0) {
                double S=Y-T*(edge2[1].y-edge2[0].y)-/
edge2[0].y;

                if (S>=0.0&&S<=1.0) {
                    polygon[polyVerts].x=X;
                    polygon[polyVerts].y=Y-S;
                    polyVerts++;

```

```

        break;
    }
}
}
}
if (counter==1) {
    double X=x+1;
    double Y=y;
    double denom=edge2[1].y-edge2[0].y;
    if (denom!=0.0) {
        double T=(Y-edge2[0].y)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=X-T*(edge2[1].x-edge2[0].x)-/
edge2[0].x;

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X-S;
                polygon[polyVerts].y=Y;
                polyVerts++;
                break;
            }
        }
    }
}
if (counter==2) {
    double X=x;
    double Y=y;
    double denom=edge2[1].x-edge2[0].x;
    if (denom!=0.0) {
        double T=(X-edge2[0].x)/denom;
        if (T>=0.0&&T<=1.0) {

```

```

    edge2[0].y);

    double S=edge2[0].y-Y+T*(edge2[1].y-/

    if (S>=0.0&&S<=1.0) {
        polygon[polyVerts].x=X;
        polygon[polyVerts].y=Y+S;
        polyVerts++;
        break;
    }
}

}

if (counter==3) {
    double X=x;
    double Y=y+1;
    double denom=edge2[1].y-edge2[0].y;
    if (denom!=0.0) {
        double T=(Y-edge2[0].y)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=edge2[0].x-X+T*(edge2[1].x-/

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X+S;
                polygon[polyVerts].y=Y;
                polyVerts++;
                break;
            }
        }
    }
}

}

double polyArea=0.0;

```

```

        for (int polyCount=0;polyCount<polyVerts;polyCount++) /
polyArea+=0.5*(polygon[0].x*(polygon[(polyCount+2)%polyVerts].y-polygon[(/
polyCount+1)%polyVerts].y)+polygon[(polyCount+1)%polyVerts].x*(polygon[0].y-/
polygon[(polyCount+2)%polyVerts].y)+polygon[(polyCount+2)%polyVerts].x*(polygon[(/
polyCount+1)%polyVerts].y-polygon[0].y));
        areaSum+=polyArea*(getpixel(image, x, y)&0xFF);
        continue;
    }
    break;
case 2:
    {
        vertex polygon[5];
        int polyVerts=0;
        int t=0;
        for (;t<3;t++) if (!corners[t]) break;
        vertex edge1[2]={tri[(t+1)%3], tri[t]};
        t=(t+2)%3;
        vertex edge2[2]={tri[t], tri[(t+1)%3]};
        polygon[0]=edge1[0]; polyVerts++;
        polygon[1]=edge2[0]; polyVerts++;
        int counter=0;
        for (;counter<4;counter++) {
            if (counter==0) {
                double X=x+1;
                double Y=y+1;
                double denom=edge2[1].x-edge2[0].x;
                if (denom!=0.0) {
                    double T=(X-edge2[0].x)/denom;
                    if (T>=0.0&&T<=1.0) {
                        double S=Y-T*(edge2[1].y-edge2[0].y)-/
edge2[0].y;

```

```

        if (S>=0.0&&S<=1.0) {
            polygon[polyVerts].x=X;
            polygon[polyVerts].y=Y-S;
            polyVerts++;
            break;
        }
    }
}

if (counter==1) {
    double X=x+1;
    double Y=y;
    double denom=edge2[1].y-edge2[0].y;
    if (denom!=0.0) {
        double T=(Y-edge2[0].y)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=X-T*(edge2[1].x-edge2[0].x)-/
edge2[0].x;

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X-S;
                polygon[polyVerts].y=Y;
                polyVerts++;
                break;
            }
        }
    }
}

if (counter==2) {
    double X=x;
    double Y=y;
    double denom=edge2[1].x-edge2[0].x;

```

```

    if (denom!=0.0) {
        double T=(X-edge2[0].x)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=edge2[0].y-Y+T*(edge2[1].y-
edge2[0].y);

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y+S;
                polyVerts++;
                break;
            }
        }
    }
}
if (counter==3) {
    double X=x;
    double Y=y+1;
    double denom=edge2[1].y-edge2[0].y;
    if (denom!=0.0) {
        double T=(Y-edge2[0].y)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=edge2[0].x-X+T*(edge2[1].x-
edge2[0].x);

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X+S;
                polygon[polyVerts].y=Y;
                polyVerts++;
                break;
            }
        }
    }
}

```



```

    }
}

switch (counter) {
    case 0:
    {
        double X=x+1;
        double Y=y+1;
        double denom=edge1[1].x-edge1[0].x;
        if (denom!=0.0) {
            double T=(X-edge1[0].x)/denom;
            if (T>=0.0&&T<=1.0) {
                double S=Y-T*(edge1[1].y-edge1[0].y/
)-edge1[0].y;

                if (S>=0.0&&S<=1.0) {
                    polygon[polyVerts].x=X;
                    polygon[polyVerts].y=Y-S;
                    polyVerts++;
                }
            }
        }
        break;
    case 1:
    {
        double X=x+1;
        double Y=y;
        double denom=edge1[1].y-edge1[0].y;
        if (denom!=0.0) {
            double T=(Y-edge1[0].y)/denom;
            if (T>=0.0&&T<=1.0) {

```

```

    )-edge1[0].x;

    double S=X-T*(edge1[1].x-edge1[0].x/

    if (S>=0.0&&S<=1.0) {
        polygon[polyVerts].x=X-S;
        polygon[polyVerts].y=Y;
        polyVerts++;
    }
    }
    }
    break;
case 2:
    {
        double X=x;
        double Y=y;
        double denom=edge1[1].x-edge1[0].x;
        if (denom!=0.0) {
            double T=(X-edge1[0].x)/denom;
            if (T>=0.0&&T<=1.0) {
                double S=edge1[0].y-Y+T*(edge1[1].y/

            -edge1[0].y);

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y+S;
                polyVerts++;
            }
        }
    }
    break;
case 3:

```



```

    )-edge1[0].y;

    double S=Y-T*(edge1[1].y-edge1[0].y/

    if (S>=0.0&&S<=1.0) {
        polygon[polyVerts].x=X;
        polygon[polyVerts].y=Y;
        polyVerts++;
        polygon[polyVerts].x=X;
        polygon[polyVerts].y=Y-S;
        polyVerts++;
    }
    }
    }
    break;
case 1:
    {
        double X=x+1;
        double Y=y;
        double denom=edge1[1].y-edge1[0].y;
        if (denom!=0.0) {
            double T=(Y-edge1[0].y)/denom;
            if (T>=0.0&&T<=1.0) {
                double S=X-T*(edge1[1].x-edge1[0].x/

                if (S>=0.0&&S<=1.0) {
                    polygon[polyVerts].x=X;
                    polygon[polyVerts].y=Y;
                    polyVerts++;
                    polygon[polyVerts].x=X-S;
                    polygon[polyVerts].y=Y;
                    polyVerts++;
                }
            }
        }
    }
}

```

```

        }
    }
}
break;
case 2:
{
    double X=x;
    double Y=y;
    double denom=edge1[1].x-edge1[0].x;
    if (denom!=0.0) {
        double T=(X-edge1[0].x)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=edge1[0].y-Y+T*(edge1[1].y/
-edge1[0].y);

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y;
                polyVerts++;
                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y+S;
                polyVerts++;
            }
        }
    }
}
break;
case 3:
{
    double X=x;
    double Y=y+1;

```

```

        double denom=edge1[1].y-edge1[0].y;
        if (denom!=0.0) {
            double T=(Y-edge1[0].y)/denom;
            if (T>=0.0&&T<=1.0) {
                double S=edge1[0].x-X+T*(edge1[1].x/
-edge1[0].x);

                if (S>=0.0&&S<=1.0) {
                    polygon[polyVerts].x=X;
                    polygon[polyVerts].y=Y;
                    polyVerts++;
                    polygon[polyVerts].x=X+S;
                    polygon[polyVerts].y=Y;
                    polyVerts++;
                }
            }
        }
    }
}

double polyArea=0.0;
for (int polyCount=0;polyCount<polyVerts;polyCount++) /
polyArea+=0.5*(polygon[0].x*(polygon[(polyCount+2)%polyVerts].y-polygon[(/
polyCount+1)%polyVerts].y)+polygon[(polyCount+1)%polyVerts].x*(polygon[0].y-/
polygon[(polyCount+2)%polyVerts].y)+polygon[(polyCount+2)%polyVerts].x*(polygon[(/
polyCount+1)%polyVerts].y-polygon[0].y));

    areaSum+=polyArea*(getpixel(image, x, y)&0xFF);
    continue;
}
break;
case 3:
{

```

```

        areaSum+=(0.5*(tri[0].x*(tri[2].y-tri[1].y)+tri[1].x*(tri[0].y/
-tri[2].y)+tri[2].x*(tri[1].y-tri[0].y))*(getpixel(image, x, y)&0xFF))/255.0;
        continue;
    }
    break;
}
continue;
}
bool edges[3]={false, false, false};
int edgecount=0;
for (int t=0;t<3;t++) {
    double dx=tri[(t+1)%3].x-tri[t].x;
    double dy=tri[(t+1)%3].y-tri[t].y;
    if (dy==0.0) {
        if (tri[t].y>=y&&tri[t].y<y+1) {
            edges[t]=true;
            edgecount++;
        }
    }
    else {
        if (dx==0.0) {
            if (tri[t].x>=x&&tri[t].x<x+1) {
                edges[t]=true;
                edgecount++;
            }
        }
        else {
            double M=dy/dx;
            double C=tri[t].y-M*tri[t].x;
            double height=M*x+C;
            if (height>=y&&height<y+1) {

```

```

        if ((tri[t].x<=x&&tri[(t+1)%3].x>=x)||((tri[(t+1)%3].x<=x/
&&tri[t].x>=x)) {

            edges[t]=true;
            edgecount++;
            continue;
        }
    }
    if (height+M>=y&&height+M<y+1) {
        if ((tri[t].x<=x+1&&tri[(t+1)%3].x>=x+1)||((tri[(t+1)%3].x/
<=x+1&&tri[t].x>=x+1)) {
            edges[t]=true;
            edgecount++;
            continue;
        }
    }
    M=dx/dy;
    C=tri[t].x-M*tri[t].y;
    double width=M*y+C;
    if (width>=x&&width<x+1) {
        if ((tri[t].y<=y&&tri[(t+1)%3].y>=y)||((tri[(t+1)%3].y<=y/
&&tri[t].y>=y)) {
            edges[t]=true;
            edgecount++;
            continue;
        }
    }
    if (width+M>=x&&width+M<x+1) {
        if ((tri[t].y<=y+1&&tri[(t+1)%3].y>=y+1)||((tri[(t+1)%3].y/
<=y+1&&tri[t].y>=y+1)) {
            edges[t]=true;
            edgecount++;

```



```

        continue;
    }
}
}
}
}
if (edgecount>0) {
    switch (edgecount) {
        case 1:
        {
            vertex polygon[5];
            int polyVerts=0;
            int t=0;
            for (;t<3;t++) if (edges[t]) break;
            vertex edge[2]={tri[t],tri[(t+1)%3]};
            for (int counter=0;counter<4;counter++) {
                switch (counter) {
                    case 0:
                    {
                        double X=x+1;
                        double Y=y+1;
                        vertex edgeDir={edge[1].y-edge[0].y,edge[0].x-/
edge[1].x};
                        if (edgeDir.x*(X-edge[0].x)+edgeDir.y*(Y-/
edge[0].y)>0.0) {
                            polygon[polyVerts].x=X;
                            polygon[polyVerts].y=Y;
                            polyVerts++;
                        }
                        double denom=edge[1].x-edge[0].x;
                        if (denom!=0.0) {

```

```

double T=(X-edge[0].x)/denom;
if (T>=0.0&&T<=1.0) {
    double S=Y-T*(edge[1].y-edge[0].y)-/
edge[0].y;

    if (S>=0.0&&S<=1.0) {
        polygon[polyVerts].x=X;
        polygon[polyVerts].y=Y-S;
        polyVerts++;
    }
}
}
break;
case 1:
{
    double X=x+1;
    double Y=y;
    vertex edgeDir={edge[1].y-edge[0].y,edge[0].x-/
edge[1].x};

    if (edgeDir.x*(X-edge[0].x)+edgeDir.y*(Y-/
edge[0].y)>0.0) {

        polygon[polyVerts].x=X;
        polygon[polyVerts].y=Y;
        polyVerts++;
    }
    double denom=edge[1].y-edge[0].y;
    if (denom!=0.0) {
        double T=(Y-edge[0].y)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=X-T*(edge[1].x-edge[0].x)-/
edge[0].x;

```

```

        if (S>=0.0&&S<=1.0) {
            polygon[polyVerts].x=X-S;
            polygon[polyVerts].y=Y;
            polyVerts++;
        }
    }
}
break;
case 2:
{
    double X=x;
    double Y=y;
    vertex edgeDir={edge[1].y-edge[0].y,edge[0].x-/
edge[1].x};

    if (edgeDir.x*(X-edge[0].x)+edgeDir.y*(Y-/
edge[0].y)>0.0) {

        polygon[polyVerts].x=X;
        polygon[polyVerts].y=Y;
        polyVerts++;
    }
    double denom=edge[1].x-edge[0].x;
    if (denom!=0.0) {
        double T=(X-edge[0].x)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=edge[0].y-Y+T*(edge[1].y-/
edge[0].y);

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y+S;
                polyVerts++;
            }
        }
    }
}

```

```

    }
    }
    }
    }
    break;
case 3:
    {
        double X=x;
        double Y=y+1;
        vertex edgeDir={edge[1].y-edge[0].y,edge[0].x-/
edge[1].x};

        if (edgeDir.x*(X-edge[0].x)+edgeDir.y*(Y-/
edge[0].y)>0.0) {

            polygon[polyVerts].x=X;
            polygon[polyVerts].y=Y;
            polyVerts++;
        }
        double denom=edge[1].y-edge[0].y;
        if (denom!=0.0) {
            double T=(Y-edge[0].y)/denom;
            if (T>=0.0&&T<=1.0) {
                double S=edge[0].x-X+T*(edge[1].x-/
edge[0].x);

                if (S>=0.0&&S<=1.0) {
                    polygon[polyVerts].x=X+S;
                    polygon[polyVerts].y=Y;
                    polyVerts++;
                }
            }
        }
    }
}

```

```

    }
}
double polyArea=0.0;
for (int polyCount=0;polyCount<polyVerts;polyCount++) /
polyArea+=0.5*(polygon[0].x*(polygon[(polyCount+2)%polyVerts].y-polygon[(/
polyCount+1)%polyVerts].y)+polygon[(polyCount+1)%polyVerts].x*(polygon[0].y-/
polygon[(polyCount+2)%polyVerts].y)+polygon[(polyCount+2)%polyVerts].x*(polygon[(/
polyCount+1)%polyVerts].y-polygon[0].y));
areaSum+=polyArea*(getpixel(image, x, y)&0xFF);
continue;
}
break;
case 2:
{
vertex polygon[6];
int polyVerts=0;
int t=0;
for (;t<3;t++) if (!edges[t]) break;
t=(t+1)%3;
vertex edge1[2]={tri[t], tri[(t+1)%3]};
t=(t+1)%3;
vertex edge2[2]={tri[t], tri[(t+1)%3]};
int counter=0;
int intersectCorners=0;
int intersectCount=0;
for (;counter<4;counter++) {
if (counter==0) {
double X=x+1;
double Y=y+1;
double denom=edge1[1].x-edge1[0].x;
if (denom!=0.0) {

```

```

double T=(X-edge1[0].x)/denom;
if (T>=0.0&&T<=1.0) {
    double S=Y-T*(edge1[1].y-edge1[0].y)-/
edge1[0].y;

    if (S>=0.0&&S<=1.0) {
        intersectCount++;
        polygon[polyVerts].x=X;
        polygon[polyVerts].y=Y-S;
        polyVerts++;
        if (intersectCount==2) break;
        intersectCorners=counter;
    }
}
}
}
if (counter==1) {
    double X=x+1;
    double Y=y;
    double denom=edge1[1].y-edge1[0].y;
    if (denom!=0.0) {
        double T=(Y-edge1[0].y)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=X-T*(edge1[1].x-edge1[0].x)-/
edge1[0].x;

            if (S>=0.0&&S<=1.0) {
                intersectCount++;
                polygon[polyVerts].x=X-S;
                polygon[polyVerts].y=Y;
                polyVerts++;
                if (intersectCount==2) break;
                intersectCorners=counter;
            }
        }
    }
}
}

```

```

        }
    }
}

if (counter==2) {
    double X=x;
    double Y=y;
    double denom=edge1[1].x-edge1[0].x;
    if (denom!=0.0) {
        double T=(X-edge1[0].x)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=edge1[0].y-Y+T*(edge1[1].y-
edge1[0].y);

            if (S>=0.0&&S<=1.0) {
                intersectCount++;
                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y+S;
                polyVerts++;
                if (intersectCount==2) break;
                intersectCorners=counter;
            }
        }
    }
}

if (counter==3) {
    double X=x;
    double Y=y+1;
    double denom=edge1[1].y-edge1[0].y;
    if (denom!=0.0) {
        double T=(Y-edge1[0].y)/denom;
        if (T>=0.0&&T<=1.0) {

```

```

edge1[0].x);

double S=edge1[0].x-X+T*(edge1[1].x-/

if (S>=0.0&&S<=1.0) {
    intersectCount++;
    polygon[polyVerts].x=X+S;
    polygon[polyVerts].y=Y;
    polyVerts++;
    if (intersectCount==2) break;
    intersectCorners=counter;
}
}
}
}

if ((polygon[0].x-edge1[0].x)*(polygon[0].x-edge1[0].x)+(/
polygon[0].y-edge1[0].y)*(polygon[0].y-edge1[0].y)>(polygon[1].x-edge1[0].x)*(polygon/
[1].x-edge1[0].x)+(polygon[1].y-edge1[0].y)*(polygon[1].y-edge1[0].y)) {
    vertex temp=polygon[1];
    polygon[1]=polygon[0];
    polygon[0]=temp;
    counter=intersectCorners;
}

for (int tempCount=0;tempCount<4;tempCount++) {
    switch (counter) {
    case 0:
    {
        double X=x+1;
        double Y=y+1;
        vertex triDir={tri[1].y-tri[0].y,tri[0].x-tri[1].x};
        if (triDir.x*(X-tri[0].x)+triDir.y*(Y-tri[0].y)/
>0.0) {

```



```

        triDir.x=tri[2].y-tri[1].y,
        triDir.y=tri[1].x-tri[2].x;
        if (triDir.x*(X-tri[1].x)+triDir.y*(Y-tri[1]./
y)>0.0) {

            triDir.x=tri[0].y-tri[2].y,
            triDir.y=tri[2].x-tri[0].x;
            if (triDir.x*(X-tri[2].x)+triDir.y*(Y-
tri[2].y)>0.0) {

                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y;
                polyVerts++;
            }
        }
    }
    double denom=edge2[1].x-edge2[0].x;
    if (denom!=0.0) {
        double T=(X-edge2[0].x)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=Y-T*(edge2[1].y-edge2[0].y/
)-edge2[0].y;

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y-S;
                polyVerts++;
            }
        }
    }
}
break;
case 1:
{

```

```

double X=x+1;
double Y=y;
vertex triDir={tri[1].y-tri[0].y,tri[0].x-tri[1].x};
if (triDir.x*(X-tri[0].x)+triDir.y*(Y-tri[0].y)/
>0.0) {

    triDir.x=tri[2].y-tri[1].y,
    triDir.y=tri[1].x-tri[2].x;
    if (triDir.x*(X-tri[1].x)+triDir.y*(Y-tri[1]./
y)>0.0) {

        triDir.x=tri[0].y-tri[2].y,
        triDir.y=tri[2].x-tri[0].x;
        if (triDir.x*(X-tri[2].x)+triDir.y*(Y-/
tri[2].y)>0.0) {

            polygon[polyVerts].x=X;
            polygon[polyVerts].y=Y;
            polyVerts++;
        }
    }
}

double denom=edge2[1].y-edge2[0].y;
if (denom!=0.0) {
    double T=(Y-edge2[0].y)/denom;
    if (T>=0.0&&T<=1.0) {
        double S=X-T*(edge2[1].x-edge2[0].x/
)-edge2[0].x;

        if (S>=0.0&&S<=1.0) {
            polygon[polyVerts].x=X-S;
            polygon[polyVerts].y=Y;
            polyVerts++;
        }
    }
}

```

```

    }
}
break;
case 2:
{
    double X=x;
    double Y=y;
    vertex triDir={tri[1].y-tri[0].y,tri[0].x-tri[1].x};
    if (triDir.x*(X-tri[0].x)+triDir.y*(Y-tri[0].y)/
>0.0) {

        triDir.x=tri[2].y-tri[1].y,
        triDir.y=tri[1].x-tri[2].x;
        if (triDir.x*(X-tri[1].x)+triDir.y*(Y-tri[1]./
y)>0.0) {

            triDir.x=tri[0].y-tri[2].y,
            triDir.y=tri[2].x-tri[0].x;
            if (triDir.x*(X-tri[2].x)+triDir.y*(Y-/
tri[2].y)>0.0) {

                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y;
                polyVerts++;
            }
        }
    }
    double denom=edge2[1].x-edge2[0].x;
    if (denom!=0.0) {
        double T=(X-edge2[0].x)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=edge2[0].y-Y+T*(edge2[1].y/
-edge2[0].y);

            if (S>=0.0&&S<=1.0) {

```

```

        polygon[polyVerts].x=X;
        polygon[polyVerts].y=Y+S;
        polyVerts++;
    }
}
}
break;
case 3:
{
    double X=x;
    double Y=y+1;
    vertex triDir={tri[1].y-tri[0].y,tri[0].x-tri[1].x};
    if (triDir.x*(X-tri[0].x)+triDir.y*(Y-tri[0].y)/
>0.0) {
        triDir.x=tri[2].y-tri[1].y,
        triDir.y=tri[1].x-tri[2].x;
        if (triDir.x*(X-tri[1].x)+triDir.y*(Y-tri[1]./
y)>0.0) {
            triDir.x=tri[0].y-tri[2].y,
            triDir.y=tri[2].x-tri[0].x;
            if (triDir.x*(X-tri[2].x)+triDir.y*(Y-/
tri[2].y)>0.0) {
                polygon[polyVerts].x=X;
                polygon[polyVerts].y=Y;
                polyVerts++;
            }
        }
    }
    double denom=edge2[1].y-edge2[0].y;
    if (denom!=0.0) {

```

```

        double T=(Y-edge2[0].y)/denom;
        if (T>=0.0&&T<=1.0) {
            double S=edge2[0].x-X+T*(edge2[1].x/
-edge2[0].x);

            if (S>=0.0&&S<=1.0) {
                polygon[polyVerts].x=X+S;
                polygon[polyVerts].y=Y;
                polyVerts++;
            }
        }
    }
}

counter++;
counter=counter%4;
}

double polyArea=0.0;
for (int polyCount=0;polyCount<polyVerts;polyCount++) /
polyArea+=0.5*(polygon[0].x*(polygon[(polyCount+2)%polyVerts].y-polygon[(/
polyCount+1)%polyVerts].y)+polygon[(polyCount+1)%polyVerts].x*(polygon[0].y-/
polygon[(polyCount+2)%polyVerts].y)+polygon[(polyCount+2)%polyVerts].x*(polygon[(/
polyCount+1)%polyVerts].y-polygon[0].y));

    areaSum+=polyArea*(getpixel(image, x, y)&0xFF);
    continue;
}
break;
case 3:
{
    areaSum+=0.25;
}
break;

```

```

    }
}
bool intri=true;
for (int t=0;t<3;t++) {
    vertex v1={tri[(t+1)%3].y-tri[t].y, tri[t].x-tri[(t+1)%3].x};
    vertex v2={x-tri[t].x, y-tri[t].y};
    if (v1.x*v2.x+v1.y*v2.y<0.0) {
        intri=false;
        break;
    }
}
if (intri) {
    areaSum+=getpixel(image, x, y)&0xFF;
    continue;
}
}

if (SDL_MUSTLOCK(image)) SDL_UnlockSurface(image);
double areaFraction=0.0;
if (trueArea!=0.0) areaFraction=areaSum/trueArea;
if (!std::isfinite(areaFraction)) areaFraction=0.5;
if (areaFraction<0.0) areaFraction=0.0;
if (areaFraction>1.0) areaFraction=1.0;

if (SDL_MUSTLOCK(image)) SDL_LockSurface(image);
if (SDL_MUSTLOCK(mask)) SDL_LockSurface(mask);
{
    double regionArea=0.5*(x0*y2-x0*y1+x2*y1-x2*y0+x1*y0-x1*y2);
    double subregionArea=regionArea*areaFraction*correction;
    static double previousPx=cx;
    static double previousPy=cy;

```

```

double Px,Py;

double majorAxisX=0.5*(x2+x1)-x0;
double majorAxisY=0.5*(y2+y1)-y0;
{
    double majorAxisLength=sqrt(majorAxisX*majorAxisX+majorAxisY*
majorAxisY);
    majorAxisX/=majorAxisLength;
    majorAxisY/=majorAxisLength;
}

if (((x1-x0)*(x1-x0)+(y1-y0)*(y1-y0))>((x2-x0)*(x2-x0)+(y2-x0)*(y2-y0))) {
    double& Mx=majorAxisX;
    double& My=majorAxisY;
    double t=(My*y0-My*y2+Mx*x0-Mx*x2)/(-My*y1+My*y0-Mx*x1+Mx*x0/
);

    double x3=x0+(x1-x0)*t;
    double y3=y0+(y1-y0)*t;
    double a=0.5*(x0*y2-x0*y3+x2*y3-x2*y0+x3*y0-x3*y2);
    if (a<subregionArea) {
        subregionArea=regionArea-subregionArea;
        a=regionArea-a;
        a*=2.0;
        if (a==0.0) t=0.0;
        else t=sqrt(2.0*a*subregionArea)/a;
        Px=x1-t*(0.5*(x2+x3)-x1);
        Py=y1-t*(0.5*(y2+y3)-y1);
    }
    else {
        a*=2.0;
        if (a==0.0) t=0.0;

```

```

        else t=sqrt(2.0*a*subregionArea)/a;
        Px=t*(0.5*(x2+x3)-x0)+x0;
        Py=t*(0.5*(y2+y3)-y0)+y0;
    }
}
else {
    double& Mx=majorAxisX;
    double& My=majorAxisY;
    double t=(My*y0-My*y1+Mx*x0-Mx*x1)/(-My*y2+My*y0-Mx*x2+Mx*x0/
);
    double x3=x0+(x2-x0)*t;
    double y3=y0+(y2-y0)*t;
    double a=0.5*(x0*y3-x0*y1+x3*y1-x3*y0+x1*y0-x1*y3);
    if (a<subregionArea) {
        subregionArea=regionArea-subregionArea;
        a=regionArea-a;
        a*=2.0;
        if (a==0.0) t=0.0;
        else t=sqrt(2.0*a*subregionArea)/a;
        Px=x2-t*(0.5*(x1+x3)-x2);
        Py=y2-t*(0.5*(y1+y3)-y2);
    }
    else {
        a*=2.0;
        if (a==0.0) t=0.0;
        else t=sqrt(2.0*a*subregionArea)/a;
        Px=t*(0.5*(x1+x3)-x0)+x0;
        Py=t*(0.5*(y1+y3)-y0)+y0;
    }
}
}

```



```

previousPx=Px;
previousPy=Py;
distances<<areaFraction<<std::endl;
coordinates<<Px<<"\t"<<image->h-Py<<std::endl;

        if (particle1Found&&trackParticle1&&!trackParticle2) {
part2.x=Px;
part2.y=Py;
trackParticle2=true;
}

if (particle1Found&&!trackParticle1) {
    part1.x=Px;
    part1.y=Py;
    trackParticle1=true;
}

//Particle Tracking
if (trackParticle1) {
    double DirX=y3-y4;
    double DirY=x4-x3;
    double VerX=part1.x-x4;
    double VerY=part1.y-y4;
    if (DirX*VerX+DirY*VerY>0.0) {
        DirX=y0-y3;
        DirY=x3-x0;
        VerX=part1.x-x3;
        VerY=part1.y-y3;
        if (DirX*VerX+DirY*VerY>0.0) {
            DirX=y4-y0;

```

```

    DirY=x0-x4;
    VerX=part1.x-x0;
    VerY=part1.y-y0;
    if (DirX*VerX+DirY*VerY>0.0) {
        double& tx=part1.x;
        double& ty=part1.y;
        double s=(tx*y0-tx*Py+ty*Px+x0*Py-y0*Px-ty*x0)/(-/
majorAxisX*tx+x0*majorAxisX-ty*majorAxisY+y0*majorAxisY);
        part1.x=Px+s*majorAxisY;
        part1.y=Py-s*majorAxisX;
        if (trackParticle1&&trackParticle2)
            particle<<"Particle_1:"<<sqrt((/
part1.x-part2.x)*(part1.x-part2.x)+(part1.y-part2.y)*(part1.y-part2.y))<<std::endl;
    }
}
}

if (trackParticle2) {
    double DirX=y3-y4;
    double DirY=x4-x3;
    double VerX=part2.x-x4;
    double VerY=part2.y-y4;
    if (DirX*VerX+DirY*VerY>0.0) {
        DirX=y0-y3;
        DirY=x3-x0;
        VerX=part2.x-x3;
        VerY=part2.y-y3;
        if (DirX*VerX+DirY*VerY>0.0) {
            DirX=y4-y0;
            DirY=x0-x4;

```

```

VerX=part2.x-x0;
VerY=part2.y-y0;
if (DirX*VerX+DirY*VerY>0.0) {
    double& tx=part2.x;
    double& ty=part2.y;
    double s=(tx*y0-tx*Py+ty*Px+x0*Py-y0*Px-ty*x0)/(-/
majorAxisX*tx+x0*majorAxisX-ty*majorAxisY+y0*majorAxisY);
    part2.x=Px+s*majorAxisY;
    part2.y=Py-s*majorAxisX;
    if (trackParticle1&&trackParticle2)
        particle<<"Particle_2:"<<sqrt((/
part1.x-part2.x)*(part1.x-part2.x)+(part1.y-part2.y)*(part1.y-part2.y))<<std::endl;
    }
}
}
}

for (int y=output[0];y<output[1];y++) {
    double B=y+0.5;
    for (int x=yBufferLeft[y];x<=yBufferRight[y];x++) {
        putpixel(mask, x, y, 0x01);// THIS IS THE TRANSPARENT COLOUR /
KEY VALUE

        double A=x+0.5;
        double d=(A-Px)*majorAxisX+(B-Py)*majorAxisY;
        if (d<=-0.5) {
            putpixel(image, x, y, SDL_MapRGB(image->format, 255, 210, 255));
            continue;
        }
        if (d>=0.5) {
            putpixel(image, x, y, SDL_MapRGB(image->format, 150, 55, 0));

```

```

        continue;
    }
    d=0.5-d;
    unsigned char colr=(unsigned char) (150.0+d*105.0);
    unsigned char colg=(unsigned char) (55.0+d*155.0);
    unsigned char colb=(unsigned char) (d*255.0);
    putpixel(image, x, y, SDL_MapRGB(image->format, colr, colg, colb));
}
}
}

if (SDL_MUSTLOCK(mask)) SDL_UnlockSurface(mask);
if (SDL_MUSTLOCK(image)) SDL_UnlockSurface(image);

for(int y=0;y<image->h;y++) {
    yBufferLeft[y]=0;
    yBufferRight[y]=image->w-1;
}

triple[0][0]=x4;
triple[0][1]=y4;
triple[1][0]=x3;
triple[1][1]=y3;

scanConvertTriangle( triple, yBufferLeft, yBufferRight, output );
if (SDL_MUSTLOCK(mask)) SDL_LockSurface(mask);
    for (int y=output[0];y<output[1];y++) {
        for (int x=yBufferLeft[y];x<=yBufferRight[y];x++) {
            putpixel(mask, x, y, SDL_MapRGB(mask->format, 180, 180, 180));// THIS/
            IS THE MASK COLOUR
        }
    }
}

```

```

    if (SDL_MUSTLOCK(mask)) SDL_UnlockSurface(mask);
    delete [] yBufferRight;
    delete [] yBufferLeft;
}

void CSurface::update(double gamma)
{
    A.x=C.x;
    A.y=C.y;
    E.x=D.x;
    E.y=D.y;
    distances<<gamma-pi/2<<"\t";
    vertex polygon[]={0.85736514974659426029608593904448, 0.525}, /
    {-0.85736514974659426029608593904448, 0.525}, {0.0, -0.96}}; // TRIANGLE
    //vertex polygon[]={1, 0}, {0.86603, 0.5}, {-0.5, 0.86603}, {-0.866, 0.5}, {-0.5, /
    -0.866}, {0.0, -1.0}}; // CLIPPED CORNER TRIANGLE
    //vertex polygon[]={0.0,0.85736514974659426029608593904588}, {-0.2, /
    -0.85736514974659426029608593904588}, {0.2, /
    -0.85736514974659426029608593904588}}; // TRIANGLE
    //vertex polygon[]={0.95, 0.95}, {-0.95, 0.95}, {-0.95, -0.95}, {0.95, -0.95}}; // /
    SQUARE
    //vertex polygon[]={0.95, 0.69}, {-0.39, 0.69}, {-0.95, -0.69}, {0.39, -0.69}}; // /
    PARALLELOGRAM
    //vertex polygon[]={0.95, 0.54}, {-0.95, 0.54}, {-0.95, -0.54}, {0.95, -0.54}}; // /
    RECTANGLE
    //vertex polygon[]={0.45, 0.99}, {-0.45, 0.99}, {-0.99, -0.99}, {0.99, -0.99}}; // /
    FISH
    //vertex polygon[]={0.25, 0.5}, {-0.25, 0.5}, {-0.5, -0.5}, {0.5, -0.5}}; // FISH 2
    //vertex polygon[]={0.99, 0.99}, {-0.45, 0.99}, {-0.99, -0.99}, {0.45, -0.99}}; // /
    PARALLELOGRAM

```

```

//vertex polygon[]={{0.0, 1.0},{-0.95105651629515357211643933337938, /
0.30901699437494742410229341718282},{-0.58778525229247312916870595463907, /
-0.80901699437494742410229341718282},{0.58778525229247312916870595463907, /
-0.80901699437494742410229341718282},{0.95105651629515357211643933337938, /
0.30901699437494742410229341718282}}; //PENTAGON

// vertex polygon[50]; // CIRCLE
// for (int x=0;x<50;x++) {
// polygon[x].x=0.99*cos(0.12566370614359172953850573533118*x)+((double)rand()/((double/
)RAND_MAX)*0.00001;
// polygon[x].y=0.99*sin(0.12566370614359172953850573533118*x)+((double)rand()/((double/
)RAND_MAX)*0.00001;
// }

// int n=5; // REGULAR N-GON
// double step=6.283185307179586476925286766559/((double)n;
// vertex polygon[n];
// if (n%2) {
// for (int x=0;x<n;x++) {
// polygon[x].x=0.99*cos(step*x);
// polygon[x].y=0.99*sin(step*x);
// }
// }
// else {
// for (int x=0;x<n;x++) {
// polygon[x].x=0.99*cos(step*x)+((double)rand()/((double)RAND_MAX)*0.0001;
// polygon[x].y=0.99*sin(step*x)+((double)rand()/((double)RAND_MAX)*0.0001;
// }
// }

```

```

// vertex polygon[49]; // CAPSULE
// for (int x=0;x<25;x++) {
// polygon[x].x=0.65*cos(0.13089969389957471826927680763665*x/
// +1.5707963267948966192313216916398)-0.3;
// polygon[x].y=0.65*sin(0.13089969389957471826927680763665*x/
// +1.5707963267948966192313216916398);
// }
// for (int x=25;x<49;x++) {
// polygon[x].x=0.65*cos((x-25)/
// *0.13659098493868666254185406014259+4.7123889803846898576939650749193)+0.3;
// polygon[x].y=0.65*sin((x-25)/
// *0.13659098493868666254185406014259+4.7123889803846898576939650749193);
// }

// vertex polygon[50]; // ELLIPSE
// for (int x=0;x<50;x++) {
// polygon[x].x=0.99*cos(0.12566370614359172953850573533118*x)+((double)rand()/((double/
// )RAND_MAX)*0.00001;
// polygon[x].y=0.56*sin(0.12566370614359172953850573533118*x)+((double)rand()/((double/
// )RAND_MAX)*0.00001;
// }

// vertex polygon[50]; // EGG
// for (int x=0;x<50;x++) {
// polygon[x].x=0.99*cos(0.12566370614359172953850573533118*x)+((double)rand()/((double/
// )RAND_MAX)*0.00001;
// polygon[x].y=(2.0-1.0f*cos(0.12566370614359172953850573533118*x))*sin/
// (0.12566370614359172953850573533118*x)/3.0f+((double)rand()/((double)RAND_MAX)/
// *0.00001;

```

```
// }
```

```
double surfaceNormalX=cos(gamma);
```

```
double surfaceNormalY=sin(gamma);
```

```
double& surfaceX=surfaceNormalY;
```

```
double surfaceY=-surfaceNormalX;
```

```
// SETUP SORTED LIST
```

```
int polySize=sizeof(polygon)/sizeof(vertex);
```

```
int* sortedIndex=new int[polySize];
```

```
for(int t=0;t<polySize;t++) sortedIndex[t]=t;
```

```
bool swap;
```

```
do {
```

```
    swap=false;
```

```
    for(int t=0;t<polySize-1;t++) {
```

```
        if ((polygon[sortedIndex[t]].x*surfaceNormalX+polygon[sortedIndex[t]].y*/
surfaceNormalY)>
```

```
            (polygon[sortedIndex[t+1]].x*surfaceNormalX+polygon[sortedIndex[t+1]].y*/
surfaceNormalY)) {
```

```
                int temp=sortedIndex[t+1];
```

```
                sortedIndex[t+1]=sortedIndex[t];
```

```
                sortedIndex[t]=temp;
```

```
                swap=true;
```

```
            }
```

```
    }
```

```
} while (swap);
```

```
// CALCULATE THE INITIAL AREA
```

```
double area=0.0;
```



```

for(int t=1;t<polySize-1;t++) {
    area+=0.5*((polygon[0].x-polygon[t].x)*(polygon[0].y-polygon[t+1].y)-(polygon[0]./
x-polygon[t+1].x)*(polygon[0].y-polygon[t].y));
}
double fillArea=area*fillFraction;

// INITIALIZE SUBREGION TO ALLOW LOWER TRIANGULAR REGION
vertex subregion[4]={ {polygon[sortedIndex[0]].x,polygon[sortedIndex[0]].y},{polygon[/
sortedIndex[0]].x,polygon[sortedIndex[0]].y}};
double regionArea[2]={0.0};
int subregionMarker=2;

for(int sortedCounter=1;sortedCounter<polySize-1;sortedCounter++) {
    // LOCATE THE INTERSECTION OF THE EDGE
    vertex intersection;
    intersection.x=polygon[sortedIndex[sortedCounter]].x;
    intersection.y=polygon[sortedIndex[sortedCounter]].y;
    int currentVertex;
    for (currentVertex=0;currentVertex<polySize;currentVertex++) {
        double& x0=polygon[currentVertex].x;
        double& y0=polygon[currentVertex].y;
        if (sortedIndex[sortedCounter]==currentVertex) continue;
        double& x1=polygon[(currentVertex+1)%polySize].x;
        double& y1=polygon[(currentVertex+1)%polySize].y;
        if (sortedIndex[sortedCounter]==currentVertex+1) continue;
        double t1=surfaceX*y0-surfaceY*x0;
        double denom=t1-surfaceX*y1+surfaceY*x1;
        if (denom==0.0) continue;
        double s=(t1-surfaceX*polygon[sortedIndex[sortedCounter]].y+surfaceY*/
polygon[sortedIndex[sortedCounter]].x)/denom;

```

```

    if (s<0.00001||s>1.00001) continue;
    double X=x0+s*(x1-x0);
    double Y=y0+s*(y1-y0);
    intersection.x=X;
    intersection.y=Y;

    if (X>polygon[sortedIndex[sortedCounter]].x+0.000001&&X<polygon[sortedIndex[sortedCounter]].x-0.000001
        &&Y>polygon[sortedIndex[sortedCounter]].y+0.000001&&Y<polygon[sortedIndex[sortedCounter]].y-0.000001) continue;
    else break;
}

// GENERATE CURRENT AREA
vertex* tempRegion=new vertex[polySize];
int marker=0;
double tempX=intersection.x-polygon[sortedIndex[sortedCounter]].x;
double tempY=intersection.y-polygon[sortedIndex[sortedCounter]].y;
double tempDirection=tempX*surfaceX+tempY*surfaceY;
if (tempDirection>0.0) {
    tempRegion[marker++]=intersection;
    for(int t=sortedIndex[sortedCounter];t!=currentVertex;t=(t+1)%polySize) {
        tempRegion[marker++]=polygon[t];
    }
    tempRegion[marker++]=polygon[currentVertex];

    subregion[subregionMarker]=polygon[sortedIndex[sortedCounter]];
    subregion[subregionMarker+1]=intersection;
    subregionMarker=2-subregionMarker;
}

```

```

    else {
        tempRegion[marker++] = intersection;
        for(int t = (currentVertex + 1) % polySize; t != sortedIndex[sortedCounter]; t = (t /
+ 1) % polySize) {
            tempRegion[marker++] = polygon[t];
        }
        tempRegion[marker++] = polygon[sortedIndex[sortedCounter]];

        subregion[subregionMarker + 1] = polygon[sortedIndex[sortedCounter]];
        subregion[subregionMarker] = intersection;
        subregionMarker = 2 - subregionMarker;
    }
    // CALCULATE REGIONS AREA
    double tempRegionArea = 0.0;
    for (int t = 1; t < marker - 1; t++) {
        tempRegionArea += 0.5 * ((tempRegion[0].x - tempRegion[t].x) * (tempRegion[
0].y - tempRegion[t + 1].y) - (tempRegion[0].x - tempRegion[t + 1].x) * (tempRegion[0].y - /
tempRegion[t].y));
    }
    delete [] tempRegion;
    regionArea[(2 - subregionMarker) / 2] = tempRegionArea;

    // ESCAPE OR ALLOW TOP TRIANGULAR REGION
    if (tempRegionArea >= fillArea) break;
    if (sortedCounter == polySize - 2) {
        subregion[subregionMarker] = polygon[sortedIndex[polySize - 1]];
        subregion[subregionMarker + 1] = polygon[sortedIndex[polySize - 1]];
        subregionMarker = 2 - subregionMarker;
        regionArea[(2 - subregionMarker) / 2] = area;
    }
}

```

```
double x0=subregion[subregionMarker].x;
double y0=subregion[subregionMarker].y;

double x1=subregion[subregionMarker+1].x;
double y1=subregion[subregionMarker+1].y;
subregionMarker=2-subregionMarker;

double a0=regionArea[(2-subregionMarker)/2];

double x2=subregion[subregionMarker].x;
double y2=subregion[subregionMarker].y;

double x3=subregion[subregionMarker+1].x;
double y3=subregion[subregionMarker+1].y;
subregionMarker=2-subregionMarker;

double a=(x3-x1)*(y2-y0)-(y3-y1)*(x2-x0);
double b=0.5*((x1-x0)*(y3-y1)-(y1-y0)*(x3-x1)+(x1-x0)*(y2-y0)-(y1-y0)*(/
x2-x0));
double c=a0-fillArea;
double t;
if (a!=0) t=(-b+sqrt(b*b-2.0*a*c))/a;
else t=-c/b;

D.x=x0+t*(x2-x0);
D.y=y0+t*(y2-y0);
C.x=x1+t*(x3-x1);
C.y=y1+t*(y3-y1);
```

```
    if ((!std::isfinite(D.x))||(!std::isfinite(D.y))) {
        D.x=0.0;
        D.y=0.0;
    }

    if ((!std::isfinite(C.x))||(!std::isfinite(C.y))) {
        C.x=0.0;
        C.y=0.0;
    }

    delete [] sortedIndex;
{
    double& x0=A.x;
    double& y0=A.y;
    double& x1=E.x;
    double& y1=E.y;
    double sx=D.x-C.x;
    double sy=D.y-C.y;
    double& cx=C.x;
    double& cy=C.y;
    double t1=sx*y0-sy*x0;
    double denom=t1-sx*y1+sy*x1;
    if (denom!=0.0) {
        double s=(t1-sx*cy+sy*cx)/denom;
        B.x=x0+s*(x1-x0);
        B.y=y0+s*(y1-y0);
    }
}
}
```