**Solidification, growth mechanisms, and associated properties of Al-Si and
magnesium lightweight casting alloys**

by

Timothy Al Hosch

A dissertation submitted to the graduate faculty
in partial fulfillment of the requirements for the degree of
DOCTOR OF PHILOSOPHY

Major: Materials Science and Engineering

Program of Study Committee:
Ralph E. Napolitano, Major Professor
Iver Anderson
Kai-Ming Ho
Rohit Trivedi
Ersan Üstündag

Iowa State University

Ames, Iowa

2010

Copyright © Timothy Al Hosch, 2010. All rights reserved.

DEDICATION

I would like to dedicate this thesis to my family, major professor, committee members, and everyone else who helped to provide the general impetus to create such a document.

TABLE OF CONTENTS

LIST OF TABLES	vi
LIST OF FIGURES	vii
ABSTRACT	xix
INTRODUCTION	1
Introduction	1
Dissertation Organization	5
Literature Review	6
Eutectic Solidification Theory	6
Growth Mechanisms of Al-Si Eutectic Alloys	24
Interfacial Anisotropy	30
Interface Stability	34
References	41
ANALYSIS OF THE HIGH GROWTH-RATE TRANSITION	
IN AL-SI EUTECTIC SOLIDIFICATION	48
Abstract	48
Introduction	49
Experimental methods and findings	52
Analysis and discussion	53
Conclusions	62
References	64

THE EFFECT OF THE FLAKE-FIBER TRANSITION IN SILICON

MORPHOLOGY ON THE TENSILE PROPERTIES OF

AL-SI EUTECTIC ALLOYS 68

Abstract 68

Introduction 69

Experimental 70

Results and Discussion 71

Summary and Conclusions 75

References 76

MORPHOLOGICAL INSTABILITY OF SILICON DURING

DIRECTIONAL SOLIDIFICATION OF AN AL-SI EUTECTIC ALLOY 83

Abstract 83

Introduction 83

Experiments 85

Instability Analysis 86

 Model Results 92

Discussion 92

Conclusions 94

References 95

METALLURGICAL ANALYSIS OF FLOW-LINE INDICATIONS IN

MG-BASED ALLOY CASTINGS 114

Abstract 114

Introduction 114

Scope 115

Test Specimens 115

Microstructural Analysis 116

 Alloy WE43B 118

 Alloy EV31A 120

Alloy ZE41	123
Tension Testing and Fracture Analysis	127
Conclusions	131
ADDITIONAL ANALYSES OF THREE-DIMENSIONAL	
MICROSTRUCTURES	136
Introduction	136
Liquid structure quantification using a minimum spanning tree analysis	136
Minimum Spanning Tree	137
Algorithm	137
Structures	140
Results and Analysis	142
Conclusion	145
Local microstructure characterization through use of the radial distribution function	161
Calculation of the RDF	162
Results	163
Calculation of the critical diameter function in an Al-Si alloy	163
References	166
CONCLUSION	168
APPENDIX	171

LIST OF TABLES

Table 1	Reported Al-Si eutectic growth parameters from length scale (λ) measurements.	51
Table 2	Reported transition velocities.	52
Table 1	Temperature gradients measured in the present Bridgman-type directional solidification apparatus.	71
Table 2	Results of tension tests performed on Al-Si eutectic samples solidified at different velocities.	75
Table 1	Definition of symbols. *Composition parameters converted from wt% silicon to mol/m ³ for calculation of gradient.	100
Table 1	Nominal compositions of test specimens, provided by Wellman Dynamics.	116
Table 2	EDS analysis of alloy WE43B.	119
Table 3	EDS analysis of alloy EV31A (sample A). (Values in wt. percent.) . .	122
Table 4	EDS analysis of alloy ZE41 (sample 3B). (Values in wt. percent.) . . .	124
Table 5	Summary of tensile results for all samples.	130
Table 1	Percentage deviation range over which each MST statistic category is most similar for the ideal structures versus the MD simulated structures.	143

LIST OF FIGURES

Figure 1	Phase diagram of the Al-Si system.(4)	4
Figure 2	Schematic of eutectic solidification. The periodic arrangement of phases minimizes average diffusion distance, thus maximizing the efficiency of solute transport.	7
Figure 3	Shape of (a) nonfaceted and (b) faceted interfaces. From Kurz and Fisher.(5)	8
Figure 4	Comparison of structure in a regular eutectic (left) and irregular eutectic (right). The regular eutectic displays a relatively constant phase spacing and morphology while the irregular eutectic structure contains extreme variation in local spacing and features. Images from England.(6)	8
Figure 5	Schematic of a lamellar fault, as proposed by Jackson and Hunt.(7) Relative motion of the lamellar fault to the right will increase the average spacing of the system, while motion to the left will decrease the average spacing.	10
Figure 6	Planar interface morphology assumed in the calculation of the planar diffusion solution.	14
Figure 7	Phase diagram showing definition of several pertinent constants. . . .	14
Figure 8	Interface morphology assumed for the calculation of capillary undercooling showing definition of S and θ	15

Figure 9	Schematic showing the relation between spacing and total undercooling. The main undercooling contributor at low spacings is the capillary undercooling, while the diffusional undercooling dominates at large spacings. Regular eutectics tend to grow at or near the extremum point (minimum undercooling) while irregular eutectics grow at larger spacing values.	18
Figure 10	Interface morphologies assumed by (a) Jackson and Hunt(7), (b) Sato and Sayama(14), and (c) Fisher and Kurz(15).	21
Figure 11	Schematic of small angle (top) and large angle (bottom) branching mechanisms in eutectic silicon. Small angle branching involves a succession of displacement twins which create a branched arm and then cause it to diverge from the initial eutectic flake at a small, arbitrary angle. Large angle branching is possible when a silicon tetrahedron attaches to the $\{111\}$ surface of a silicon flake in orientation A. The attached tetrahedron will be in orientation B, and may then form a new orientation (C) by twinning. If a second twin of orientation B forms on C, it will produce a self-perpetuating groove which allows TPREG growth in another direction. Adapted from Shamsuzzoha and Hogan(25).	27
Figure 12	Schematic Wulff constructions: (a) The Wulff construction for the determination of the equilibrium shape from the γ -plot (b) The reverse Wulff construction for determining the γ plot from an equilibrium shape (c) The reverse Wulff construction performed on a body of missing orientations yields only a minimum γ -plot. From Miller and Chadwick(58)	33

Figure 13	Schematic of constitutional supercooling. Solidification in a binary alloy system is proceeding from left to right along the x axis with solute being rejected into the melt. As the composition of the liquid decreases with distance ahead of the interface, the equilibrium melting temperature increases. If the effect of the compositional profile on the equilibrium melting temperature is greater than the actual temperature gradient experienced during solidification, there will exist a region of constitutional supercooling (shaded region) where the liquid is at a lower temperature than its melting point. This region of liquid is thermodynamically unstable.	36
Figure 14	Amplification rate behavior of the stability of a perturbation of specified wavelength on a planar interface.	41
Figure 1	Typical silicon morphology after directional solidification of an Al-Si eutectic alloy in a temperature gradient of 7–14 K/mm over a velocity range from 10 to 950 $\mu\text{m/s}$. The Al-rich phase has been chemically removed.	54
Figure 2	Schematic demonstrating the calculation of the parallel and random line spacing measurements. The $\lambda_{ }$ measurement is determined by drawing lines perpendicular to parallel silicon plates while the λ_L measurement involves the generation of random lines over the eutectic area.	55
Figure 3	Graph of three different spacing parameters. The λ_L measurement reflects the average spacing considering all possible line orientations. The $\lambda_{ }$ measurement indicates the average spacing perpendicular to parallel silicon particles. The λ_A measurement is effectively an inverse particle density function, indicating that particles become more numerous within a constant volume at high velocities. Error bars represent the 95% confidence interval of the mean.	56

Figure 4	Graph of the average particle aspect ratio for the range of velocities studied. The aspect ratio displays distinct transition points near 135 $\mu\text{m/s}$ and 880 $\mu\text{m/s}$, which are near the visually observed onset and conclusion velocities of the flake to fiber transition. Error bars represent the 95% confidence interval of the mean.	57
Figure 5	Details of the quench modification process in an Al-13wt% Si alloy. At velocities from 10 to 250 $\mu\text{m/s}$ the transition is evident in the breakup of plates, with rodlike structures evolving within the envelope of the plate. At higher velocities, the next stage of the transition becomes evident as out-of-plane rod growth begins around 500 $\mu\text{m/s}$ (shown by arrows) and a shift to rod dominance occurs by 950 $\mu\text{m/s}$	61
Figure 6	Occurrence of morphological instability of plate faces in a sample grown at 950 $\mu\text{m/s}$	62
Figure 1	Comparison of the ultimate tensile strength measured in the present study and the aspect ratio and spacing results (λ_A) reported in (6). The tensile strength increases significantly with increasing solidification rate throughout the velocity range studied. Note the transition in slope of the tensile strength data that occurs at 250 $\mu\text{m/s}$. Similarly, the aspect ratio data display a transition in slope near 125 $\mu\text{m/s}$. The spacing is seen to decrease continuously with increasing solidification velocity. . .	80
Figure 2	Al-Si eutectic structures after directional solidification at velocities of 20, 250, and 950 $\mu\text{m/s}$ (6). These structures represent flake (left), mixed flake/fiber (center), and fiber (right) morphologies, respectively. . . .	81
Figure 3	The stress-strain behavior of samples directionally solidified at 20, 80, and 600 $\mu\text{m/s}$. The velocities represent predominantly flake (20 and 80 $\mu\text{m/s}$) and predominantly fibrous (600 $\mu\text{m/s}$) morphologies, respectively. . .	81
Figure 4	(a) Low, (b) intermediate, and (c) high magnification images of tension test specimen fracture surfaces solidified at 20, 250, and 1000 $\mu\text{m/s}$. . .	82

Figure 1	Al-Si interface schematic demonstrating both the development of in-plane (right) and out-of-plane (left) silicon protrusions. Both types of protrusion formation may develop separately, together, or not at all. The type of instability that forms on any given plate will be governed by the local flake growth conditions, which are widely variable and difficult to predict due to the inherent irregularity of the Al-Si structure.	99
Figure 2	Images showing examples of in-plane silicon plate instability. This instability leads to the development of in-plane protrusions which are shown in (a) and indicated by white arrows, and eventually results in the development of in-plane rod structures, as shown in (b) and (c). Images are from an Al-20 wt% Si alloy (a, b) and an Al-7 wt% Si alloy (c) and were solidified at 200 $\mu\text{m/s}$ (a) and 50 $\mu\text{m/s}$ (b, c). All samples prepared by deep etching in a 2.5% HCl -1.5% HNO_3 -1% HF reagent.	101
Figure 3	Observed out of plane instability formations in an Al-7 wt% Si alloy (c) and Al-20 wt% Si alloy (a, b, d-f): a) Columnar instability, b) Columnar instability with additional row instability, c) Grid instability, d) Separate grid and columnar instabilities on same plate, e) Advanced grid instability displaying interconnecting bands, and f) instability formation at multiple levels of advancement. Solidification velocities include 50 $\mu\text{m/s}$ (b), 100 $\mu\text{m/s}$ (d, e), 200 $\mu\text{m/s}$ (a), and 1000 $\mu\text{m/s}$ (c, f). Samples were prepared by deep-etching in a 2.5% HCl -1.5% HNO_3 -1% HF reagent (a-e) or NaOH solution (f).	102
Figure 4	Comparison of in-plane protrusion spacings measured on a hypoeutectic (7 wt%) and hypereutectic (20 wt%) Al-Si alloy. Error bars represent the 95% confidence interval of the mean.	103
Figure 5	Comparison of out-of-plane protrusion spacings measured on a hypoeutectic (7 wt%) and hypereutectic (20 wt%) Al-Si alloy. Error bars represent the 95% confidence interval of the mean.	104

Figure 6	Comparison of protrusion spacings in an Al-20 wt% Si alloy for in-plane and out-of-plane protrusions. Error bars represent the 95% confidence interval of the mean.	105
Figure 7	Al-Si interface shapes as specified by Magnin and Kurz (MK) (20) and Guzik and Kopycinski (GK) (21) for two different silicon contact angles. Calculated at 1000 $\mu\text{m/s}$	106
Figure 8	Comparison of a planar interface composition contour map and the Al-Si interface shape.	107
Figure 9	The marginal instability spacing calculated as a function of x in front of the silicon phase for both the Magnin-Kurz and Guzik-Kopycinski interface shapes. Calculations are for a solidification velocity of 1000 $\mu\text{m/s}$ and a silicon phase contact angle of 85° . Also shown is the marginal stability spacing calculated using the average composition gradient ahead of the silicon phase.	108
Figure 10	Near-equilibrium silicon shapes of an Al-20 wt% Si alloy held at 592 $^\circ\text{C}$ for 125 hours. Silicon particles approximate an octahedral shape, with triangular facets consistent with faceting on $\{111\}$ silicon planes. Particles were extracted after rapid quenching and deep-etching in a 2.5% HCl -1.5% HNO_3 -1% HF reagent.	109
Figure 11	Comparison of experimental silicon protrusion spacings and calculated minimum and maximum instability spacings with multiple values of anisotropy. Calculated values are based on the Magnin-Kurz(20) interface shape at a silicon phase contact angle of 85°	110
Figure 12	Effect of anisotropy and wetting angle on the calculated minimum and maximum instability wavelengths at a 1000 $\mu\text{m/s}$ solidification velocity. Results were obtained using the Magnin-Kurz (20) interface shape. . .	111
Figure 13	Effect of alloy composition on calculated marginal and maximum amplification rate instability spacings.	112

Figure 14	Planar interface stability range as a function of perturbation wavelength and solidification velocity. Inside the unstable region perturbations will grow with time and destabilize the planar interface. Higher anisotropy serves to decrease the extent of the instability range to a minor degree.	113
Figure 1	X-ray radiographs of the six tensile specimens showing flowline indications.	117
Figure 2	X-ray radiographs of the six coupon specimens showing flowline indications.	118
Figure 3	Representative microstructures of the test alloys (backscattered electron images): (a) WE43B (left), (b) EV31A (middle), and (c) ZE41 (right).	119
Figure 4	EDS linescan showing the variation in Zr content in the WE43B alloy (V19).	120
Figure 5	Optical micrographs of alloy WE43B with corresponding microhardness and composition profiles.	121
Figure 6	Backscattered electron image (left) of EV31A (sample A) and corresponding dot maps, showing Nd-rich grain boundary particles (center) and Zr-rich intragranular plates (right).	123
Figure 7	A high-magnification secondary electron image, showing a colony of Zr-rich fine plate precipitates in alloy EV31A (sample A).	124
Figure 8	Optical micrograph showing the structure of alloy EV31A and corresponding microhardness measurements. Note that the scale of the microhardness measurements is compressed when compared with that of the optical image. Only the flow line location is consistent.	125
Figure 9	Electron backscattered image (top) for alloy ZE41 (sample 3A) and corresponding dot maps showing the distribution of elements in the different phases.	126

Figure 10	Optical micrographs for alloy ZE41 (sample 3B) demonstrating a clear change in grain size across the sample. This change coincides with the location of the flow line, which is roughly in the middle of the upper image.	127
Figure 11	Microhardness and grain size measurements for alloy ZE41 (sample 3B). There is a noticeable change in grain size at the flow line, though this change in grain size does not significantly affect the measured hardness.	128
Figure 12	Microhardness, grain size, and composition measurements for alloy ZE41 (sample 3A). There is no noticeable change in hardness, grain size, or composition across the sample.	129
Figure 13	Uniaxial tension test results for all specimens.	130
Figure 14	Representative fracture surfaces of alloy WE43B (samples V17B and V17C).	131
Figure 15	Representative fracture surfaces of alloy EV31A (samples C and I). . .	132
Figure 16	Representative fracture surfaces of alloy ZE41 (samples 1 and 2). . . .	133
Figure 17	Representative fracture surfaces observed after fracture by three-point bending with a pre-machined groove at the flow-line location. The appearance of the fracture surface is similar to the tensile specimen fracture surfaces for all three test alloys.	134
Figure 1	Example of a minimum spanning tree constructed between a set of 11 points.	138
Figure 2	Schematic of PointGrid structure, which separates points into individual grid boxes based upon their location.	140
Figure 3	Pseudocode for the PointGrid algorithm used for calculation of the MST.	141
Figure 4	MST segment length mean comparison for ideal and MD simulated BCC and FCC structures.	146
Figure 5	MST angle mean comparison for ideal and MD simulated BCC and FCC structures.	146

Figure 6	MST junction frequency comparison for ideal and simulated BCC structures. Solid lines represent ideal structures while dashed lines are MD simulated structures.	147
Figure 7	MST junction frequency comparison for ideal and simulated FCC structures. Solid lines represent ideal structures while dashed lines are MD simulated structures.	147
Figure 8	MST junction length comparison for ideal and simulated BCC structures. Solid lines represent ideal structures while dashed lines are MD simulated structures.	148
Figure 9	MST junction length comparison for ideal and simulated FCC structures. Solid lines represent ideal structures while dashed lines are MD simulated structures.	148
Figure 10	Segment length mean versus standard deviation of different structures. The CuZr structures are simulated using MD at temperatures from 100–1800 K, BCC MD are simulated from 1000–2200 K, and FCC MD structures are simulated from 200–1400 K. The rest of the structures were generated with 0–50% noise added.	149
Figure 11	Segment angle mean versus standard deviation of different structures. The CuZr structures are simulated using MD.	150
Figure 12	Schematic of packing arrangement adopted to approximate an icosahedral structure.	151
Figure 13	Normalization length, segment length mean, and angle mean MST statistics obtained from a MD simulated CuZr structure quenched from 1800 K to a range of temperatures. Clear transition points are evident in the data, though the actual transition temperature value includes temperatures of 650 K (in black), to 700 K (in red), and 750 K (in orange).	151

Figure 14	MST segment length mean from a MD simulated CuZr structure without length scale normalization. The data display the same transition in slope at 700 K noted in the normalized data, though the data also display increased scatter.	152
Figure 15	MST segment length standard deviation versus mean of a MD simulated CuZr structure. The glass transition around 700 K is fairly faint but noticeable.	152
Figure 16	MST angle standard deviation versus mean of a MD simulated CuZr structure. The glass transition is readily apparent at 750 K, with the angle standard deviation versus mean relationship showing significantly different trends above and below this temperature.	153
Figure 17	MST junction type frequency of a MD simulated CuZr structure. Junction frequency decreases with junction type number from J1 up through J5. Additionally, the dominance of J1-type junctions increases at lower temperatures at the expense of J2 frequency.	154
Figure 18	MST segment length separated by junction type for a MD simulated CuZr structure. The glass transition at 700 K is only readily apparent in the J3-type junctions. The J3 junction data are shown separately at the right for clarity. The transition does not show up at all for J1 or J2 junctions, and may be faintly apparent if at all in the J4 and J5 junction types.	154
Figure 19	MST segment length separated by segment type and showing only the first three J3-type segments for a MD simulated CuZr structure. The glass transition near 700 K shows up in all three segment types. This transition does not show up in any segments not containing a J3 junction.	155

Figure 20	Frequency of MST segment types separated by junction atoms in a MD simulated CuZr structure. Copper to copper segments are significantly more frequent than zirconium to zirconium connections, which is likely due to the larger zirconium atom.	156
Figure 21	Length of MST segment types separated by junction atoms in a MD simulated CuZr structure. Due to atom size, Zr-Zr segments are the longest, followed by CuZr, and Cu-Cu segments are the shortest. Note the transition in the Cu-Cu data and the increase in Zr-Zr segment length at lower temperatures.	157
Figure 22	Frequency of MST segment types separated by junction atoms in a MD simulated CuZr structure of varying composition at 300 K. With increasing fraction of copper the prevalence of Zr-Zr segments plunges to near 0 at 50% copper.	158
Figure 23	Length of MST segment types separated by junction atoms in a MD simulated CuZr structure of varying composition at 300 K. Length of Zr-Zr segments increases linearly until disappearing above 57 atomic %Cu, while the length of CuZr segments begins to increase rapidly above 57% copper.	159
Figure 24	MST angle mean as a function of temperature in a MD simulated CuZr structure. The angles between copper atoms display a similar trend with temperature as the overall MST data, while the angles between zirconium atoms do not appear to display any meaningful trend. . . .	160
Figure 25	Schematic of pixel distance frequency around a central pixel. Each box represents a pixel of an image, and the numbers in each pixel represent the square of the distance between that pixel and the central pixel. Each distance occurs four times, except for 5, which occurs eight times.	164

Figure 26	Example calculation of the radial distribution function (RDF) demonstrating its effectiveness in determining average local spacing. The three peaks of the RDF function in this example correspond to the concentric circles drawn around the object being measured.	164
Figure 27	Cumulative probability distributions for the average spacing around a silicon particle for a range of solidification velocities.	165
Figure 28	Relationship of spacing versus velocity for the average particle spacing determined at three different cumulative probability values.	165
Figure 29	Critical diameter as a function of velocity. The measurement contains significant scatter and it is difficult to determine an overall trend. . . .	166

ABSTRACT

Continually rising energy prices have inspired increased interest in weight reduction in the automotive and aerospace industries, opening the door for the widespread use and development of lightweight structural materials. Chief among these materials are cast Al-Si and magnesium-based alloys. Utilization of Al-Si alloys depends on obtaining a modified fibrous microstructure in lieu of the intrinsic flake structure, a process which is incompletely understood. The local solidification conditions, mechanisms, and tensile properties associated with the flake to fiber growth mode transition in Al-Si eutectic alloys are investigated here using Bridgman type gradient-zone directional solidification. Resulting microstructures are examined through quantitative image analysis of two-dimensional sections and observation of deep-etched sections showing three-dimensional microstructural features. The transition was found to occur in two stages: an initial stage dominated by in-plane plate breakup and rod formation within the plane of the plate, and a second stage where the onset of out-of-plane silicon rod growth leads to the formation of an irregular fibrous structure. Several microstructural parameters were investigated in an attempt to quantify this transition, and it was found that the particle aspect ratio is effective in objectively identifying the onset and completion velocity of the flake to fiber transition. The appearance of intricate out-of-plane silicon instability formations was investigated by adapting a perturbed-interface stability analysis to the Al-Si system. Measurements of silicon equilibrium shape particles provided an estimate of the anisotropy of the solid Si/liquid Al-Si system and incorporation of this silicon anisotropy into the model was found to improve prediction of the instability length scale.

Magnesium alloys share many of the benefits of Al-Si alloys, with the added benefit of a 1/3 lower density and increased machinability. Magnesium castings often contain additions of

heavier elements, such as zinc, zirconium, and rare earth elements, which significantly improve high temperature performance. However, additions of these elements can lead to macrosegregational effects in castings, which are detectable by radiographic scans. The effect of these flow-line indications on alloy mechanical properties is not well quantified. An examination of these flow-line indications and their effects on mechanical properties in three magnesium-based casting alloys was performed here in order to determine the best practice for dealing with affected castings. Preliminary results suggest the flow-lines do not measurably impact bulk material properties.

Three additional methods of characterizing three-dimensional material structures are also presented: a minimum spanning tree analysis is utilized to quantify local structure in Cu-Zr liquid phase simulations obtained from molecular dynamics; the radial distribution function is applied to directionally solidified Al-Si structures in an attempt to extract local spacing data; and the critical diameter measurement is also defined and applied to irregular eutectic Al-Si structures.

INTRODUCTION

Introduction

Continually rising energy prices have inspired increased interest in weight reduction in the automotive and aerospace industries, opening the door for the widespread use and development of lightweight structural materials. Among the most common of these materials are alloys based upon the elements aluminum and magnesium. Both aluminum and magnesium are known for having high corrosion resistance, low density, and high relative abundance. Although extremely soft and ductile as pure metals, aluminum and magnesium can achieve significant strength levels when alloyed. Magnesium alloying additions often target improved strength and high temperature resilience through solid solution and precipitation hardening, often requiring heat treatments post casting. Aluminum, however, may be alloyed with silicon, forming the irregular eutectic Al-Si, which is commonly used in the as-cast condition, eliminating the need for extensive post-cast processing. Both alloys present intriguing possibilities as light-weight structural materials, with properties that depend strongly on the mechanisms occurring during the solidification process.

Al-Si irregular eutectic alloys are used for their superior castability, excellent corrosion resistance, low density, and good mechanical properties. The superior castability attributed to eutectic alloys are related to their relatively low solidification temperatures and small or nonexistent mushy zones (see Fig. 1). The excellent corrosion resistance is from surface passivation due to the strongly adherent oxide layers inherent to both aluminum and silicon. Similarly, both elements possess low density ($\rho_{Al} = 2.7g/cm^3$, $\rho_{Si} = 2.3g/cm^3$) leading to the corresponding lightweight nature of Al-Si alloys. While the traditional flake morphology

observed in pure Al-Si castings imparts poor mechanical properties, a significant improvement in strength and ductility is obtained with microstructural modification, which generally involves adding trace impurity elements to the alloy to elicit the growth of a more desirable fibrous morphology. Typical mechanical properties for Al-Si alloys range from 120-280 MPa tensile strength and elongations from 2-12%(1-3), allowing for the use of Al-Si alloys in numerous automotive and aerospace applications.

Most applications for aluminum casting alloys are based on the A300 series of aluminum alloys (from the Aluminum Association numbering system) which consist of aluminum alloyed with silicon and small amounts of copper or magnesium. A common alloy of this series is the A356 alloy, which contains 7% Si, 0.25% Cu, and 0.35% Mg. This alloy finds significant use in engine block castings, alloy wheels, and various other automotive and aircraft fittings. A drawback to this and other hypoeutectic Al-Si alloys is their mediocre wear resistance. In many applications this can be countered by the use of a wear liner made of cast iron or some other material around the lightweight hypoeutectic Al-Si body. An alternative is the use of hypereutectic Al-Si alloys which contain large primary silicon particles that significantly improve wear resistance. The development of hypereutectic alloys such as A390 which contains 17% silicon has allowed the production of full aluminum engine blocks without the need for liners. The performance of these industrial alloys depends strongly on both alloy composition and microstructural modification.

Microstructural modification in Al-Si alloys is the process by which the eutectic silicon microstructure is transformed from a bulky flake structure to a fine fibrous structure with an attendant improvement in both strength and ductility. In pure Al-Si castings the typical microstructure is one with large interconnected silicon plates which is predictably not favorable for mechanical properties. This microstructure may be altered to a finer fibrous morphology with additions of trace impurity elements such as sodium and strontium or with an increase in solidification velocity, known as impurity modification and quench modification, respectively. Quench modification and impurity modification occur by fundamentally different mechanisms, as ascribed to by different respective structures on the microscopic scale: quench modified

fibers are microscopically smooth and globular, while impurity modified fibers are microfaceted with a high twin density. Impurity modification is the most common modification procedure used in foundries due to the ability to achieve relatively uniform microstructures throughout a bulk casting. However, the use of impurity modification is associated with significant adverse effects such as increased porosity, hot tearing, and poor surface finish. The use of quench modification to achieve a fibrous structure may avoid these drawbacks, though the onset and degree of quench modification as a function of melt conditions are not well understood which limits the prediction of eutectic microstructure at different locations within a casting.

Magnesium alloys share many of the benefits of Al-Si alloys, with the added benefits of a 1/3 lower density ($1.7\text{--}1.9\text{ g/cm}^3$) and increased machinability. While magnesium alloys often cost more than Al-Si alloys, their extreme weight saving benefits are of use in high-demand applications such as aeroengines, helicopter transmissions, and racing and high performance cars. The alloys that will be focused on here are WE43B, EV31A, and ZE41. These alloys include additions of heavier metals, such as zinc, zirconium, and rare earth elements, which significantly improve high temperature performance. However, the addition of these heavier elements can also lead to the development of segregational flow lines in castings, which are detectable in x-ray radiographs of cast sections. The effects of these flowlines on the mechanical properties of the alloy are not well understood, therefore requiring castings that contain flowlines to be discarded, which raises the overall cost of producing these alloys. More thorough characterization of the effects of these flowlines on tensile properties could allow for a more economic treatment of affected castings.

The primary advantage of Al-Si and magnesium casting alloys in commercial applications is weight reduction when compared to alternative material options. Weight reduction in automotive applications leads to a direct improvement in vehicle fuel efficiency, which is becoming increasingly important due to rising energy costs and stricter fuel economy requirements. As a result, utilization of weight-reducing alloys is expected to increase in the near future. The efficient adoption of Al-Si alloys to new applications requires a detailed knowledge of the solidification patterns of the alloy system, allowing for rapid and adequate prediction of the eutectic

Figure 1 Phase diagram of the Al-Si system.(4)

microstructure throughout the casting. However, solidification models of Al-Si alloys are incomplete and lacking in the area of microstructure prediction. Specifically, the occurrence of the flake to fiber transition in terms of applied melt conditions is not well understood. A more complete understanding of the solidification process in Al-Si alloys based upon the response of the system to extrinsic melt parameters is required in order to facilitate the adaptation of the alloy to the numerous applications that would benefit from the inherent advantages of the Al-Si system. Additionally, increased knowledge of the effects of segregational flowlines on tensile properties in heavy metal containing magnesium alloys is required to determine the best practice for dealing with flowline-containing castings.

Dissertation Organization

This dissertation is written in an alternate format and consists of one published paper and three original manuscripts. References are presented separately following each chapter, and table and figure numbering is also reset between chapters. A general introduction and literature review are contained in this, the first chapter, and a general conclusion is presented in the last chapter. Additional material includes a chapter containing three methods of microstructural characterization not covered in the complete manuscripts, and an appendix listing the computer code used in the calculation methods presented within this document.

Following the literature review are three manuscripts dealing with the Al-Si system. First, in “Analysis of the high growth-rate transition in Al-Si eutectic solidification”, an in-depth investigation of the flake to fiber transition (quench modification) in a eutectic composition Al-Si alloy is made, with an emphasis on elucidating the growth mechanisms occurring during the transition, and searching for quantitative measurement parameters that can be used to identify the flake to fiber transition. Following that, in “The effect of the flake-fiber transition in silicon morphology on the tensile properties of Al-Si eutectic alloys”, the flake to fiber transition is investigated with regards to its effect on bulk tensile properties and fracture behavior, with correlations made between observed tensile properties and the predicted morphology based upon the quantitative measurement parameters specified in the first paper. Then, in “Morphological instability of silicon during directional solidification of an Al-Si eutectic alloy”, observations are made regarding the appearance of extensive out-of-plane silicon instability formations, and a perturbed interface instability analysis is adapted to the Al-Si system in an attempt to model the length scale behavior of protrusion formation with the goal of highlighting some of the basic mechanisms operating during the Al-Si flake to fiber transition.

Following the three manuscripts concerning the Al-Si system is a manuscript dealing with magnesium casting alloys, “Metallurgical analysis of flow-line indications in Mg-based alloy castings”, where the effects of segregational flowlines in magnesium alloys WE43B, EV31A, and ZE41 are investigated. The investigation includes determination of microstructure, chemical composition, tensile properties, and microhardness. After this is a special chapter, “Addi-

tional analyses of three-dimensional microstructures”, which presents three additional methods for microstructural characterization, namely the minimum spanning tree, radial distribution function, and critical diameter. These methods are grouped together as they have not been incorporated into a complete manuscript.

Literature Review

Eutectic Solidification Theory

A eutectic reaction involves the isothermal transformation of a homogeneous liquid solution to two separate solid phases. Solidification of a eutectic alloy proceeds via the formation of a diffusion couple with the two solid phases growing in close proximity in an alternating arrangement which allows for the most efficient transport of solute (Fig. 2). The ideal phase arrangement will then be either a regularly spaced array of lamellae or rods depending on the respective volume fraction of the phases. Eutectic systems which adopt this phase arrangement are aptly termed *regular* eutectics due to their relatively invariant microstructural features. However, large variations in the interfacial properties of the components of a eutectic can cause the system to deviate wildly from this idealized form of regular growth.

The ability of a eutectic alloy to grow in a regular manner depends on the interface characteristics of its constituent phases. Crystal structures that consist of mostly isotropic bonding tend to form atomically rough or diffuse interfaces due to the lack of a strongly preferred crystallographic surface orientation. These rough interfaces, also known as nonfaceted interfaces, contain many steps and ledges at the surface that allow for easy attachment of atoms migrating from the liquid phase. Conversely, crystal structures with highly directional bonding tend to form an atomically smooth interface with facets along preferred crystallographic orientations, Fig. 3. The atomically smooth interface does not allow for easy attachment of atoms from the liquid, as individual atoms that attach to a smooth interface will significantly increase the number of broken bonds locally contributing to an unfavorable high energy situation. Such atoms are likely to jump back to the liquid, restricting the growth of the solid in that direction. Migration of faceted interfaces requires the activation of a ledge-step propagation mechanism

Figure 2 Schematic of eutectic solidification. The periodic arrangement of phases minimizes average diffusion distance, thus maximizing the efficiency of solute transport.

whereby individual growth steps are generated normal to the surface and easy atomic attachment at the step interface ensues until the superficial layer is complete. This mechanism is much more restrictive than the continuous growth of a rough interface, ultimately limiting the ability of the faceted material to adjust its growth direction to accommodate changing melt conditions during solidification. Accordingly, small local variations in extrinsic melt parameters, such as temperature or composition, will not elicit a concomitant ameliorating response in the faceted material, preventing the attainment of the idealized regular structure. Instead, the structure of a eutectic consisting of both a nonfaceted and a faceted component is *irregular*, exhibiting widely variant local morphologies and spacings, Fig. 4.

Figure 3 Shape of (a) nonfaceted and (b) faceted interfaces. From Kurz and Fisher.(5)

Figure 4 Comparison of structure in a regular eutectic (left) and irregular eutectic (right). The regular eutectic displays a relatively constant phase spacing and morphology while the irregular eutectic structure contains extreme variation in local spacing and features. Images from England.(6)

Regular Eutectic Growth

Significant success has been obtained in modeling the growth of regular eutectic systems. Jackson and Hunt(7), building upon the work by Zener(8), Brandt(9), Tiller(10), and Hillert(11), developed a comprehensive theory for the growth of lamellar and rod eutectic systems. They proposed that fine adjustments to the eutectic spacing are attained via motion of lamellar faults (Fig. 5) while catastrophic microstructural changes can occur through the formation of instabilities.

Contributions to Undercooling The energy required to drive the solidification process is manifested as an undercooling, or departure from equilibrium. In general, the undercooling at the solid-liquid interface can be considered as a sum of three energy barriers: a diffusive barrier, a capillary barrier, and a kinetic barrier.

$$\Delta T_{\text{Tot}} = \Delta T_{\text{Diff}} + \Delta T_{\text{Cap}} + \Delta T_{\text{Kin}} \quad (1)$$

The kinetic barrier arises from the difficulty of a migrating atom to find a suitable attachment site on the surface of the solid phase. This barrier is often ignored to simplify undercooling calculations as atomic attachment to a rough interface as is found in the components of a regular eutectic is not a limiting process. Therefore, the interplay between solute diffusion and interfacial energy minimization are considered as the primary processes determining the growth characteristics of the eutectic microstructure. The diffusive barrier, also known as the constitutional undercooling, arises from the compositional departure from the eutectic composition on the interface.

$$\Delta T_{\text{Diff}} = m(C_E - C(x)) \quad (2)$$

where m is the slope of the liquidus line, C_E is the eutectic composition, and $C(x)$ is the composition with respect to point x on the interface. The capillary barrier depends on the surface free energy and interfacial curvature.

$$\Delta T_{\text{Cap}} = \frac{\Gamma}{r(x)} \quad (3)$$

Figure 5 Schematic of a lamellar fault, as proposed by Jackson and Hunt.(7) Relative motion of the lamellar fault to the right will increase the average spacing of the system, while motion to the left will decrease the average spacing.

where Γ is the Gibbs-Thompson coefficient and $r(x)$ is the radius of curvature with respect to point x on the interface. Although the individual undercooling contributions from the diffusive and capillary barriers vary across the interface, their sum is constant. Therefore, the average interfacial composition and curvature are sufficient to determine the total interfacial undercooling.

Diffusion Fick's Laws

The diffusion of atoms plays a major role in many phase transitions, including the problem of directional solidification, where both solute and heat diffusion are important processes. In its simplest form diffusion can be envisioned as a random jump process. Consider the case of interstitial diffusion: the motion of an individual atom depends on the frequency of making successful jumps into unoccupied lattice positions. The probability of making a successful jump decreases with increasing composition. Therefore, if a composition gradient exists in a

material atomic jumps will be more successful in the direction leading down the composition gradient. This is the essence of Fick's first law of diffusion, which states:

$$J = -D \frac{\partial C}{\partial x} \quad (4)$$

where J is the flux in atoms/m² and D is the diffusivity in m²/s. This equation accounts for steady state diffusion where the concentration profile of the diffusing species is not changing with time. However, a time dependent concentration model is often more appropriate. This can be obtained by applying a mass balance criterion based upon the difference of atomic flux into and out of an area:

$$\frac{\partial C}{\partial t} = -\frac{\partial J}{\partial x}$$

Fick's law can be substituted for the flux to obtain:

$$\frac{\partial C}{\partial t} = -\frac{\partial}{\partial x} \left(-D \frac{\partial C}{\partial x} \right)$$

which is known as Fick's second law. A common assumption is that the diffusion coefficient does not vary with composition, or:

$$\frac{\partial C}{\partial t} = D \frac{\partial^2 C}{\partial x^2}$$

This equation can also be stated in three dimensions,

$$\frac{\partial C}{\partial t} = D \frac{\partial^2 C}{\partial x^2} + D \frac{\partial^2 C}{\partial y^2} + D \frac{\partial^2 C}{\partial z^2} \quad (5)$$

Directional Growth Equation

Fick's second law, Eq. 5, gives a useful relation for the diffusion of species through a single point in a material with time. However, solidification problems require the description of diffusion along a moving interface. To accommodate the interface motion, Eq. 5 must be transformed from a stationary coordinate system to one that is moving with the interface, at velocity V . The relation between a stationary coordinate system, (y', z') , and a coordinate system moving in the z direction with velocity V , (y, z) , is:

$$(x, y, z) = (x', y', z' - Vt) \quad (6)$$

The following relations can be derived from Eq. 6:

$$\frac{\partial z}{\partial z'} = 1, \quad \frac{\partial^2 z}{\partial z'^2} = 1$$

which leads to:

$$\left(\frac{\partial^2 C}{\partial z^2} \right) \left(\frac{\partial z^2}{\partial z'^2} \right) = \frac{\partial^2 C}{\partial z'^2} = \frac{\partial^2 C}{\partial z^2} \quad (7)$$

Therefore, applying the coordinate transformation in Eq. 6 to Eq. 5 leaves the right hand side of the equation unchanged. As the coordinate system advances, conservation of mass is obtained by subtracting the rate of solute influx from the diffusion equation, giving:

$$\frac{\partial C}{\partial t} \rightarrow \frac{\partial C}{\partial t} - V \frac{\partial C}{\partial z}$$

Finally, considering that composition in the x direction is constant and overall composition is unchanging with time at steady state, the diffusion equation becomes:

$$-V \frac{\partial C}{\partial z} = D \frac{\partial^2 C}{\partial y^2} + D \frac{\partial^2 C}{\partial z^2}$$

Rearranging terms leads to the traditional form of the directional growth equation:

$$D \frac{\partial^2 C}{\partial y^2} + D \frac{\partial^2 C}{\partial z^2} + \frac{V}{D} \frac{\partial C}{\partial z} = 0 \quad (8)$$

Planar Diffusion Solution

In order to model the complex processes occurring during solidification several simplifying assumptions are made. The present consideration involves a lamellar eutectic with a constant phase spacing. The material is assumed to be uniform in one direction (x axis) and is undergoing unidirectional solidification with the interface advancing at a constant velocity in the z direction. The interface is also assumed to be at steady state, which implies that all diffusion profiles and phase morphologies are independent of time. Further, the interface morphology is defined as planar, and boundary conditions are defined so that the composition approaches a constant value at some distance ahead of the interface and the composition gradient is zero in the middle of each phase. The applicable boundary conditions for the advancement of a planar interface at steady state are then:

$$\begin{array}{lll} C = C_E + C_\infty & at & z = \infty \\ \frac{\partial C}{\partial x} = 0 & at & x = S_\alpha \quad \text{and} \quad x = 2S_\alpha + S_\beta \end{array}$$

where S_α and S_β are half-phase widths as defined in Fig. 6, C_E is the eutectic composition, and C_∞ is the difference between the eutectic composition and the actual composition far from the interface. The conservation of matter at the interface requires:

$$\begin{aligned} \left(\frac{\partial C}{\partial z}\right)_{z=0} &= -\frac{VC_0^\alpha}{D} & S_\alpha \leq x < 2S_\alpha \\ \left(\frac{\partial C}{\partial z}\right)_{z=0} &= \frac{VC_0^\beta}{D} & 2S_\alpha \leq x < 2S_\alpha + S_\beta \end{aligned}$$

where C_0^α and C_0^β are defined in Fig. 7. A solution to the directional growth equation with the given boundary parameters was presented by Jackson and Hunt(7):

$$C = C_E + C_\infty + B_0 \exp\left(-\frac{V}{D}z\right) + \sum_{n=1}^{\infty} B_n \cos\left(\frac{2\pi n}{\lambda}(x - S_\alpha)\right) \exp\left(-\frac{2\pi n}{\lambda}z\right) \quad (9)$$

where:

$$\begin{aligned} B_0 &= C_0^\alpha f_\alpha - C_0^\beta f_\beta \\ B_n &= \frac{\lambda}{(n\pi)^2} \frac{V}{D} C_0 \sin(n\pi f_\alpha) \end{aligned}$$

This solution is valid when:

$$\frac{4\pi n}{\lambda} \gg \frac{V}{D}$$

The average composition in the liquid in front of the α phase is obtained by integrating Eq. 9 over the interval 0 to S_α and dividing by S_α :

$$\bar{C}_\alpha = C_E + C_\infty + B_0 + \frac{1}{S_\alpha} \int_0^{S_\alpha} \sum_{n=1}^{\infty} B_n \cos \frac{2\pi n}{\lambda} dx$$

which yields:

$$\bar{C}_\alpha = C_E + C_\infty + B_0 + \frac{\lambda}{f_\alpha} \frac{V}{D} C_0 P$$

where P is a function of the phase fraction and is given by:

$$P = \sum_{n=1}^{\infty} \left(\frac{1}{\pi n}\right)^3 \sin^2(\pi n f_\alpha)$$

Similarly, for the β phase:

$$\bar{C}_\beta = C_E + C_\infty + B_0 - \frac{\lambda}{f_\alpha} \frac{V}{D} C_0 P$$

Figure 6 Planar interface morphology assumed in the calculation of the planar diffusion solution.

Figure 7 Phase diagram showing definition of several pertinent constants.

Figure 8 Interface morphology assumed for the calculation of capillary undercooling showing definition of S and θ .

Curvature To obtain the capillary undercooling of the interface, a curved interface shape is assumed as shown in Fig. 8. If the origin is taken to be the center of a lamellae, where the slope of the interface is 0, the average curvature can be calculated by integration of the lamellae shape:

$$\bar{\kappa} = \frac{1}{S} \int_0^S \kappa(x) dx$$

The definition of curvature states:

$$\kappa = \frac{-\frac{d^2 y}{dx^2}}{\left(1 + \left(\frac{dy}{dx}\right)^2\right)^{\frac{3}{2}}}$$

The integral can be solved to yield:

$$\bar{\kappa} = \frac{1}{S} \sin \theta$$

where S and θ are defined in Fig. 8.

Total Undercooling The average composition and curvature of the interface can be used to determine the diffusive and capillary undercoolings:

$$\Delta T_{\text{Diff}}^{\alpha} = m_{\alpha} \left(C_{\infty} + B_0 + \frac{\lambda}{f_{\alpha}} \frac{V}{D} C_0 P \right) \quad (10a)$$

$$\Delta T_{\text{Diff}}^{\beta} = m_{\beta} \left(-C_{\infty} + B_0 + \frac{\lambda}{f_{\beta}} \frac{V}{D} C_0 P \right) \quad (10b)$$

$$\Delta T_{\text{Cap}}^{\alpha} = \frac{\Gamma_{\alpha} \sin \theta_{\alpha}}{S_{\alpha}} \quad (10c)$$

$$\Delta T_{\text{Cap}}^{\beta} = \frac{\Gamma_{\beta} \sin \theta_{\beta}}{S_{\beta}} \quad (10d)$$

The total undercooling is then obtained by substituting the undercooling from the individual contributions, Eq. 10, into Eq. 1. Then, the fact that the undercooling ahead of each phase is equal leads to the well known undercooling relation:

$$\Delta T = K_1 V \lambda + \frac{K_2}{\lambda} \quad (11)$$

where:

$$K_1 = \frac{m P C_0}{f_{\alpha} f_{\beta} D}$$

$$K_2 = 2m \left(\frac{\Gamma_{\alpha} \sin \theta_{\alpha}}{f_{\alpha} m_{\alpha}} + \frac{\Gamma_{\beta} \sin \theta_{\beta}}{f_{\beta} m_{\beta}} \right)$$

A schematic of this undercooling relation is shown for a fixed velocity in Fig. 9. At small spacings, the undercooling is dominated by the capillary term, while at large spacings the diffusive term dominates. However, an additional condition is needed to discern the actual operating spacing-undercooling state chosen by the system. Zener(8) proposed for regular eutectics that growth occurs at the extremum, or minimum undercooling, an assertion supported by Jackson and Hunt(7). This condition makes the calculation of the following undercooling relationships possible by taking the derivative of the undercooling with respect to λ and setting equal to 0:

$$\lambda^2 V = \frac{K_2}{K_1}$$

$$\frac{\Delta T^2}{V} = 4 K_1 K_2$$

$$\Delta T \lambda = 2 K_2$$

Jackson and Hunt also solved the diffusion equation for the case of a rod eutectic. The derivation is similar to that for lamellar growth except the solution is obtained using a cylindrical Bessel function series instead of a circular cosine series. The solution for rod growth yields the same relation as Eq. 11 except the constants are defined as:

$$K_1 = \frac{2mC_0M}{f_\beta D}$$

$$K_2 = 4m\sqrt{f_\alpha} \left(\frac{\Gamma_\alpha^R \sin \theta_\alpha^R}{f_\alpha m_\alpha} + \frac{\Gamma_\beta^R \sin \theta_\beta^R}{(1-f_\alpha)m_\beta} \right)$$

where:

$$M = \sum_{n=1}^{\infty} \frac{1}{\gamma_n^3} \frac{J_1^2 f_\alpha \gamma_n}{J_0^2 \gamma_n}$$

The regular eutectic system is thus seen as operating under a minimum undercooling criterion whereby small spacing adjustments are continually made to conform to minor changes in the local solidification conditions. Furthermore, major changes to the local solidification conditions may give rise to catastrophic changes due to instabilities. In both cases the result is the maintenance of the steady state lamellar or rod interface with a relatively constant phase spacing. Jackson and Hunt's model has been found to work well for prediction of regular eutectic growth, though its application to irregular eutectics requires some consideration.

Irregular Eutectic Growth

The modeling of irregular eutectic growth contains a number of additional difficulties that are not present for regular eutectics. These include i) the assumption that kinetic undercooling is negligible is not necessarily true for irregular eutectic systems, ii) irregular eutectics grow with a significantly nonisothermal interface which drastically complicates any exact mathematical analysis of the growth front, and iii) irregular eutectics display a widely variable local spacing and flake orientation which does not conform to a the minimum undercooling criterion. These issues must be addressed in order to develop new models or extend existing models for regular eutectics to adequately characterize irregular eutectic growth.

The contribution of kinetic undercooling was addressed by Steen and Hellawell(12) for the Al-Si system. In order to estimate the kinetic undercooling for an Al-Si alloy, they used Jack-

Figure 9 Schematic showing the relation between spacing and total undercooling. The main undercooling contributor at low spacings is the capillary undercooling, while the diffusional undercooling dominates at large spacings. Regular eutectics tend to grow at or near the extremum point (minimum undercooling) while irregular eutectics grow at larger spacing values.

son's model for the free energy change upon addition or subtraction of a monolayer from a smooth crystal face(13) and estimated the kinetic undercooling to be $\approx 1\%$ of the total undercooling at a growth rate of 1 mm/s, and therefore negligible. This indicates that treatment of the kinetic undercooling may not be necessary to obtain adequate growth models of irregular eutectic systems such as Al-Si.

The lack of an isothermal interface significantly complicates the calculation of the diffusion solution for irregular eutectic growth. One of the first attempts to treat an isothermal interface was that of Sato and Sayama(14) who postulated that if a depression forms at the center of the major phase then only the parts of the major phase interface that remain isothermal with the minor phase need to be modeled. Their interface morphology assumption is shown in Fig. 10b. This assumption altered the concentration field ahead of the interface as well as the average interface curvature. The result is an equation similar to Eq. 11 with different constant values, which can yield up to 50% higher total undercooling for Al-Si than the Jackson-Hunt analysis(15).

The operating state of irregular eutectics does not follow the simple minimum undercooling criterion. In addition to estimating the kinetic undercooling of an Al-Si alloy, Steen and Hellawell(12) also estimated the value of interfacial undercooling, and also found it to be significantly smaller than the total undercooling, which indicates that constitutional undercooling is the primary contributor to the total undercooling. This result supports the experimental observations that irregular eutectics display a larger average spacing than regular eutectics solidified under comparable conditions, as the operating point of the undercooling-spacing relationship is far to the right in irregular eutectics (Fig. 9) where most of the undercooling is due to the constitutional term.

Instead of the minimum undercooling criterion, the irregular eutectic system can be envisioned as operating within a range of local spacings which is stabilized by the difficulty of the faceted phase to change growth directions. Implementation of this operating range along with the concept of a nonisothermal interface has led to several modifications to the regular eutectic model. A stability criterion was developed by Fisher and Kurz(15) that describes the

operating state of an irregular system in terms of a minimum and maximum local spacing which define the bounds of the eutectic growth. With this criterion, converging fibers are expected to grow until interfacial effects cause one of the fibers to halt its growth, while diverging fibers are expected to grow until increased local supersaturation causes branching of one of the fibers. Branching was assumed to occur as a result of a depression in the faceted phase interface due to a Mullins and Sekerka(16) type instability (to be discussed in detail later). As two fibers diverge, they are expected to thicken laterally to maintain relative phase fraction. The interface instability is assumed to occur when the minor phase width reaches 1.5 times the minimum stable perturbation wavelength, or $2S_\beta = 1.5\lambda_{crit}$. The value of 1.5 was chosen since the phase width approaches the instability condition from below. Further, by assuming thermal conductivities and temperature gradients of the solid and liquid are similar, that the partition coefficient, k , is equal to 0, and that the temperature gradient does not have much of an effect on the wave stability, they arrived at the relation:

$$\lambda_{crit} = 2\pi\sqrt{\frac{\Gamma}{mG_c}} \quad (12)$$

where m is the slope of the liquidus line, Γ is the Gibbs-Thompson coefficient, and G_c is the composition gradient in front of the liquid. The maximum eutectic spacing, λ_{branch} , is then:

$$\lambda_{branch} = \frac{3\lambda_{crit}}{2f_\beta} \quad (13)$$

The maximum (branching) spacing is then obtained by substituting Eq. 12 into Eq. 13, which can then be substituted into Eq. 11 to obtain a new undercooling-spacing relation:

$$\Delta T = \frac{K_3 V}{\sqrt{G_c}} + K_4 \sqrt{G_c} \quad (14)$$

where:

$$K_3 = \frac{3\pi PC_0 \sqrt{m\Gamma_\beta}}{f_\alpha f_\beta^2 D}$$

$$K_4 = \frac{2f_\beta m^{3/2}}{3\pi \sqrt{\Gamma_\beta}} \left(\frac{\Gamma_\alpha \sin \theta_\alpha}{f_\alpha m_\alpha} + \frac{\Gamma_\beta \sin \theta_\beta}{f_\beta m_\beta} \right)$$

It was also noted that the composition gradient is proportional to velocity, which allows for the substitution $\sqrt{G_c} = K\sqrt{V}$. With this change, Eq. 14 can be restated as:

$$\frac{\Delta T^2}{V} = K_5$$

Figure 10 Interface morphologies assumed by (a) Jackson and Hunt(7), (b) Sato and Sayama(14), and (c) Fisher and Kurz(15).

This gives the same relationship as the Jackson-Hunt analysis(7) without using the minimum undercooling criterion, though it should be noted that Eq. 12 is not generally true and G_c is also a function of λ .

Fisher and Kurz(15) incorporated their stability criterion into a coherent growth model along with considerations for a nonisothermal interface and the corresponding effect of the temperature gradient on the irregular interface. They assumed the temperature gradient is uniform and is a linear function of position in the melt. In this manner, the temperature of the interface is a simple function of position in the melt, or interface shape. This approach considers the depressed portion of the major phase in the analysis, unlike the corresponding treatments from Jackson and Hunt(7) and Sato and Sayama(14). This coupling condition was satisfied at the bottom of the major phase depression and in an average fashion along the non-depressed portion of the interface (0 to $S_\alpha + w$ in Fig. 10c). However, their treatment resulted only in a numerical solution to the growth problem which is difficult to handle and ultimately impractical.

Magnin and Kurz(17) developed a model for irregular eutectic growth similar to that of Fisher and Kurz(15) except solvable by analytical means rather than numerical. In this model a nonisothermal condition was adopted over the entire interface which allows for an analytical solution rather than numerical. The nonisothermal coupling condition equates the average

undercooling in front of each phase, or:

$$\Delta\bar{T} = \frac{1}{S_\alpha} \int_0^{S_\alpha} [\Delta T_{\text{Diff}}(x) + \Delta T_{\text{Cap}}(x) + GI(x)] dx = \frac{1}{S_\beta} \int_{-S_\beta}^0 [\Delta T_{\text{Diff}}(x) + \Delta T_{\text{Cap}}(x) + GI(x)] dx$$

where I is a cubic function representing the shape of the interface.

Magnin and Kurz(17) used the planar diffusion model of Jackson and Hunt(7) to calculate constitutional undercooling and calculated the capillary undercooling from the curvature of I . This led to a modified version of the Jackson and Hunt undercooling equation:

$$\Delta T = K_1 \lambda V + \frac{K_2}{\lambda} + F$$

where K_1 and K_2 are the same constants in the Jackson and Hunt analysis and F is a correction factor that reflects the nonisothermal nature of the assumed interface, given by:

$$F = -mG \left(\frac{\frac{\delta_\alpha}{2} - \frac{\lambda f_\alpha \tan \theta_\alpha}{24}}{|m_\alpha|} + \frac{\frac{\delta_\beta}{2} - \frac{\lambda f_\beta \tan \theta_\beta}{24}}{m_\beta} \right)$$

where δ is the depression depth of each phase and is given by:

$$\delta_i = \frac{2f_i^2}{48\Gamma_i + f_i^2\lambda^2G} \left[\left(\frac{|m_i|C_0}{D} \left(\Pi_i - \frac{P}{f_i} \right) V - \frac{f_i \tan \theta_i}{24} G \right) \lambda^3 - \frac{2\Gamma_i}{f_i} (2 \tan \theta_i + \sin \theta_i) \lambda \right] \quad i = \alpha, \beta$$

and Π depends on the phase fraction as shown:

$$\Pi_i = \sum_{n=1}^{\infty} \frac{1}{(n\pi)^2} \sin(n\pi f_i) \quad i = \alpha, \beta$$

The correction imposed by the gradient term F is only significant at very low solidification velocities ($< 0.1\mu\text{m/s}$). This was later rationalized to be due to the small scale of observed thermal fluctuations in the interface when compared with the diffusion length.(18) In order to determine the system operating point, Magnin and Kurz used a characteristic depression depth to determine the point of branching rather than a simple stability criterion as used by Fisher and Kurz(15). In order to describe the operating range, they introduced a parameter, ϕ , which relates the average spacing to the minimum spacing:

$$\phi = \frac{\bar{\lambda}}{\lambda_{ex}} = \frac{\lambda_{ex} + \lambda br}{2\lambda_{ex}}$$

Experimental observations allowed the determination of a depression depth range over which branching always occurs which was found to be between the point where the depression begins

at the center of the faceted phase (curvature = 0) and the point where the depression center reaches the level of the three phase junction. The operating range parameter ϕ was then theoretically predicted to be:

$$\phi = \frac{1}{2} + \left[\left(\frac{f_\beta \pi_\beta}{P} - 1 \right) \left(f_\alpha + f_\beta \frac{m_\beta \Gamma_\alpha \sin \theta_\alpha}{|m_\alpha| \Gamma_\beta \sin \theta_\beta} \right) \right]^{-\frac{1}{2}}$$

The parameter ϕ is effectively constant at normal growth rates ($> 0.1 \mu\text{m/s}$), therefore irregular eutectic growth can be described by the following relations:

$$\begin{aligned} \lambda^2 V &= \phi^2 \frac{K_2}{K_1} \\ \frac{\Delta T^2}{V} &= \left(\phi + \frac{1}{\phi} \right)^2 K_1 K_2 \\ \lambda \Delta T &= (\phi^2 + 1) K_2 \end{aligned}$$

Magnin and Trivedi(19) presented a modified form of the Jackson and Hunt(7) eutectic growth theory in which they accounted for the densities of the eutectic phases and imposed a three-phase junction equilibrium criterion in lieu of the Jackson and Hunt isothermal interface assumption. Noting that the Jackson and Hunt theory is only valid for eutectic phases of equal density, the following density correction to the C_0 term was derived:

$$C_0^* = \frac{\rho_\alpha(C_e - C_\alpha^0) + \rho_\beta(C_\beta^0 - C_e)}{\bar{\rho}}$$

This value is substituted for C_0 in Eq. 11 to obtain the modified result. The density correction factor plays a major role in eutectics such as Fe-C, but has less of an effect when applied to eutectic systems that contain components with similar densities, such as Al-Si.

Magnin, Mason, and Trivedi(18) found that experimentally determined spacings and undercoolings correspond well to a point on the theoretically determined λ - ΔT graph to the right of the extremum. The experimental and theoretical results were obtained independently which is provided as justification for the use of the average operating range parameter, ϕ , as a physical growth mechanism rather than an arbitrary fitting parameter. A parameter, η , was defined as the operating range extent, specifically:

$$\eta = \frac{\lambda_{branch} - \lambda_{min}}{\bar{\lambda}}$$

The variable spacings within irregular eutectic structures were suggested to be a result of locally different velocities with significant local departures from steady state rather than locally variant undercoolings. It was also suggested that the minimum spacings observed in irregular eutectics are larger than the extremum spacing value, contrary to general belief.

Growth Mechanisms of Al-Si Eutectic Alloys

While the periodic and relatively predictable growth characteristics of regular eutectic alloys allow for accurate predictions of final eutectic morphology, the inherent anisotropy and unpredictability present in irregular eutectics give rise to a diverse array of growth mechanisms, morphologies, and properties which vastly increases their commercial applicability. Among the most industrially useful of these alloys is Al-Si, whose high strength to weight ratio, good corrosion resistance, and excellent melt fluidity contribute to its wide use in high performance castings. In order to obtain these desirable properties the conditions of alloy solidification must be carefully controlled. Specifically, the intrinsic structure of a pure Al-Si alloy obtained at low solidification rates is a coarse flake morphology which imparts poor mechanical properties, while the superior mechanical strength of an industrial Al-Si alloy is obtained through *modification* of the intrinsic structure. Modification involves the transformation of the silicon flake structure to a globular or fibrous morphology with an attendant 50% improvement in tensile strength and threefold improvement in ductility.(1–3; 20) This microstructural alteration may be obtained via impurity additions to the melt or high rate solidification, processes known as impurity modification and quench modification, respectively. The mechanisms leading to both forms of modification appear to be fundamentally different and are incompletely understood. Any attempts to understand the complex processes occurring during modification must first begin with a thorough understanding of the solidification behavior of the unmodified alloy.

Normal (Unmodified) Al-Si Growth

In normal Al-Si alloys, the anisotropic nature of silicon prevents continuous growth; instead the silicon solidifies in the form of flakes with broad faces parallel to $\{111\}$ directions and

containing multiple $\{111\}$ twin traces.(21) Growth of the silicon plates must occur by a ledge-step mechanism whereby the presence of growth ledges leads to continuous lateral propagation of atomic planes resulting in stepwise plate growth perpendicular to the plane of the plate. The origin of sufficient growth ledges on the relatively low energy $\{111\}$ silicon planes may be due to the presence of multiple twins which give rise to self-perpetuating re-entrant edges and grooves as described in the twin plane re-entrant edge mechanism (TPRE), identified by Wagner(22) and Hamilton and Seidensticker(23). Alternatively, the presence of intrinsic steps on the silicon plane face may override the TPRE mechanism due to the relatively large average spacing between silicon twins, making the TPRE mechanism incidental to plate growth.(24) Regardless, multiple twinning is important in that it allows a single nucleation event to give rise to a wide range of orientations in the resulting microstructure through branching.

Branching of eutectic silicon is possible via both large angle and small angle branching modes, illustrated in Fig. 11.(25; 26). Large angle branching involves the formation of a new twin orientation on the face of a growing plate. Once two parallel twin planes are present on the side branch then propagation via TPRE growth can occur at a 70.5° angle to the original growth direction (the angle between separate $\{111\}$ planes). In the small angle branching mode, successive twinning leads to progressive displacement of two growing regions of a single plate, allowing the branches to diverge at an arbitrary angle based on the spacing of each new twin. Another possible mechanism for the development of adjacent silicon particles with slightly differing orientations is the fragmentation of the growing silicon phase. Such a fragmentation mechanism was suggested to occur for the Fe_3C phase in white cast iron.(11) The Fe_3C phase was found to fragment at the growing interface with such fragments forming new plates after presumably rotating randomly in the liquid, thus forming a so-called broomlike morphology. The similarity between this morphology and the Al-Si flake morphology prompted Fredriksson *et al.* and Hillert(27; 28) to suggest a similar event occurs in Al-Si. Of course, in order to form a new plate via the TPRE mechanism the initial fragment must contain two parallel twin planes. The fragmentation possibility was questioned by Dahle *et al.* (29) who noted the silicon ductile to brittle transition is around 550°C (30) and questioned the origin of the stresses capable of

fracturing the phase at the melt front.

Impurity-Modified Al-Si Growth

The most common commercial modification practice used today is impurity modification(26; 31–33). A number of elements have been found to induce some modifying effect when added to the Al-Si melt, though the most commonly used elements are sodium, strontium, and antimony. The first and most potent modifier element used was sodium, but its high vapor pressure at the Al-Si melt temperature makes it hard to retain in solution and therefore difficult to add in specific amounts. Sodium also tends to aggressively attack mold materials, and adding too much can result in a phenomenon called overmodification in which bands of primary aluminum are formed which degrade mechanical properties of the casting.(27) Strontium does not have as pronounced a modifying effect as sodium, but it is free from the vaporization and overmodification problems of sodium. Unfortunately, strontium modified alloys have a tendency toward excessive gas pickup and porosity formation in castings. Antimony additions yield a refined flake structure without as significant of benefits as the former additives, but antimony modification has a near permanent effect on the melt and may be more useful in large castings where microstructural homogeneity is desired.

The modified structures produced by sodium and strontium addition are finer than the corresponding flake structure and consist of microfaceted fibers with a large twin density, with twin spacings as low as 5 nm.(26) This twin density leads to a structure which appears smooth at low magnifications but is actually faceted on a fine scale. The microstructural changes associated with impurity modification of castings are attended by a depression in eutectic nucleation and growth temperature(33–37), reduction in eutectic colony nucleation frequency(37), and a flattening of the growth interface (particularly with sodium modification)(38–40).

The most likely mechanism of impurity modification is thought to be from growth restriction of the silicon phase by selective adsorption of impurity atoms onto fast growing silicon faces(41). Lu and Hellawell(26) noted that common modifying agents share similar atomic radii and proposed that adsorbed modifier elements promote twinning by displacing a $\{111\}$

Figure 11 Schematic of small angle (top) and large angle (bottom) branching mechanisms in eutectic silicon. Small angle branching involves a succession of displacement twins which create a branched arm and then cause it to diverge from the initial eutectic flake at a small, arbitrary angle. Large angle branching is possible when a silicon tetrahedron attaches to the $\{111\}$ surface of a silicon flake in orientation A. The attached tetrahedron will be in orientation B, and may then form a new orientation (C) by twinning. If a second twin of orientation B forms on C, it will produce a self-perpetuating groove which allows TPPE growth in another direction. Adapted from Shamsuzzoha and Hogan(25).

monolayer growth step to the next alternative stacking sequence. The increased incidence of twinning would then be responsible for the microfaceted fibrous shape of the eutectic, while the large undercooling observed in modified eutectic growth may be required to overcome the added twin boundary energy(42).

The altered nucleation frequency observed in modified eutectic growth may also play a role in its ultimate morphology. Crosley and Mondolfo(43) identified AlP particles as potent nuclei for eutectic silicon and suggested the modifying action of Na depends in part on the neutralization of these potent nuclei. The nucleation potential of AlP was verified by Dahle(33) who examined the AlP-silicon interface using high resolution transmission electron microscopy (TEM) imaging and found no lattice mismatch between the two. The effect of melt impurities was demonstrated by McDonald *et al.* (37) who found significant flake refinement and growth temperature depression in Al-Si alloys prepared from ultra high purity components rather than lower purity components. Additionally, the nucleating potential of iron-containing compounds was investigated by Shankar(44–46) who also found significant nucleation and growth temperature depression in a higher purity alloy versus a less pure alloy. Interestingly, they found eutectic flake refinement occurred in an alloy with ultra low iron content compared with an alloy of higher iron content but similar phosphorous content, indicating the flake refinement was not simply a result of phosphorous interactions. A recent study by Cho *et. al.*(47) detected decreased eutectic nucleation frequency in an Al-Si alloy of 1.1 wt% iron versus one of 0.5 wt% iron, suggesting that the iron content serves to nucleate on and passivate free AlP particles. They noted a similar effect in strontium-containing alloys, supporting the idea that the modifying effect of strontium may be related to AlP passivation, in addition to the growth restriction mentioned previously. The effect of iron on eutectic silicon grain nucleation was also reported previously by Dinnis *et. al.*(48).

The widespread commercial use of impurity modification has led to numerous efforts to characterize the transformed structure. Many of these attempts have been focused on defining and measuring a quantitative parameter that describes the degree of microstructural modification, or the silicon modification level. Early methods of determining silicon modification

level were based on subjectively characterizing the microstructure as compared to the standard American Foundry Society (AFS) Chart for Microstructure Control, which can be combined with thermal analysis observations to allow fast modification level estimation(36). Drawbacks to this method include the lack of objectivity when working with the microstructure chart and a decrease in resolution of thermal analysis at higher cooling rates. Recently, computer image analysis technology has been applied to increase the objectivity and accuracy of these measurements. Jiang *et al.* (49) used an image analysis system to objectively determine Al-Si modification level with increasing strontium additions. Particle length, width, and perimeter were found to correlate well with changing levels of structure modification. Hernandez and Sokolowski(50) expanded this technique to work on hypereutectic alloys and also found particle perimeter to effectively differentiate modification level.

Despite the significant advantages afforded by impurity modification, there are some significant drawbacks to its use. The volatility and reactive nature of sodium makes it difficult to obtain uniformly modified structures, while strontium contributes to porosity in castings. Other deleterious effects of impurity modification include hot tearing, poor surface finish, and recycling restrictions from the use of different modifying elements which do not mix well(31; 33).

Quench-Modified Al-Si Growth

An alternative to impurity modification is to elicit structure refinement via increased cooling rates, a process known as quench modification(21). Quench modification yields a fine fibrous structure that is microscopically smooth and infrequently twinned, thus differentiating it from the microfaceted impurity modified structure.(24; 51) The transition from the growth of silicon as bulky, faceted plates to smooth, globular fibers has been observed to occur over a wide range of velocities (Table 2), though specific descriptions of the flake to fiber transition mechanism in the literature are lacking. Khan and Elliott(52) found the transition to be accompanied by a significant drop in undercooling and suggested fibrous growth is then a departure from the normal, stepwise growth of broad faceted flakes toward continuous growth of a nonfaceted phase, thereby resembling aspects of a regular eutectic.

Interfacial Anisotropy

The importance of interfacial energy and anisotropy are evident upon examination of the irregular nature of the Al-Si eutectic structure. To aid in the understanding of anisotropic effects on Al-Si solidification, a more thorough understanding of the determination of surface free energy and anisotropy is useful.

There are several different techniques utilized in the measurement of interfacial free energy. The first method to be utilized, and potentially the simplest is the nucleation method(53; 54). The undercooling required for homogeneous nucleation of a solid within a liquid is strongly dependent on the solid-liquid interfacial energy barrier. The average system interfacial energy can then be determined indirectly from the undercooling required for homogenous nucleation. Of course, heterogeneous nucleation sites are almost impossible to eliminate and will inevitably influence the measured nucleation undercooling. Therefore, nucleation experiments effectively determine a lower limit for the interfacial free energy.

Another useful technique for the determination of interfacial free energy utilizes the shape of the grain boundary at the three-phase junction. There are two separate approaches with regard to this technique. One method is to measure the dihedral angle of the grain boundary triple point(55; 56). The grain boundary energy must then be independently determined to obtain interfacial energy. Another, more recent method involves the use of the interfacial grain boundary groove shape.(57) An advantage of this method is that the interfacial energy can be determined independent of the grain boundary energy.

While the nucleation undercooling and grain boundary groove shape methods yield useful results for the average interfacial free energy of a system, they are not effective in obtaining the relationship between interfacial energy and orientation, or the anisotropy. A more useful technique to obtain the interfacial energy anisotropy is the equilibrium shape method(57–59). In this method, droplets of one phase immersed in another are allowed to approach equilibrium and the resulting particle shapes are analyzed to determine the interfacial energy as a function of orientation.

The relation between the shape of a particle in equilibrium and its interfacial free energy was

described by Wulff in 1901(60): “*When a crystalline body is in its equilibrium shape, free from interfacial constraints and not acted upon by external body forces, there exists a point whose perpendicular distances from the surfaces of the body are in proportion to their free energies per unit area.*” This point coincides with the geometric center of the body for non-polar crystal structures. Effectively, the larger the surface energy of a particular crystallographic orientation, the farther away its surface will be from the center of the crystal. This leads to the relation:

$$\frac{\gamma_1}{\gamma_2} = \frac{r_1}{r_2}$$

where γ is the surface energy of a specific crystallographic orientation and r is the corresponding distance of that surface from the center of the body.

If the variation of the surface energy with orientation (the γ -plot) is known for a crystal structure then the equilibrium shape of the crystal can be reconstructed using the Wulff construction, which states: “*If planes be drawn normal to the radius vectors of the polar gamma-plot where they intersect its surface, then the envelope formed by points which can be reached from the origin without crossing any of these planes is geometrically similar to the equilibrium shape.*” The schematic shown in Fig. 12 demonstrates the determination of the equilibrium shape from a corresponding γ -plot, the reverse Wulff construction for the determination of a γ -plot from an equilibrium shape, and the issues encountered when an equilibrium shape contains sharp edges. While the equilibrium shape can be uniquely determined from the γ -plot, the reverse is not always true, as high energy crystallographic orientations are not necessarily present in the equilibrium shape of a particle. Missing orientations are generally only present when the anisotropy exceeds a certain value, so this permits measurement of interfacial free energy over all orientations for most low-anisotropy metallic systems. Examination of the Wulff construction further reveals that the equilibrium shape of a particle is always convex.

In order to simplify the relation between the equilibrium shape and its corresponding γ -plot, a two-dimensional slice of the crystal may be considered. The surface energy-anisotropy

relation on a cross-section of indices $\{hkl\}$ can then be expressed as:

$$\frac{\gamma_{hkl}(\theta)}{\gamma_0} = 1 + \sum_{m=2}^k \epsilon_m^{hkl} \cos m\theta$$

where m is the degree of rotational crystal symmetry for each mode about the section plane normal and θ is the angular deviation from the cross-section normal, $\langle hkl \rangle$. For a cubic metal, it is often assumed that $\gamma(\theta)$ is taken on a $\{100\}$ plane (common direction of growth for cubic metals) and is dominated by the $m = 4$ term (four fold symmetry around the $\{100\}$ axis). This leads to:

$$\gamma(\theta) = \gamma_0 (1 + \epsilon_4 \cos 4\theta) \quad (15)$$

An important parameter to consider is the intrinsic resistance of the interface to curvature. This resistance to curvature is known as the interfacial stiffness and is defined as:

$$\tilde{\gamma} \equiv \gamma(\theta) + \gamma''(\theta) \quad (16)$$

where $\gamma''(\theta)$ is the second derivative of the interfacial free energy with respect to θ and is calculated as follows:

$$\begin{aligned} \frac{\partial \gamma(\theta)}{\partial \theta} &= -4\gamma_0 \epsilon_4 \sin 4\theta \\ \frac{\partial^2 \gamma(\theta)}{\partial \theta^2} &= -16\gamma_0 \epsilon_4 \cos 4\theta \end{aligned}$$

Solving Eq. 16 for $\gamma(\theta)$, substituting into Eq. 15, and rearranging terms gives a relation for the stiffness as a function of anisotropy and crystallographic orientation:

$$\tilde{\gamma}(\theta) = \gamma_0 (1 - 15\epsilon_4 \cos 4\theta) \quad (17)$$

Comparison of Eq. 15 and Eq. 17 indicates that the stiffness is significantly more anisotropic than the interfacial energy, and thus is more easily detected in experiment. It is seen from Eq. 17 that when the anisotropy parameter, ϵ_4 , exceeds ≈ 0.067 then the equation predicts negative stiffness. This is a physical impossibility, and in reality manifests itself as missing orientations, as described earlier.

As it measures the resistance of a surface to curvature, the stiffness influences the selection of growth morphology in an evolving structure rather than strictly the free energy. The stiffness

Figure 12 Schematic Wulff constructions: (a) The Wulff construction for the determination of the equilibrium shape from the γ -plot (b) The reverse Wulff construction for determining the γ plot from an equilibrium shape (c) The reverse Wulff construction performed on a body of missing orientations yields only a minimum γ -plot. From Miller and Chadwick(58)

is closely related to the Gibbs-Thompson coefficient, which is generally stated as:

$$\Gamma = \frac{\tilde{\gamma}}{\Delta S_f}$$

However, the Gibbs-Thompson coefficient is usually stated as an average value over all crystal orientations, in which case the anisotropy term from Eq. 17 cancels out (the average value of $\cos \theta$ from 0 to 2π is equal to 0), and then the average stiffness is equal to the average, or isotropic value of the surface free energy. Consequently, the Gibbs-Thompson parameter is often stated as the average surface free energy over the entropy of fusion, $\Gamma = \gamma / \Delta S_f$. The use of the stiffness parameter in the Gibbs-Thompson coefficient is only important when considering the interfacial behavior of individual crystallographic directions in a material demonstrating anisotropic effects.

Interface Stability

Pure Metals

The interface shape during solidification of pure metals is governed by the competing effects of surface energy minimization and heat flow. Surface energy tends to flatten the interface in order to reduce overall interfacial area while the effect of heat diffusion depends on the direction of heat flow across the interface. During directional solidification, or in the columnar zone of a casting, the latent heat of solidification is conducted away through the solid as the liquid ahead of the interface is at a higher temperature than the solid. In this case, if a perturbation forms at the solid interface protruding into the liquid it will increase the temperature gradient in the liquid and decrease the temperature gradient in the solid. This will cause more heat to flow into the tip from the liquid than out of it into the solid, and the temperature of the tip will rise and cause it to melt back toward the planar interface. In the reverse case of a perturbation causing a depression in the solid phase, the temperature gradient will be decreased in the liquid and increased in the solid, causing less heat to flow into the depression than out of it, lowering the depression temperature and causing it to solidify and advance toward the planar interface. In both cases the planar interface is stabilized and no dendritic structures will develop. The opposite situation occurs in equiaxed solidification where a free crystal grows into an undercooled melt. In this case, latent heat of solidification causes the temperature of the solid crystal to rise and heat is then conducted away through the liquid. In this case, the temperature gradient in the solid is essentially zero, and any solid perturbation extending into the liquid phase will cause the tip to experience a higher negative temperature gradient, eject more heat, and cause it to grow faster into the melt. The initially spherical precipitate will then form a dendritic structure. Of course, there are no segregation effects in a pure metal system and the dendritic growth form is subsequently not detectable. The temperature gradient criterion for instability of a planar interface in pure metals is thus $G < 0$.

Metal Alloys

Constitutional Supercooling Criterion The stability of a planar interface for alloy solidification requires the additional consideration of solute effects. After some solidification time, the diffusion profile ahead of the interface will become constant with respect to the moving interface which signifies that steady state has been achieved. In the case of a general alloy system where solute is rejected into the melt (such as in the Al-Si system, Fig. 1) there will be a compositional maximum in the liquid at the interface. Composition ahead of the interface will then decay exponentially with distance to the initial alloy composition. This region of disturbed solute profile is the diffusional boundary layer. The stability of the interface is influenced by the shape of this boundary layer due to compositional effects on the equilibrium melting temperature. For the present alloy system, the maximum liquid composition corresponds to the minimum equilibrium melting temperature. Ahead of the interface the equilibrium melting temperature increases with the decrease in composition, while the actual liquid temperature increases as a linear function of the imposed temperature gradient. If the compositional gradient ahead of the interface is great enough (corresponding to a thin boundary layer or high solidification velocity) then it is possible for the equilibrium melting temperature to increase faster than the actual liquid temperature. There will then be a volume of liquid ahead of the interface which is at a lower actual temperature than its equilibrium melting point—it is supercooled (Fig. 13). This is known as a region of constitutional supercooling(61). In this volume, there is a driving force for the development and amplification of perturbations contributing to the breakdown of the planar interface.

The condition of constitutional supercooling requires that the equilibrium temperature gradient is greater than the actual temperature gradient of the liquid at the interface. The equilibrium temperature gradient is obtained by multiplying the composition gradient and the change of liquidus temperature with composition (slope of the liquidus line). The criterion for the occurrence of constitutional supercooling in alloys is thus $G < mG_c$.

Figure 13 Schematic of constitutional supercooling. Solidification in a binary alloy system is proceeding from left to right along the x axis with solute being rejected into the melt. As the composition of the liquid decreases with distance ahead of the interface, the equilibrium melting temperature increases. If the effect of the compositional profile on the equilibrium melting temperature is greater than the actual temperature gradient experienced during solidification, there will exist a region of constitutional supercooling (shaded region) where the liquid is at a lower temperature than its melting point. This region of liquid is thermodynamically unstable.

Perturbed Interface Model The constitutional supercooling criterion approach for the determination of interface stability includes several drawbacks: it does not take into account capillarity effects, only the competition between thermal and solutal effects are considered; it does not give any indicator of the scale of perturbations which are expected to destabilize a planar interface; and it does not take account of the temperature gradient in the solid. Mullins and Sekerka(16) approached the problem of interface stability by calculating the stability of an infinitesimal sinusoidal perturbation introduced into a planar interface. The sinusoidal perturbation can be considered to represent a component of a Fourier series which generalizes the treatment to the appearance of any possible disturbance. In this case, the interface is stable only if there is no perturbation of any wavelength that will increase in amplitude with time. The derivative of the perturbation amplitude with respect to time is the parameter that must be considered.

Consider a coordinate system attached to a planar interface moving with velocity V . A perturbation is introduced into the interface with the mathematical form of:

$$z = \delta(t) \sin \omega y$$

where δ is the wave amplitude and $\omega = 2\pi/\lambda$. In order to determine whether the perturbation will grow and destabilize the planar interface, the velocity of the interface at all points, $V(y)$, must be determined. This is accomplished by considering the local thermal and diffusion fields at the interface given appropriate boundary conditions. The present solution assumes local equilibrium at the interface, treats the surface as isotropic, and requires steady state conditions. This means that once a perturbation becomes stable and begins to grow in amplitude the assumption of steady state is no longer valid and the analysis breaks down. Therefore, only the scale of the initial planar interface breakdown can be obtained directly from the instability analysis while the resultant macrostructural scale can only be inferred.

The thermal and diffusion fields ahead of the interface under steady state are given by

forms of the directional growth equation, Eq. 8:

$$\frac{\partial^2 C}{\partial y^2} + \frac{\partial^2 C}{\partial z^2} + \frac{V}{D} \frac{\partial C}{\partial z} = 0 \quad (18)$$

$$\frac{\partial^2 T_S}{\partial y^2} + \frac{\partial^2 T_S}{\partial z^2} + \frac{V}{D_S} \frac{\partial T_S}{\partial z} = 0 \quad (19)$$

$$\frac{\partial^2 T_L}{\partial y^2} + \frac{\partial^2 T_L}{\partial z^2} + \frac{V}{D_L} \frac{\partial T_L}{\partial z} = 0 \quad (20)$$

where T_L and T_S represent the temperatures in the liquid and solid, respectively, while D_L and D_S are the respective liquid and solid thermal diffusivities.

The temperature at the interface (melting temperature) is influenced by the local composition and the effect of curvature. The corresponding boundary condition is:

$$T_i = T_m - mC_i - \Gamma\omega^2\delta \sin \omega y \quad (21)$$

where Γ is the Gibbs-Thompson coefficient and values at the interface are denoted with i .

The velocity calculated at the interface is also required to be consistent between the heat flow and solute flow constraints, which leads to:

$$V(y) = \frac{\kappa_S G_S^i - \kappa_L G_L^i}{\Delta H_f} = \frac{DG_c^i}{C_i(k-1)} \quad (22)$$

where G_S and G_L are the temperature gradients with respect to the z axis in the solid and liquid respectively, G_c is the composition gradient in the z direction, and κ denotes the conductivity of the solid and liquid.

Appropriate solutions to Eqs. 18–20 are:

$$C = C_\infty + \frac{G_c D}{V} \left[1 - \exp\left(-\frac{Vz}{D}\right) \right] + \delta(b - G_c) \sin \omega y \exp(-b_c z) \quad (23)$$

$$T_L = T_i + \frac{G_L D_L}{V} \left[1 - \exp\left(-\frac{Vz}{D_L}\right) \right] + \delta(a - G_L) \sin \omega y \exp(-b_L z) \quad (24)$$

$$T_S = T_i + \frac{G_S D_S}{V} \left[1 - \exp\left(-\frac{Vz}{D_S}\right) \right] + \delta(a - G_S) \sin \omega y \exp(-b_S z) \quad (25)$$

where:

$$\begin{aligned} b_c &= \frac{V}{2D} + \left[\left(\frac{V}{2D} \right)^2 + \omega^2 \right]^{\frac{1}{2}} \\ b_L &= \frac{V}{2D_L} + \left[\left(\frac{V}{2D_L} \right)^2 + \omega^2 \right]^{\frac{1}{2}} \\ b_S &= \frac{V}{2D_S} + \left[\left(\frac{V}{2D_S} \right)^2 + \omega^2 \right]^{\frac{1}{2}} \end{aligned}$$

and a and b are constants. These constants can be determined by substituting Eqs. 23–25 into Eqs. 21 and 22 and rearranging.

Finally, the local velocity of the interface is assumed to be:

$$V(y) = V + \dot{\delta} \sin \omega y$$

where $\dot{\delta} = \frac{\delta(t)}{dt}$ and $V(y)$ is given by Eq. 22. Performing this substitution and equating like coefficients leads to the central stability result:

$$\frac{\dot{\delta}}{\delta} = \frac{V\omega \left(-2\Gamma\omega^2 [b_c - (Vp/D)] - (G'_L + G'_S) [b_c - (Vp/D)] + 2mG_c [b_c - (V/D)] \right)}{(G'_S - G'_L) [b_c - (Vp/D)] + 2\omega mG_c}$$

where:

$$G'_L = (\kappa_L/\bar{\kappa}) G$$

$$G'_S = (\kappa_S/\bar{\kappa}) G$$

$$\bar{\kappa} = \frac{\kappa_S + \kappa_L}{2}$$

$$p = 1 - k$$

This equation can be simplified by assuming the temperature gradients and thermal conductivities of the solid and liquid phases are equal. This leads to:

$$\frac{\dot{\delta}}{\delta} = \frac{V}{mG_c} \left(-\Gamma\omega^2 [b_c - (Vp/D)] - G [b_c - (Vp/D)] + mG_c [b_c - (V/D)] \right)$$

A plot of this equation is shown in Fig. 14. The most important aspect of this equation is when the amplification of the perturbation amplitude is equal to 0, the case of marginal stability.

This is true for:

$$\Gamma\omega^2 [b_c - (Vp/D)] - G [b_c - (Vp/D)] + mG_c [b_c - (V/D)] = 0 \quad (26)$$

In the case that this quantity is less than 0 the perturbation amplitude shrinks and the perturbation will eventually die out. Conversely, in the case the amplification rate is positive, the perturbation will grow in magnitude and cause the breakdown of the planar interface. The equation indicates that the stability of the planar interface is determined by three factors, namely the effect of capillarity, the thermal field, and the solute field. The first two terms have a negative sign, indicating the effects of surface energy and a positive thermal gradient are to stabilize the planar interface. The solute term, on the other hand, has a positive contribution to perturbation stability, thus tending to destabilize the planar interface.

Further examination of the individual effects on planar stability is beneficial. From the relation between the first two terms of Eq. 26, it is apparent the temperature gradient must be similar in scale to $\Gamma\omega^2$ in order to have a significant effect on the overall stability. However, given the small scale eutectic structures typically operate under, the temperature gradient often turns out to be insignificant in the development of planar instabilities. Consider the case of an Al-Si irregular eutectic over common solidification velocities (10–1000 $\mu\text{m/s}$) and reasonable temperature gradients (1–10 K/mm). The maximum corresponding microstructural length scale is about 10 μm while the Gibbs-Thompson coefficient is about 2×10^{-7} m-K. This gives $\Gamma\omega^2 = 80000$ K/m, compared to a max gradient of $G = 10000$ K/m. This implies a maximum gradient contribution around 12% under the most favorable circumstances. Furthermore, it will be shown that planar silicon instabilities in this alloy operate at higher velocities and at smaller length scales than the eutectic spacing. Under these scenarios, the effect of the temperature gradient compared to capillarity is much less than one percent and may be safely ignored. Removing the temperature gradient from Eq. 26 and rearranging yields:

$$\frac{mG_c}{\Gamma} \frac{b_c - V/D}{b_c - Vp/D} = \left(\frac{2\pi}{\lambda} \right)^2$$

Another observation from the Al-Si system is that $p \approx 1$. Applying this simplification yields a

Figure 14 Amplification rate behavior of the stability of a perturbation of specified wavelength on a planar interface.

simple relationship for the case of marginal stability:

$$\lambda = 2\pi \sqrt{\frac{\Gamma}{mG_c}}$$

This is the same relation as given by Fisher and Kurz(15). It represents the point at which instability wavelengths of increasing magnitude first become stable. Finally, noting that if the composition gradient is proportional to velocity, then the equation above can be rearranged to obtain $\lambda^2 V = \text{constant}$.

References

- [1] Nabil Fat-Halla. Structure, mechanical properties and fracture of aluminum alloy a-356 modified with aluminum-strontium (al-5sr) master alloy. *Journal of materials science*, 22(3):1013, 1987.
- [2] N. Fat-Halla. Structural modification of aluminum-silicon eutectic alloy by strontium and its effect on tensile and fracture characteristics. *Journal of materials science*, 24(7):2488, 1989.

- [3] Mahmoud Fouad Hafiz, Nabil K. Fat-Halla, and Sherif B. Moshref. The effect of casting parameters on the modification process of a commercial hypoeutectic aluminum-silicon alloy. *Zeitschrift fr Metallkunde*, 81(1):70, 1990.
- [4] J. L. Murray and A. J. McAllister. The al-si (aluminum-silicon) system. *Bulletin of Alloy Phase Diagrams*, 5(1):74, 1984.
- [5] W. Kurz and D. J. Fisher. *Fundamentals of Solidification*. Trans Tech Publications Ltd, Enfield, NH, fourth revised edition edition, 1998.
- [6] L. G. England. High-rate growth-mode transitions in al-si eutectics. Master’s thesis, Iowa State University, 2004.
- [7] K. A. Jackson and J. D. Hunt. Lamellar and rod eutectic growth. *Transactions of the Metallurgical Society of AIME*, 236(8):1129, 1966.
- [8] C. Zener. Kinetics of the decomposition of austenite. *AIME Transactions*, 167:550–595, 1946.
- [9] W. H. Brandt. Solution of the diffusion equation applicable to the edgewise growth of pearlite. *Journal Of Applied Physics*, 16(3):139–146, 1945.
- [10] W. A. Tiller. *Liquid Metals and Solidification*. ASM, Cleveland, 1958.
- [11] M. Hillert and H. Steinhuser. The structure of white cast iron. *Jernkontorets Annaler*, 144:520–559, 1960.
- [12] H. A. H. Steen and A. Hellawell. The growth of eutectic silicon - contribution to under-cooling. *Acta Metallurgica*, 23:529–535, 1975.
- [13] K. A. Jackson. Mechanism of growth. In *Liquid Metals and Solidification*, pages 174–186. ASM, 1958.
- [14] T. Sato and Y. Sayama. Completely and partially co-operative growth of eutectics. *Journal of Crystal Growth*, 22:259–271, 1974.

- [15] D. J. Fisher and W. Kurz. A theory of branching limited growth of irregular eutectics. *Acta Metallurgica*, 28(6):777, 1980.
- [16] W. W. Mullins and R. F. Sekerka. Stability of a planar interface during solidification of a dilute binary alloy. *Journal of Applied Physics*, 35:445–451, 1964.
- [17] P. Magnin and W. Kurz. An analytical model of irregular eutectic growth and its application to fe - c. *Acta Metallurgica*, 35(5):1119–1128, 1987.
- [18] P. Magnin, J. T. Mason, and R. Trivedi. Growth of irregular eutectics and the aluminum-silicon system. *Acta metallurgica et materialia*, 39(4):469, 1991.
- [19] P. Magnin and R. Trivedi. Eutectic growth: A modification of the jackson and hunt theory. *Acta Metall Mater*, 39(4):453–467, 1991.
- [20] Mahmoud Hafiz and Toshiro Kobayashi. Tensile properties influencing variables in eutectic al-si casting alloys. *Scripta metallurgica et materialia*, 31(6):701, 1994.
- [21] M. G. Day and A. Hellawell. Microstructure and crystallography of aluminum-silicon eutectic alloys. *Proceedings - Royal Society. Mathematical, physical and engineering sciences*, 305(1483):473, 1968.
- [22] R. S. Wagner. On the growth of germanium dendrites. *Acta Metallurgica*, 8(1):57–60, 1960.
- [23] D. R. Hamilton and R. G. Seidensticker. Propagation mechanism of germanium dendrites. *Journal of Applied Physics*, 31:1165–1168, 1960.
- [24] Shu Zu Lu and Angus Hellawell. Growth mechanisms of silicon in aluminum-silicon alloys. *Journal of Crystal Growth*, 73(2):316, 1985.
- [25] M. Shamsuzzoha and L. M. Hogan. Crystal morphology of unmodified aluminum-silicon eutectic microstructures. *Journal of Crystal Growth*, 76(2):429, 1986.

- [26] Shu Zu Lu and A. Hellawell. The mechanism of silicon modification in aluminum-silicon alloys: impurity induced twinning. *Metallurgical transactions. A, Physical metallurgy and materials science*, 18A(10):1721, 1987.
- [27] H. Fredriksson, M. Hillert, and N. Lange. Modification of aluminum-silicon alloys by sodium. *Journal of the Institute of Metals*, 101(Nov):285, 1973.
- [28] M. Hillert. Discussion of "eutectic solidification of al-si alloys". *Metallurgical and Materials Transactions; A; Physical Metallurgy and Materials Science*, 34A(11):2688, 2003.
- [29] A. K. Dahle, S. D. McDonald, and K. Nogita. Authors' reply to discussion of "eutectic solidification of al-si alloys". *Metallurgical and Materials Transactions; A; Physical Metallurgy and Materials Science*, 34A(11):2690–2693, 2003.
- [30] Y. B. Xin and K. J. Hsia. A technique to generate straight through thickness surface cracks and its application to studying dislocation nucleation in si. *Acta Materialia*, 44(3):845–853, 1996.
- [31] M. Garat, G. Laslaz, S. Jacob, P. Meyer, P. H. Guerin, and R. Adam. State-of-art use of sb-, na- and sr-modified al-si casting alloys. *Transactions of the Amercian Foundrymen's Society*, 100:821–832, 1992.
- [32] D. C. Jenkinson and L. M. Hogan. Modification of aluminum-silicon alloys with strontium. *Journal of Crystal Growth*, 28(2):187, 1975.
- [33] A. K. Dahle, K. Nogita, S. D. McDonald, C. Dinnis, and L. Lu. Eutectic modification and microstructure development in alsil alloys. *Materials Science and Engineering A*, 413-414:243–248, 2005.
- [34] R. C. Plumb and J. E. Lewis. The modification of aluminum-silicon alloys by sodium. *Journal of the Institute of Metals*, 86:393–400, 1958.
- [35] C. B. Kim and R. W. Heine. Fundamentals of modification in the aluminum-silicon system. *Journal of the Institute of Metals*, 92:367, 1963.

- [36] N. Tenekedjiev and J. E. Gruzleski. Thermal analysis of strontium treated hypoeutectic and eutectic aluminum-silicon casting alloys. *Transactions of the American Foundry Society*, 99:1–6, 1991.
- [37] Stuart D. McDonald, Kazuhiro Nogita, and Arne K. Dahle. Eutectic nucleation in al-si alloys. *Acta Materialia*, 52(14):4273–4280, 2004.
- [38] S. C. Flood and J. D. Hunt. Modification of aluminum-silicon eutectic alloys with sodium. *Metal science*, 15(7):287, 1981.
- [39] M. D. Hanna, Shu Zu Lu, and A. Hellawell. Modification of the aluminum-silicon system. *Metallurgical transactions. A, Physical metallurgy and materials science*, 15A(3):459, 1984.
- [40] Q. F. Hamed, R. Elliott, and P. S. Cooper. Solidification characteristics of sodium and strontium modified aluminum-silicon casting alloys. *Light Metals*, pages 1391–1397, 1992.
- [41] V. L. Davies and J. M. West. Factors affecting the modification of the aluminum-silicon eutectic. *Journal of the Institute of Metals*, 92:175–180, 1963.
- [42] L. M. Hogan and H. Song. Interparticle spacings and undercoolings in aluminum-silicon eutectic microstructures. *Metallurgical transactions. A, Physical metallurgy and materials science*, 18A(4):707, 1987.
- [43] P. B. Crosley and L. F. Mondolfo. The modification of aluminum-silicon alloys. *Modern casting*, 49(3):89, 1966.
- [44] Sumanth Shankar, Yancy W. Riddle, and Makhlof M. Makhlof. Nucleation mechanism of the eutectic phases in aluminum-silicon hypoeutectic alloys. *Acta Materialia*, 52(15):4447–4460, 2004.
- [45] S. Shankar, Y. W. Riddle, and M. M. Makhlof. The role of iron in the nucleation of eutectic silicon in aluminum-silicon hypoeutectic alloys. *Solidification of Aluminum Alloys*, pages 103–109, 2004.

- [46] Sumanth Shankar, Y. W. Riddle, and M. M. Makhlof. Eutectic solidification of aluminum-silicon alloys. *Metallurgical and Materials Transactions; A; Physical Metallurgy and Materials Science*, 35A(9):3038, 2004.
- [47] Y. H. Cho, H. C. Lee, K. H. Oh, and A. K. Dahle. Effect of strontium and phosphorus on eutectic al-si nucleation and formation of α -al₅fe_{si} in hypoeutectic al-si foundry alloys. *Metallurgical and Materials Transactions; A; Physical Metallurgy and Materials Science*, 39(10):2435, 2008.
- [48] C. Dinnis, J. A. Taylor, and A. K. Dahle. Interactions between iron, manganese, and the al-si eutectic in hypoeutectic al-si alloys. *Metallurgical and Materials Transactions; A; Physical Metallurgy and Materials Science*, 37A(11):3283, 2006.
- [49] H. Jiang, J. H. Sokolowski, M. B. Djurdjevic, and W. J. Evans. Recent advances in automated evaluation and on-line prediction of al-si eutectic modification level. *Transactions of the Amercian Foundry Society*, 108:505–510, 2000.
- [50] Francisco C. Robles Hernandez and Jerry H. Sokolowski. Novel image analysis to determine the si modification for hypoeutectic and hypereutectic al-si alloys. *Journal of Metals*, 57, 2005.
- [51] L. M. Hogan and M. Shamsuzzoha. Crystallography of the flake-fiber transition in the aluminum-silicon eutectic. *Materials forum*, 10(4):270, 1987.
- [52] S. Khan and R. Elliott. Quench modification of aluminum-silicon eutectic alloys. *Journal of Materials Science*, 31(14):3731–3737, 1996.
- [53] D. Turnbull. Formation of crystal nuclei in liquid metals. *Journal of Applied Physics*, 21:1022–1027, 1950.
- [54] D. Turnbull and R. E. Cech. Microscopic observation of the solidification of small metal droplets. *Journal of Applied Physics*, 21:804–810, 1950.

- [55] M. E. Glicksman and C. L. Vold. Determination of absolute solid-liquid interfacial free energies in metals. *Acta Metallurgica*, 17:1–11, 1969.
- [56] J. W. Taylor. An evaluation of interface energies in metallic systems. *Journal of the Institute of Metals*, 86:456–463, 1958.
- [57] R. E. Napolitano, S. Liu, and R. Trivedi. Experimental measurement of anisotropy in crystal-melt interfacial energy. *Interface Science*, 10(2-3):217–232, 2002.
- [58] W. A. Miller and G. A. Chadwick. Equilibrium shapes of small liquid droplets in solid-liquid phase mixtures - metallic hcp and metalloid systems. *Proceedings Of The Royal Society Of London Series A-Mathematical And Physical Sciences*, 312(1509):257–266, 1969.
- [59] R. E. Napolitano and S. Liu. Three-dimensional crystal-melt wulff-shape and interfacial stiffness in the al-sn binary system. *Physical Review B*, 70(21):11, 2004.
- [60] G. Wulff. On the question of the rate of growth and dissolution of crystal surfaces. *Z. Krystall.*, 34:449–530, 1901.
- [61] W. A. Tiller, K. A. Jackson, J. W. Rutter, and B. Chalmers. The redistribution of solute atoms during the solidification of metals. *Acta Metallurgica*, 1:428–437, 1953.

ANALYSIS OF THE HIGH GROWTH-RATE TRANSITION IN AL-SI EUTECTIC SOLIDIFICATION

A paper published in the Journal of Materials Science

T. Hosch, L. G. England, and R. E. Napolitano

Abstract

The local solidification conditions and mechanisms associated with the flake to fiber growth mode transition in Al-Si eutectic alloys are investigated here using Bridgman-type gradient-zone directional solidification. Resulting microstructures are examined through quantitative image analysis of two-dimensional sections and observation of deep-etched sections showing three-dimensional microstructural features. Several microstructural parameters were investigated in an attempt to quantify this transition, and it was found that the particle aspect ratio is effective in objectively identifying the onset and completion velocity of the flake to fiber transition, while traditional spacing parameters are not effective indicators of the transition. For a thermal gradient of 7–14 K/mm, the transition was found to occur in two stages, appearing over velocity regimes from 0.10 to 0.50 mm/s and from 0.50 to 0.95 mm/s. The initial stage is dominated by in-plane plate breakup and rod formation within the plane of the plate, while the second stage is characterized by the onset of out-of-plane silicon rod growth, leading to the formation of an irregular fibrous structure. The boundary between the two stages is marked by widespread fibrous growth and the disappearance of the remnant flake structure, indicating a transition in the structural feature that governs the relevant diffusion length, from inter-flake spacing to inter-rod spacing.

Introduction

The alloys based on the Al-Si eutectic system are among the most versatile for use in net shape castings, offering an extensive range of properties suitable for a broad scope of engineering applications calling for high strength to weight ratio, good corrosion resistance, and excellent castability. While the intrinsic flake silicon morphology observed in conventional Al-Si castings imparts relatively poor mechanical properties, ductility and strength are significantly improved through modification practices, where the silicon flake structure is refined to a globular or fibrous morphology. Typical effects of modification on Al-Si cast alloys include a 50% improvement in tensile strength and a threefold improvement in ductility.(1–5) These enhanced properties are most commonly achieved through impurity modification,(6–9) a practice where trace amounts of certain alloying additions (Na or Sr) induce a high density of $\{111\}$ twins,(7) giving rise to a microfaceted growth structure which allows local response to the evolving diffusion field and the formation of a fibrous morphology. However, in addition to increased costs and recycling limitations, impurity modification is also associated with deleterious effects such as porosity, hot tearing, and poor surface finish.(6; 9) An alternative is to elicit a similar structural refinement and morphological transition through increased cooling rates, a practice known as quench modification.(10; 11) Quench modification yields a fine fibrous structure that is microscopically smooth, with a low density of incidental twins, clearly differentiating it from the microfaceted impurity modified structure. However, the local interface conditions and mechanisms that lead to quench modification during solidification are not yet fully understood, thus limiting reliable prediction and effective employment of quench modification in certain casting applications. More effective utilization of this growth mode transition for property enhancement requires better description of the transition itself in terms of quantitative microstructural parameters, enabling consistent identification of the transition and reliable determination of the prevailing local growth conditions.

To date, there have been many attempts to characterize the Al-Si eutectic microstructure. In particular, microstructural changes during impurity modification have been characterized with some success. Early methods of determining silicon modification level were based on

manual microscopic assessment of the microstructure, which can be combined with thermal analysis observations to allow quick estimation of modification level.(12) Drawbacks to this method include a lack of objectivity when assessing modification level and a decrease in resolution of thermal analysis methods at higher cooling rates. Recently, computer image analysis technology has been applied to increase the objectivity and accuracy of these measurements. Jiang *et al.*(13) used two-dimensional image analysis to objectively determine the Al-Si modification level with increasing Sr additions. Particle length, width, and perimeter were found to correlate well with changing levels of structure modification. Hernandez and Sokolowski(14) expanded this technique to work on hypereutectic alloys and also found particle perimeter to effectively differentiate modification level.

The microstructural changes associated with the quench modification transition have not been characterized as thoroughly as those associated with impurity modification. Average interphase spacing has been measured by several investigators over a range of solidification velocities, as shown in Table 1, but such spacing parameters contain only indirect morphological information and do not adequately reflect the changing growth mode behavior throughout the flake to fiber transition. Due to the lack of appropriate quantitative parameters that indicate the transition, previous estimates of the velocity for the quench modification transition (Table 2) are based on qualitative analysis of structure, and results vary significantly. With no established method for objective identification of the flake-fiber transition, processing-map boundaries indicating the conditions which give rise to flake or fibrous morphologies are not well defined, and growth-mode prediction based on local solidification parameters is not reliable. A preliminary analysis(15) verified that the flake to fiber transition occurs over a wide velocity range, and suggested that new microstructural parameters may be able to indicate the transition. In the work presented here, we (i) quantitatively characterize the Al-Si microstructure over different velocities of solidification using a few selected microstructural parameters, and (ii) qualitatively analyze the process of quench modification to examine more closely the microstructural mechanisms associated with the growth mode transition indicated by the quantitative measurements.

Table 1 Reported Al-Si eutectic growth parameters from length scale (λ) measurements.

	Year	Gradient K/mm	Velocity $\mu\text{m/s}$	Measurement	A ^a	m ^a
Day and Hellawell(10)	1968	3–10	4–14	λ^b	22	-0.56
Fredriksson, Hillert, and Lange(16)	1973	30	4–400	λ^b	46	-0.61
Toloui and Hellawell(17)	1976	0.7	20–400	$\lambda_{ }, \lambda_A$	46	-0.41
		1.5	20–400	$\lambda_{ }, \lambda_A$	35	-0.42
		4.5	20–400	$\lambda_{ }, \lambda_A$	28	-0.43
		8	20–400	$\lambda_{ }, \lambda_A$	25	-0.45
		15	20–400	$\lambda_{ }, \lambda_A$	23	-0.48
Elliott, Glenister(18)	1980	0.8	5–170	λ^b	26	-0.33
Atasoy(19)	1984	12	10–474	$\lambda_{ }$	25	-0.34
Hogan and Song(20)	1987	11	10–200	$\lambda_{ }$	23	-0.45
Liu, Zhou, and Shang(21)	1990	9–11	5–200	$\lambda_{ }$	25	-0.46
Magnin, Mason, and Trivedi (22)	1991	8	0.1–500	$\lambda_{ }^c$	34	-0.5
Khan and Elliott(23)	1996	12.2	28–875	$\lambda_{ }, \lambda_A$	32	-0.47
		7.6	28–875	$\lambda_{ }, \lambda_A$	30	-0.43
Wolczynski, Billia, and Rabczak(24)	1996	8	3–70	$\lambda_{ }$	19	-0.50
		0.8	10–165	$\lambda_{ }$	24	-0.30
Cuprys, Major, and Wolczynski(25)	2000	4	300–790	$\lambda_{ }, \lambda_A$	22	-0.47
		10	300–790	$\lambda_{ }, \lambda_A$	15	-0.4
Napolitano and England(15)	2004	7.5	10–1200	λ_A	37	-0.52
		7.5	10–1200	λ_L	56	-0.49
Guzik and Kopycinski(26)	2006	10.5	0.3–11	$\lambda_{ }^d$	17	-0.35
Current Study		7–14	10–2000	$\lambda_{ }$	27	-0.53
		7–14	10–2000	λ_A	48	-0.55
		7–14	10–2000	λ_L	55	-0.52

^a All parameters given as $\lambda = AV^m$.

^b Unspecified linear intercept method.

^c Selected eutectic regions for intercept analysis not restricted to areas where lamellae were strictly parallel.

^d Lines for spacing determination were drawn parallel to the growth interface.

Table 2 Reported transition velocities.

	Gradient, K/mm	Velocity, $\mu\text{m/s}$
Steen and Hellawell(27)	11	400–800
	23	160–200
Jenkinson and Hogan(8)	9–40	480–1000
Toloui and Hellawell(17)	8	800–1000
Atasoy(19)	12	395–570
Khan and Elliott(23)	8–12	505–807
Napolitano and England(15)	7.5	100–1000

Experimental methods and findings

Alloy test specimens of an Al-13wt% Si alloy were fabricated by vacuum-arc-melting the pure constituents (<10 ppm impurities) on a water-cooled copper hearth, casting into 8 mm billets, swaging to 5 mm diameter rods, and cutting to lengths of 300 mm.

Directional solidification experiments were performed in a Bridgman-type furnace over a range of velocities from 10 $\mu\text{m/s}$ to 2000 $\mu\text{m/s}$. The temperature gradient during solidification was measured using a thermocouple embedded in a separate sample and ranged from 7 K/mm at 10 $\mu\text{m/s}$ to 14 K/mm at 1000 $\mu\text{m/s}$. Test specimens were contained in an alumina tube of 5.5 mm inner diameter and 8.0 mm outer diameter. Samples were grown over a distance of 50 mm and then quenched to 25°C in a water-cooled bath containing a Ga-In-Sn alloy. Cross sections were taken from each sample 10 mm behind the final quenched interface to obtain a representative eutectic structure and prepared for metallographic analysis. Characteristic microstructures observed over the range of velocities are shown in Fig. 1. Early stages of the flake to fiber transition can be seen at velocities as low as 20 to 50 $\mu\text{m/s}$, though the images suggest that the transition occurs primarily over the velocity range of 100 to 950 $\mu\text{m/s}$.

Specimens to be used for the determination of interflake spacing were polished and etched with a 2.5%HCl-1.5%HNO₃-1%HF reagent for 10–30 seconds for SEM analysis. Selected specimens were deep etched by suspending in an ultrasonic bath with the etchant for one hour. Three conventional spacing metrics were determined from eutectic cross sections. The spacing between parallel flakes, $\lambda_{||}$, was measured using intercept lines perpendicular to selected silicon

flakes observed to be approximately parallel. The random line spacing, λ_L , was determined using intercept lines oriented randomly. An area-based spacing parameter, λ_A , was calculated as:

$$\lambda_A = \sqrt{A/N}$$

where N is the number of particles that are observed on a two-dimensional analysis area, A . A schematic demonstrating the parallel and random line spacing measurements is shown in Fig. 2. The conventional spacing parameter results are shown in Fig. 3 as a function of velocity. All spacing parameters closely followed a $\lambda^2 V = \text{constant}$ relationship.

The average particle aspect ratio was also used to characterize the silicon structure. This is defined as the ratio of the largest caliper dimension of a particle to the smallest dimension perpendicular to the maximum. The aspect ratio as a function of velocity is displayed in Fig. 4. The data show three distinct regimes marked by abrupt transitions in slope at the velocities $135 \mu\text{m/s}$ and $880 \mu\text{m/s}$. These velocities bound the transition range, within which both flake and fiber morphologies are readily observed.

Analysis and discussion

Qualitative inspection of the microstructures shown in Fig. 1 reveals that the flake to fiber transition occurs gradually (with velocity) and in somewhat overlapping but distinct stages. In the lowest studied velocities of 10 and $20 \mu\text{m/s}$, the microstructure consists of large silicon plates. The plate structure is generally unbroken, though there are isolated instances where individual plates or groups of plates have begun to break up. This is not entirely unexpected given the inherently large variation in local spacings and solidification conditions of irregular eutectic alloys. In the 50 and $100 \mu\text{m/s}$ structures, there appears to be only slight if any deviation from the broad plate structure. Indeed, if one ignores scale, it is difficult to differentiate between the microstructures in this range, indicating the flake to fiber transition has not yet begun. However, a significant change is seen from 100 to $250 \mu\text{m/s}$. At $250 \mu\text{m/s}$, the plate structure has degraded significantly, leaving a dispersion of skeletal plates and various rodlike structures within the plane of the plate, indicating the beginning of the

Figure 1 Typical silicon morphology after directional solidification of an Al-Si eutectic alloy in a temperature gradient of 7–14 K/mm over a velocity range from 10 to 950 $\mu\text{m/s}$. The Al-rich phase has been chemically removed.

Figure 2 Schematic demonstrating the calculation of the parallel and random line spacing measurements. The $\lambda_{||}$ measurement is determined by drawing lines perpendicular to parallel silicon plates while the λ_L measurement involves the generation of random lines over the eutectic area.

Figure 3 Graph of three different spacing parameters. The λ_L measurement reflects the average spacing considering all possible line orientations. The $\lambda_{||}$ measurement indicates the average spacing perpendicular to parallel silicon particles. The λ_A measurement is effectively an inverse particle density function, indicating that particles become more numerous within a constant volume at high velocities. Error bars represent the 95% confidence interval of the mean.

Figure 4 Graph of the average particle aspect ratio for the range of velocities studied. The aspect ratio displays distinct transition points near $135 \mu\text{m/s}$ and $880 \mu\text{m/s}$, which are near the visually observed onset and conclusion velocities of the flake to fiber transition. Error bars represent the 95% confidence interval of the mean.

flake to fiber transition. Another important event is noted at $500 \mu\text{m/s}$. For the first time, out-of-plane rod growth is seen. Several examples of rods growing out of flat plate faces are evident, as well as instances of extended rod growth mixed with the eroding plate structure. Whereas at $250 \mu\text{m/s}$ rods were beginning to evolve within the plane of the plate, by $500 \mu\text{m/s}$ growth has become possible outside of the plane of the plate as well. The conversion to rod dominated growth continues through $700 \mu\text{m/s}$, and is generally complete by $950 \mu\text{m/s}$. The microstructure at $950 \mu\text{m/s}$ contains little to no remnant plate structure and is instead dominated by rod growth which is weakly aligned with the growth direction. From these observations, the flake to fiber transition is said to occur in two stages: an initial stage which begins between the velocities of 100 and $250 \mu\text{m/s}$ and involves breakup of the broad plate structure and in-plane rod formation, and a final stage which begins near the velocity $500 \mu\text{m/s}$ and involves the initiation of out-of-plane rod growth and continues until the transition to rod-dominated growth is complete by $950 \mu\text{m/s}$.

Evolving microstructural features of the eutectic with increasing velocity are representative

of the intrinsic growth mechanisms being selected. Higher magnification images of the evolving eutectic structures are shown in Fig. 5. The sequence of morphologies observed upon growth from 20 to 250 $\mu\text{m/s}$ suggests that the transition involves a selection phenomenon based on the hierarchy of interfacial properties. In particular, it is important to note the manner of plate breakup at these velocities. The appearance of gaps, holes, and other convoluted formations within otherwise well defined plate structures indicates that, while growth is still generally confined to within the $\{111\}$ plane of the plate, multiple in-plane growth directions are operative. In effect, growth is becoming more nearly isotropic in the plane of the plate. This progression leads to the development of increasingly thin skeletal plates and, ultimately, the formation of rodlike structures within the plane of the plate. Additionally, it becomes apparent at 500 $\mu\text{m/s}$ that out-of-plane growth sites are active at select locations in the structure. For example, it is seen in Fig. 5 that silicon protrusions begin to form on plate faces by 500 $\mu\text{m/s}$ and grow approximately perpendicular to the plate face. These isolated instances could be a result of locally higher eutectic spacings and subsequently increased local solute gradient. Silicon protrusion formation occurs along with an increased incidence of extended fibrous growth, both of which dominate the microstructure by 950 $\mu\text{m/s}$, as shown in Figs. 1 and 5. A more extensive example of protrusion formation is visible in Fig. 6. Here, silicon fingers grow out of plate faces at a defined spacing. As the remnant plate structure disappears in the eutectic, it becomes difficult to determine whether predominant rod growth is truly three-dimensional in nature or remains confined within the $\{111\}$ silicon planes. However, with substantial out-of-plane protrusion formation on crystal facets at velocities as low as 500 $\mu\text{m/s}$, it is reasonable to assert that localized morphological instability plays a vital role in the onset and completion of the flake to fiber transition.

If the operative flake to fiber transition mechanism involves the activation of additional growth sites with increasing solidification velocity, then one would expect the predominant silicon fiber growth axis to change with velocity as well. Favored in-plane growth directions would be prevalent at low velocities, while additional in-plane and ultimately out-of-plane fibrous growth directions would become common with increasing velocity. The crystallographic

orientation of quench modified silicon fibers has been studied to some extent. At lower velocities, Lu and Hellawell(11) and Shamsuzzoha and Hogan(28) found a tendency for plates to elongate in $\langle 211 \rangle$ directions and Lu and Hellawell similarly noted that quench modified fibers assumed a general growth direction of approximately $\langle 211 \rangle$. Steen and Hellawell(27) observed the occurrence of $\langle 211 \rangle$ fiber growth, but reported the most common growth direction was $\langle 110 \rangle$. Jenkinson and Hogan(8), meanwhile, found the dominant fiber axis to be consistently $\langle 100 \rangle$. While a relationship between velocity and fiber orientation was not reported by these authors, the variable orientation results support the possibility of fiber growth in multiple directions with a predominant growth direction that evolves with changing velocity as is consistent with the present findings. Furthermore, the observation of dominant $\langle 100 \rangle$ growth in the study by Jenkinson and Hogan is significant in that it represents an out-of-plane growth direction, proving that out-of-plane growth may increase to the extent that it dominates the quench modified microstructure.

The evolution of silicon morphology with increasing velocity is governed by distinct levels of silicon interfacial energy. At low velocities (and low undercoolings), the faceted, microscopically smooth morphology of flake silicon is preferred.(29) Increased undercooling at higher velocities precipitates the activation of additional in-plane growth directions leading to the formation of in-plane rod structures, and, at the highest velocities, the activation of out-of-plane growth directions in select locations leads to the formation of rod-type silicon protrusions approximately perpendicular to plate faces. The appearance of these fibrous growth structures eventually dominates the eutectic structure, completing the flake to fiber transition. It should be noted that an earlier study by Khan and Elliott(23) reported a discrete decrease in undercooling above $500 \mu m/s$. While this result does not appear to be consistent with the present observation of a gradual flake to fiber transition occurring over a wide range of velocities beginning below $250 \mu m/s$, it may be indicative of the onset of out-of-plane growth defining the beginning of the second stage of the transition. The erosion of the plate structure and subsequent onset of out-of-plane growth will contribute to a change in the intrinsic length scale. While the evident microstructural morphology changes gradually, it is possible that a

relatively abrupt change in the dominant diffusion length scale from interplate to interfiber distances corresponds to a sudden decrease in local undercooling.

The conventional spacing parameters examined closely follow a $\lambda^2 V = \text{constant}$ relationship and differ only by a constant multiplier, as shown in Fig. 3. As expected, the parameters do not indicate the flake to fiber transition in any way and, hence, do not reflect the evolving silicon microstructure. The spread in spacing values between different parameters is similar to the spread in previously reported spacing results. Due to the spread in magnitude of the results, the degree to which any of the spacing parameters approximate the true average interfiber diffusion distance during solidification is in doubt. Each of the three parameters has its drawbacks. The subjective, selective nature of the $\lambda_{||}$ measurement suggests it is not an accurate representation of the whole microstructure. The λ_A method suffers from the difficulty of determining which particles are separate, especially in the flake structure. Finally, the λ_L results will be biased upwards due to a reliance on random alignments that do not coincide with the minimum distance between two adjacent particles. Furthermore, all the spacing parameters employed suffer from the limitations of the two-dimensional microstructure projections they are based on.

Unlike the spacing parameters, the aspect ratio data shown in Fig. 4 indicate a transition with changing velocity. The initial abrupt change in slope of the graph occurs at about $135 \mu\text{m/s}$, indicating the data may be reflecting the onset of the flake to fiber transition, which was determined to occur between 100 and $250 \mu\text{m/s}$. This behavior is reasonable, given that the transition involves the conversion of high aspect ratio broad flakes to low aspect ratio skeletal plates and rod structures. Furthermore, the data show a second and opposite slope change near $880 \mu\text{m/s}$, which is attributed to the end of the flake to fiber transition where the conversion to rods is largely complete and the aspect ratio ceases to drop. The transitions shown by the aspect ratio data are useful in that they define objective measures for determining the onset and completion velocities of the flake to fiber transition, allowing consistent identification of the transition.

Figure 5 Details of the quench modification process in an Al-13wt% Si alloy. At velocities from 10 to 250 $\mu\text{m/s}$ the transition is evident in the breakup of plates, with rodlike structures evolving within the envelope of the plate. At higher velocities, the next stage of the transition becomes evident as out-of-plane rod growth begins around 500 $\mu\text{m/s}$ (shown by arrows) and a shift to rod dominance occurs by 950 $\mu\text{m/s}$.

Figure 6 Occurrence of morphological instability of plate faces in a sample grown at $950 \mu\text{m/s}$.

Conclusions

1. Qualitative observations suggest the flake to fiber transition occurs within two overlapping but distinct intervals, defined below.
 - (a) $100\text{--}500 \mu\text{m/s}$: This stage is characterized by a rapid increase in the in-plane breakup of broad silicon plates to skeletal plate and rodlike structures. This occurrence signifies the activation of additional in-plane growth directions due to increased undercooling.
 - (b) $500\text{--}950 \mu\text{m/s}$: This stage begins with the onset of out-of-plane silicon growth, leading to fibrous rod structures and three-dimensional growth. There is also a marked increase in incidences of silicon morphological instabilities growing out of sheer plate faces. This stage terminates with the completion of the flake to fiber transition.

Furthermore, despite descriptions of flake to fiber onset and completion velocities, a range of silicon growth mechanisms are present at all velocities studied ($10\text{--}2000\text{ }\mu\text{m/s}$), and the flake to fiber transition, as reported, simply reflects those velocities at which the dominant growth mechanism changes. Al-Si eutectic growth may, therefore, be more aptly described with a model that incorporates flake, fiber, and other mechanisms of growth at all velocities in this range.

2. The appearance of out-of-plane silicon protrusions midway through the flake to fiber transition suggests that the occurrence of three-dimensional out-of-plane silicon growth is vital to the formation of the modified fibrous microstructure. The flake to fiber transition may then be said to involve a transition from the primarily two-dimensional growth of eutectic silicon as bulky plates at low velocities to primarily three-dimensional growth of eutectic silicon as globular fibers at high velocities. A salient feature of this transformation is the change from flake-dominated diffusion distances to fiber-dominated diffusion distances.
3. None of the conventional spacing parameters are effective in indicating the flake to fiber transition. Therefore, more descriptive parameters must be included in any model meant to describe the microstructure and growth modes present in the silicon eutectic.
4. The relationship between average particle aspect ratio and solidification velocity objectively defines a velocity for both the beginning and end of the flake to fiber transition. Stages of the transition are indicated by abrupt changes in the slope of the aspect ratio versus velocity relationship. In this manner, the beginning of the flake to fiber transition in the Al-Si eutectic alloy was determined to be $135\text{ }\mu\text{m/s}$, and the end was determined to be $880\text{ }\mu\text{m/s}$. While this range compares well with the qualitative assessment based on morphology, it is a more objective measure of the flake to fiber transition and, therefore, more suitable for use in the prediction of solidification microstructure.

Acknowledgements

This work was made possible by support from the National Science Foundation (NSF), Division of Materials Research, under Award No. 0237566. The authors would also like to thank P. Matlage for experimental assistance.

References

- [1] Nabil Fat-Halla. Structure, mechanical properties and fracture of aluminum alloy a-356 modified with aluminum-strontium (al-5sr) master alloy. *Journal of materials science*, 22(3):1013, 1987.
- [2] N. Fat-Halla. Structural modification of aluminum-silicon eutectic alloy by strontium and its effect on tensile and fracture characteristics. *Journal of materials science*, 24(7):2488, 1989.
- [3] Mahmoud Fouad Hafiz, Nabil K. Fat-Halla, and Sherif B. Moshref. The effect of casting parameters on the modification process of a commercial hypoeutectic aluminum-silicon alloy. *Zeitschrift fr Metallkunde*, 81(1):70, 1990.
- [4] Mahmoud Hafiz and Toshiro Kobayashi. Tensile properties influencing variables in eutectic al-si casting alloys. *Scripta metallurgica et materialia*, 31(6):701, 1994.
- [5] M. Zuo, X. F. Liu, and Q. Q. Sun. Effects of processing parameters on the refinement of primary si in a390 alloys with a new al-si-p master alloy. *Journal of Materials Science*, 44(8):1952–1958, 2009.
- [6] M. Garat, G. Laslaz, S. Jacob, P. Meyer, P. H. Guerin, and R. Adam. State-of-art use of sb-, na- and sr-modified al-si casting alloys. *Transactions of the Amercian Foundrymen's Society*, 100:821–832, 1992.
- [7] Shu Zu Lu and A. Hellawell. The mechanism of silicon modification in aluminum-silicon alloys: impurity induced twinning. *Metallurgical transactions. A, Physical metallurgy and materials science*, 18A(10):1721, 1987.

- [8] D. C. Jenkinson and L. M. Hogan. Modification of aluminum-silicon alloys with strontium. *Journal of Crystal Growth*, 28(2):187, 1975.
- [9] A. K. Dahle, K. Nogita, S. D. McDonald, C. Dinnis, and L. Lu. Eutectic modification and microstructure development in alsi alloys. *Materials Science and Engineering A*, 413-414:243–248, 2005.
- [10] M. G. Day and A. Hellawell. Microstructure and crystallography of aluminum-silicon eutectic alloys. *Proceedings - Royal Society. Mathematical, physical and engineering sciences*, 305(1483):473, 1968.
- [11] Shu Zu Lu and Angus Hellawell. Growth mechanisms of silicon in aluminum-silicon alloys. *Journal of Crystal Growth*, 73(2):316, 1985.
- [12] N. Tenekedjiev and J. E. Gruzleski. Thermal analysis of strontium treated hypoeutectic and eutectic aluminum-silicon casting alloys. *Transactions of the Amercian Foundry Society*, 99:1–6, 1991.
- [13] H. Jiang, J. H. Sokolowski, M. B. Djurdjevic, and W. J. Evans. Recent advances in automated evaluation and on-line prediction of al-si eutectic modification level. *Transactions of the Amercian Foundry Society*, 108:505–510, 2000.
- [14] Francisco C. Robles Hernandez and Jerry H. Sokolowski. Novel image analysis to determine the si modification for hypoeutectic and hypereutectic al-si alloys. *Journal of Metals*, 57, 2005.
- [15] R. E. Napolitano and L. G. England. High-rate growth-mode transitions in al-si eutectics. *Solidification of Aluminum Alloys*, pages 445–451, 2004.
- [16] H. Fredriksson, M. Hillert, and N. Lange. Modification of aluminum-silicon alloys by sodium. *Journal of the Institute of Metals*, 101(Nov):285, 1973.

- [17] B. Toloui and A. Hellawell. Phase separation and undercooling in aluminum-silicon eutectic alloy - the influence of freezing rate and temperature gradient. *Acta Metallurgica*, 24(6):565, 1976.
- [18] R. Elliott and S. M. D. Glenister. The growth temperature and interflake spacing in aluminum-silicon eutectic alloys. *Acta Metallurgica*, 28(11):1489, 1980.
- [19] O. Atasoy. Effects of unidirectional solidification rate and composition on interparticle spacing in aluminum-silicon eutectic alloys. *Aluminium*, 60(4):275, 1984.
- [20] L. M. Hogan and H. Song. Interparticle spacings and undercoolings in aluminum-silicon eutectic microstructures. *Metallurgical transactions. A, Physical metallurgy and materials science*, 18A(4):707, 1987.
- [21] Junmin Liu, Yaohe Zhou, and Baolu Shang. Lamellar eutectic stable growth - ii. experiment on al-si eutectic. *Acta metallurgica et materialia*, 38(7):1321, 1990.
- [22] P. Magnin, J. T. Mason, and R. Trivedi. Growth of irregular eutectics and the aluminum-silicon system. *Acta metallurgica et materialia*, 39(4):469, 1991.
- [23] S. Khan and R. Elliott. Quench modification of aluminum-silicon eutectic alloys. *Journal of Materials Science*, 31(14):3731–3737, 1996.
- [24] W. Wolczynski, B. Billia, and K. Rabczak. Interlamellar spacing in al-si eutectic growth controlled by temperature gradient. *Materials science forum*, 215(Solidification):323, 1996.
- [25] R. Cuprys, B. Major, and W. Wolczynski. Transition of flake into fiber structure in eutectic al-si. *Materials science forum*, 329(Solidification):161, 2000.
- [26] E. Guzik and D. Kopycinski. Modeling structure parameters of irregular eutectic growth: modification of magnin-kurz theory. *Metallurgical and Materials Transactions; A; Physical Metallurgy and Materials Science*, 37A(10):3057, 2006.
- [27] H. A. H. Steen and A. Hellawell. Structure and properties of aluminum-silicon eutectic alloys. *Acta Metallurgica*, 20(3):363, 1972.

- [28] M. Shamsuzzoha and L. M. Hogan. Crystal morphology of unmodified aluminum-silicon eutectic microstructures. *Journal of Crystal Growth*, 76(2):429, 1986.
- [29] L. Li, Y. D. Zhang, C. Esling, Z. H. Zhao, Y. B. Zuo, H. T. Zhang, and J. Z. Cui. Formation of feathery grains with the application of a static magnetic field during direct chill casting of al-9.8wt *Journal of Materials Science*, 44(4):1063–1068, 2009.

**THE EFFECT OF THE FLAKE-FIBER TRANSITION IN SILICON
MORPHOLOGY ON THE TENSILE PROPERTIES OF
AL-SI EUTECTIC ALLOYS**

A paper submitted to Materials Science and Engineering A

T. Hosch and R. E. Napolitano

Abstract

The combined and separate effects of microstructural scale and silicon phase morphology on the mechanical properties of Al-Si eutectic alloys are investigated here. The Bridgman-type gradient-zone directional solidification method is employed to produce as-cast structures characteristic of the full range of practical (i.e. casting) growth velocities, and the corresponding mechanical properties are characterized by uniaxial tension testing. The results are analyzed in light of previously reported microstructural changes associated with the flake-fiber or “quench modification” transition. Both tensile strength and elongation were found to increase with solidification rate, as expected, based on the corresponding refinement of the eutectic structure. However, the flake-fiber transition was found to have a particular relevance here with regard to each of these properties. The velocity dependence of the tensile strength was found to become more severe at a pulling speed of approximately $250\text{ }\mu\text{m/s}$, corresponding to the transition from a flake-dominant to a fiber-dominant structure, as we have previously detailed. The elongation was found to exhibit a maximum at approximately $600\text{ }\mu\text{m/s}$, corresponding to completion of the flake-fiber transition. In addition, the overall variability of tensile properties was observed to decrease significantly with the transition to the fibrous structure. These

results emphasize the importance of understanding and controlling the flake-fiber transition that occurs with increasing solidification rate in Al-Si eutectic alloys.

Introduction

Al-Si eutectic alloys are some of the most commercially important casting alloys, with numerous aerospace and automotive applications that take advantage of the corrosion resistance, excellent castability, and high strength-to-weight ratio intrinsic to aluminum alloys. The high strength-to-weight ratio contributes to a significant weight-saving potential for these alloys, making them of increasing importance considering recent rapidly rising energy prices and the subsequent focus on energy efficiency. The irregular nature of the Al-Si eutectic gives rise to a diverse array of microstructural features that can be controlled through adjusting various solidification parameters. In particular, the mechanical properties of the casting may be improved via microstructural modification, a process in which a refined fibrous silicon structure is obtained in lieu of a coarse flake structure. Modification in the Al-Si system was recently reviewed by Hegde and Prabhu (1). In short, microstructural modification may be obtained by additions of trace impurity elements (2–5) or increased solidification velocity (6–9). In either case, the goal of modification is to improve the mechanical properties of the alloy.

The mechanical property benefits of microstructural modification in the Al-Si system arise from both the decreasing scale of the eutectic silicon microstructure and the beneficial effects of the transition to a fibrous silicon morphology. Early investigations by Justi and Bragg (10; 11) and later by Telli and Kısakürek (12) found that both hardness and strength increase with decreasing eutectic scale for the flake silicon morphology. Subsequent investigations found that the relationship between hardness and strength varied significantly based on the underlying silicon morphology involved (13–16), highlighting the contribution of silicon morphology separate from the microstructural scale. Indeed, the beneficial effect on tensile strength arising from the fibrous morphology obtained from impurity modification has been well characterized by a number of researchers (17–23). Additionally, both scale and morphology have been observed to improve fracture toughness in the Al-Si system, as exemplified in a study by Hafiz

and Kobayashi (24), where a fibrous silicon structure displayed increased toughness compared to a flake structure, though the toughness of both structures improved with decreased scale as well.

The specific effect that the flake-fiber transition has on Al-Si tensile properties, apart from the beneficial effects of the decreasing microstructural scale at higher solidification velocities, has not been well characterized to date due to a number of issues. First, the flake-fiber transition itself had not been well quantified in the literature until recently, making it difficult to correlate mechanical properties with underlying microstructural features. However, a recent study by the present authors has detailed the significant microstructural changes that occur during modification of the pure binary alloy with increasing velocity (6). In particular, it was found that the microstructural changes that occur during the flake-fiber transition may be described as occurring in distinct but overlapping stages in the velocity range of 100-1000 $\mu\text{m/s}$. Further, the velocity at which the morphological changes associated with the flake-fiber transition occurs was identified using a quantitative metric, the average particle aspect ratio, which facilitates comparison of bulk material properties with internal microstructural features. The second issue affecting the correlation of the flake-fiber transition with changes in mechanical properties is the lack of mechanical property data from within the specific velocity range of the transition; in general, the tensile properties of only a few points within the transition regime have been measured, limiting the possibility of capturing any transition-specific behavior. The present study addresses this limitation by examining the tensile behavior of an Al-Si eutectic alloy at multiple solidification velocities throughout the high growth-rate flake-fiber transition. The separate contributions of microstructural scale and morphology to the tensile strength are investigated by considering the distinct Al-Si morphology regimes that have been recently identified.

Experimental

Directional solidification experiments were performed with an Al-12 wt%Si alloy in a Bridgman-type gradient-zone furnace using pulling velocities (V) of 20, 80, 250, 600, and

Table 1 Temperature gradients measured in the present Bridgman-type directional solidification apparatus.

V ($\mu\text{m/s}$)	G (K/mm)
10	7
50	10
100	10
500	10
700	10
1000	14

1000 $\mu\text{m/s}$. This range of V was chosen to emphasize the microstructural transition. The temperature gradient (G), imposed during directional solidification, was measured using a thermocouple embedded in a test sample at several solidification velocities, as listed in Table 1. Samples were melted in an alumina crucible of 5.5 mm inner diameter and 8.0 mm outer diameter, solidified over a distance of 125 mm, and then quenched to 25°C in a water-cooled bath containing a Ga-In-Sn alloy. Sample sections approximately 6.3 cm (2.5 inches) in length were machined into tension test specimens with a 3.2 cm (1.25 inch) gauge length, 3.65 mm inner diameter, and 5.5 mm outer diameter in accordance with ASTM Standard E8. Four tension test specimens for $V=20$ $\mu\text{m/s}$ and $V=80$ $\mu\text{m/s}$, three for $V=250$ $\mu\text{m/s}$, and two for $V=600$ $\mu\text{m/s}$ and $V=1000$ $\mu\text{m/s}$ were obtained from the directionally solidified samples. For each of these specimens, tension testing was performed to fracture in displacement control mode with a crosshead speed of 0.2 mm/min. Strain measurements were taken using an extensometer.

Results and Discussion

Tension test results are reported in Table 2. The results indicate that ultimate tensile strength (UTS) is significantly affected by solidification velocity, monotonically increasing by 37% with solidification rate over the full range, from 162MPa at 20 $\mu\text{m/s}$ to 222 MPa at 1000 $\mu\text{m/s}$. Elongation increases rapidly to a maximum of 23.8% at 600 $\mu\text{m/s}$, but shows a significant decrease at 1000 $\mu\text{m/s}$.

One of the most significant trends noted in the tension test data with increasing velocity

is the increase in UTS. It is generally understood that the average eutectic spacing varies as $V^{-1/2}$, and this dependency has been clearly shown to hold for Al-Si eutectics (see, for example (25; 26)), with a range of proportionality constants arising from the various difficulties and methods associated with characterizing an irregular eutectic structure, as we have previously reviewed (6). This decrease in microstructural scale is expected to increase strength, which is observed here. Indeed, the 37% overall increase in tensile strength throughout the range of velocities studied is consistent with previous reports of mechanical property variation with directional solidification velocity (12; 13). The salient point to make here, however, is that there are two distinct regions with different velocity dependence, as indicated by the UTS data, shown in Fig. 1. The intersection of the superimposed trend-lines reveals a transition in tensile behavior at approximately $V=250 \mu\text{m/s}$, with the higher velocity regime exhibiting a stronger temperature dependence.

The bimodal UTS behavior is compared with our previously reported measurements of silicon aspect ratio and eutectic spacing (6) in Fig. 1, and the essential nature of the flake-fiber microstructural transition is illustrated by the sequence shown in Fig. 2 (6). The change in slope of the aspect ratio versus velocity relationship above $100 \mu\text{m/s}$ signifies the onset of the morphological transition, where the broad flakes that dominate the structure at lower growth rates begin to show a tendency to break up into finger-like features within the skeletal plate structure (6). Over the same velocity range, the average interparticle spacing in the alloy is seen to decrease continuously, indicating that the microstructural scale is not sensitive to the morphological transition but is, rather, related to the characteristic lengths associated with diffusion and capillarity.¹ Therefore, it can be ascertained that the monotonic structural refinement arising from increasing growth velocity, while contributing to a corresponding increase in strength, is not responsible for the discontinuity in the UTS observed at approximately $250 \mu\text{m/s}$. The present mechanical testing results can be seen as reflecting the concordant effects of both the decreasing microstructural scale and the transition in silicon morphology. Each of

¹While the characteristic microstructural length is not specifically sensitive to the specific morphology, the measurement of such a characteristic length is quite sensitive. Indeed, the methods used for such measurement must be carefully chosen in accord with the prevailing structure if the measurement is intended to be generally representative.

these structural changes exerts an important influence on tensile strength, but it is apparent that the transition in UTS behavior at $250 \mu\text{m/s}$ arises as a result of the increasing fibrous character of the microstructure. It should also be noted that while the flake-fiber transition exhibits an onset velocity of approximately $100 \mu\text{m/s}$ (6), the transition in tensile strength behavior observed presently occurs at $250 \mu\text{m/s}$. This velocity is consistent with the identified division between the first and second stages of the flake-fiber transition, where out-of-plane rod growth and a truly fibrous morphology begins to dominate the structure (6). The coincident observation of an increased velocity dependence of the UTS suggests that a certain degree of fibrous dominance is required before bulk properties are affected by the transition.

Another effect on UTS with changing velocity can be noted in the stress-strain curves shown in Fig. 3. In addition to the increase in strength with increasing solidification velocity, the stress-strain curves for $20 \mu\text{m/s}$ reveal significant variability, particularly with respect to elongation and UTS. This finding cannot be attributed to the effects discussed above, as there should be no differences in either scale or morphology among the different samples solidified at $20 \mu\text{m/s}$. However, this variable strength behavior may be explained by the coarseness of the microstructure in comparison to the sample dimensions. At $20 \mu\text{m/s}$, though the average interflake spacing is only $5\text{-}10 \mu\text{m}$, which is small compared to the sample dimensions of 3.65 mm , it is not unusual for high aspect ratio silicon flakes to extend in length many times the eutectic spacing. Therefore, it would be possible for a specific large silicon flake to be oriented in a particularly undesirable orientation such that it significantly affects the observed fracture behavior and the measured tensile properties, given the current sample dimensions. Indeed, this seems to be the case, as large planar features are observed on the fracture surface for the $V=20 \mu\text{m/s}$ specimen, as shown in Fig. 4. These large flat areas appear to be the result of fracture along the broad face of a silicon flake. This result highlights the importance of obtaining a modified or refined silicon microstructure, especially in any thin-section areas of a casting.

In light of the flake-induced fracture discussed above, we expect that the flake-fiber transition will give rise to an increase in elongation. Indeed, the test results in Table 2 reveal such a

behavior, where sample ductility improves with solidification velocity up to a maximum at 600 $\mu\text{m/s}$. Above this velocity, the transition to fibrous-structure dominance is generally complete, and additional increases in velocity are accompanied by little benefit to elongation but a continued increase in strength, arising from continued refinement. Additionally, the stress-strain curves shown in Fig. 3 demonstrate a marked decrease in fracture elongation variability with increased solidification velocity. The improved fracture reproducibility appears to be associated with the transition to a fibrous structure. This has been observed previously by Steen and Hellawell (27).

Specimen fracture surfaces, shown in Fig. 4, provide a more detailed account of the interaction of sample microstructure and tensile properties. The three velocities shown, 20, 250, and 1000 $\mu\text{m/s}$, represent a flake structure, a flake/fiber transitional structure, and a fibrous structure, respectively. At low magnifications, the widely different microstructural scale and morphology obtained from these three solidification velocities is evident. As mentioned previously, the 20 $\mu\text{m/s}$ surface is scarred with flat fracture planes likely corresponding to broad faces of large silicon flakes. It is clear from the intermediate magnification images that neither the 250 $\mu\text{m/s}$ nor the 1000 $\mu\text{m/s}$ specimens display such large, flat fracture features. However, apart from the lack of large fracture planes, the rough appearance of the 250 $\mu\text{m/s}$ fracture surface seems more like the 20 $\mu\text{m/s}$ sample than the 1000 $\mu\text{m/s}$ sample. Indeed, the rounded, dimpled surface of the 1000 $\mu\text{m/s}$ sample does not appear to contain any fracture features associated with the flake morphology. At high magnifications, the mixed flake/fiber character of the 250 $\mu\text{m/s}$ sample becomes clear as a broad flake fracture plane is seen alongside some limited areas of dimpled fibrous fracture. The high magnification image of the 1000 $\mu\text{m/s}$ fracture also displays an interesting fracture morphology, which may be the result of fracture along fibers oriented parallel to the fracture surface. In any case, it is clear that the dimpled fibrous morphology provides more desirable fracture behavior than the irregular flake structure, which contributes to the appearance of large, flat fracture planes in the structure.

Table 2 Results of tension tests performed on Al-Si eutectic samples solidified at different velocities.

V ($\mu\text{m/s}$)	UTS (MPa)	El (%)
20	162 ± 13	9 ± 4
80	179 ± 5	15 ± 6
250	190 ± 11	18 ± 4
600	207 ± 3	24 ± 1
1000	222 ± 4	13 ± 3

Summary and Conclusions

In addition to the expected increase in strength arising from the structural refinement that accompanies higher solidification velocity, the change in eutectic structure associated with the flake-fiber morphological transition has several distinct effects on the mechanical properties of an Al-Si cast alloy, as summarized below:

- The UTS increases monotonically with solidification velocity from 20 to 1000 $\mu\text{m/s}$, but the contribution from the flake-fiber transition gives rise to a discontinuous increase in the velocity dependence, which is associated with the increasing influence of the fibrous silicon morphology above 250 $\mu\text{m/s}$.
- Elongation to fracture increases with solidification velocity to a maximum at 600 $\mu\text{m/s}$, indicating an end to the effects of the flake-fiber transition, which is nearly complete at that velocity.
- The reproducibility of tensile behavior improves significantly at velocities corresponding to the flake-fiber transition.

The benefits of the fibrous morphology outlined herein emphasize the importance of understanding and controlling the high growth-rate flake-fiber transition that occurs in Al-Si alloys. Proper casting design depends both on the effective prediction of microstructure as a function of solidification parameters, which was examined in our previous study (6), and on strong knowledge of the relationship between alloy microstructure and corresponding mechanical properties, which we have addressed in the present study. While the transition to

a modified structure in Al-Si alloys may be in many ways a gradual transition, the discrete microstructural and mechanical changes associated with the formation of a modified structure confer additional importance upon precisely controlling solidification microstructure in high performance castings.

Acknowledgements

This work was made possible by support from the National Science Foundation (NSF), Division of Materials Research, under Award No. 0237566.

References

- [1] S. Hegde and K. N. Prabhu. Modification of eutectic silicon in al-si alloys. *Journal of Materials Science*, 43(9):3009–3027, 2008.
- [2] M. Garat, G. Laslaz, S. Jacob, P. Meyer, P. H. Guerin, and R. Adam. State-of-art use of sb-, na- and sr-modified al-si casting alloys. *Transactions of the Amercian Foundrymen's Society*, 100:821–832, 1992.
- [3] Shu Zu Lu and A. Hellawell. The mechanism of silicon modification in aluminum-silicon alloys: impurity induced twinning. *Metallurgical transactions. A, Physical metallurgy and materials science*, 18A(10):1721, 1987.
- [4] D. C. Jenkinson and L. M. Hogan. Modification of aluminum-silicon alloys with strontium. *Journal of Crystal Growth*, 28(2):187, 1975.
- [5] A. K. Dahle, K. Nogita, S. D. McDonald, C. Dinnis, and L. Lu. Eutectic modification and microstructure development in alsi alloys. *Materials Science and Engineering A*, 413-414:243–248, 2005.
- [6] T. Hosch, L. G. England, and R. E. Napolitano. Analysis of the high growth-rate transition in al-si eutectic solidification. *Journal of Materials Science*, 44(18):4892–4899, 2009.

- [7] M. G. Day and A. Hellawell. Microstructure and crystallography of aluminum-silicon eutectic alloys. *Proceedings - Royal Society. Mathematical, physical and engineering sciences*, 305(1483):473, 1968.
- [8] Shu Zu Lu and Angus Hellawell. Growth mechanisms of silicon in aluminum-silicon alloys. *Journal of Crystal Growth*, 73(2):316, 1985.
- [9] R. E. Napolitano and L. G. England. High-rate growth-mode transitions in al-si eutectics. *Solidification of Aluminum Alloys*, pages 445–451, 2004.
- [10] S. Justi and R. H. Bragg. Vickers hardness measurements of unidirectionally solidified al-si eutectic alloy grown at different rates. *Metallurgical Transactions a-Physical Metallurgy and Materials Science*, 7(12):1954–1957, 1976.
- [11] S. Justi and R. H. Bragg. Tensile properties of directionally solidified al-si eutectic. *Metallurgical Transactions a-Physical Metallurgy and Materials Science*, 9(4):515–518, 1978.
- [12] A. I. Telli and S. E. Kisakurek. Effect of antimony additions on hardness and tensile properties of directionally solidified al-si eutectic alloy. *Materials Science and Technology*, 4(2):153–156, 1988.
- [13] F. Vnuk, M. Sahoo, R. Vandemerwe, and R. W. Smith. Hardness of al-si eutectic alloys. *Journal of Materials Science*, 14(4):975–982, 1979.
- [14] F. Yilmaz and R. Elliott. The microstructure and mechanical-properties of unidirectionally solidified al-si alloys. *Journal of Materials Science*, 24(6):2065–2070, 1989.
- [15] A. Ourdjini, F. Yilmaz, Q. S. Hamed, and R. Elliott. Microstructure and mechanical-properties of directionally solidified al-si eutectic alloys with and without antimony. *Materials Science and Technology*, 8(9):774–776, 1992.
- [16] S. Khan, A. Ourdjini, Q. S. Hamed, M. A. A. Najafabadi, and R. Elliott. Hardness and mechanical property relationships in directionally solidified aluminum-silicon eutectic

- alloys with different silicon morphologies. *Journal of Materials Science*, 28(21):5957–5962, 1993.
- [17] N. Fatahalla, M. Hafiz, and M. Abdulkhalek. Effect of microstructure on the mechanical properties and fracture of commercial hypoeutectic al-si alloy modified with na, sb and sr. *Journal of Materials Science*, 34(14):3555–3564, 1999.
- [18] Nabil Fat-Halla. Structure, mechanical properties and fracture of aluminum alloy a-356 modified with aluminum-strontium (al-5sr) master alloy. *Journal of materials science*, 22(3):1013, 1987.
- [19] N. Fat-Halla. Structural modification of aluminum-silicon eutectic alloy by strontium and its effect on tensile and fracture characteristics. *Journal of materials science*, 24(7):2488, 1989.
- [20] Mahmoud Fouad Hafiz, Nabil K. Fat-Halla, and Sherif B. Moshref. The effect of casting parameters on the modification process of a commercial hypoeutectic aluminum-silicon alloy. *Zeitschrift fr Metallkunde*, 81(1):70, 1990.
- [21] Mahmoud Hafiz and Toshiro Kobayashi. Tensile properties influencing variables in eutectic al-si casting alloys. *Scripta metallurgica et materialia*, 31(6):701, 1994.
- [22] M. F. Hafiz and T. Kobayashi. A study on the microstructure-fracture behavior relations in al-si casting alloys. *Scripta metallurgica et materialia*, 30(4):475–480, 1994.
- [23] M. Zuo, X. F. Liu, and Q. Q. Sun. Effects of processing parameters on the refinement of primary si in a390 alloys with a new al-si-p master alloy. *Journal of Materials Science*, 44(8):1952–1958, 2009.
- [24] M. F. Hafiz and T. Kobayashi. Fracture toughness of eutectic al-si casting alloy with different microstructural features. *Journal of Materials Science*, 31(23):6195–6200, 1996.
- [25] Y. Bayraktar, D. Liang, and H. Jones. The effect of growth velocity and temperature

gradient on growth characteristics of matrix eutectic in a hypereutectic aluminium-silicon alloy. *Journal of Materials Science*, 30(23):5939–5943, 1995.

- [26] M. Gunduz, H. Kaya, E. Kadirli, and A. Ozmen. Interflake spacings and undercoolings in al-si irregular eutectic alloy. *Materials Science and Engineering A*, 369(1-2):215–229, 2004.
- [27] H. A. H. Steen and A. Hellawell. Structure and properties of aluminum-silicon eutectic alloys. *Acta Metallurgica*, 20(3):363, 1972.

Figure 1 Comparison of the ultimate tensile strength measured in the present study and the aspect ratio and spacing results (λ_A) reported in (6). The tensile strength increases significantly with increasing solidification rate throughout the velocity range studied. Note the transition in slope of the tensile strength data that occurs at 250 $\mu\text{m/s}$. Similarly, the aspect ratio data display a transition in slope near 125 $\mu\text{m/s}$. The spacing is seen to decrease continuously with increasing solidification velocity.

Figure 2 Al-Si eutectic structures after directional solidification at velocities of 20, 250, and 950 $\mu\text{m/s}$ (6). These structures represent flake (left), mixed flake/fiber (center), and fiber (right) morphologies, respectively.

Figure 3 The stress-strain behavior of samples directionally solidified at 20, 80, and 600 $\mu\text{m/s}$. The velocities represent predominantly flake (20 and 80 $\mu\text{m/s}$) and predominantly fibrous (600 $\mu\text{m/s}$) morphologies, respectively.

Figure 4 (a) Low, (b) intermediate, and (c) high magnification images of tension test specimen fracture surfaces solidified at 20, 250, and 1000 $\mu\text{m/s}$.

MORPHOLOGICAL INSTABILITY OF SILICON DURING DIRECTIONAL SOLIDIFICATION OF AN AL-SI EUTECTIC ALLOY

A paper to be submitted to Acta Materialia

T. Hosch and R. E. Napolitano

Abstract

The final stage of the high growth-rate flake to fiber transition in Al-Si eutectic alloys gives rise to the development of intricate out-of-plane protrusion formations in the silicon phase. The growth mechanisms behind these formations are investigated here in directionally solidified Al-7wt% Si and Al-20wt% Si alloys by adapting a Mullins-Sekerka type perturbed interface stability analysis to the system, which incorporates the effects of the irregular Al-Si interface shape and the anisotropy of the solid silicon/liquid Al-Si interface. Measurements of silicon equilibrium shape particles were conducted to provide an estimate of system anisotropy. The stability analysis was found to be effective in predicting the length scale behavior of silicon plate breakup throughout the flake to fiber transition, and it was concluded that out-of-plane silicon protrusion formation is vital to the formation of a modified fibrous structure at high solidification rates.

Introduction

Alloys based on the Al-Si eutectic system are widely used in lightweight shape-casting applications, offering a range of mechanical properties coupled with low density, good corrosion resistance, and excellent castability. In addition, heat-treatable Al-Si casting alloys arising from

additions of Mg and/or Cu provide high specific strengths desirable for lightweight structural or mechanical applications. The typical tensile strength of Al-Si casting alloys with a flake silicon microstructure ranges from 120 to 180 MPa with limited ductility. However, the tensile strength of Al-Si casting alloys can be improved roughly 50% with better than a threefold increase in ductility by the use of “modification” practices, which alter the structure of the silicon phase from the normal coarse-flake morphology to a fine fibrous morphology.(1–5)

In practice, this structural modification of the silicon is achieved either through the addition of trace amounts of modifying elements such as strontium (impurity modification)(6–9) or by employing rapid cooling to force an increased solidification velocity (quench modification)(10–13). The impurity modification has been shown to result from the role of impurity atoms on the growth kinetics of the Si interface, where growth at twin-plane edges becomes the dominant mechanism(7) and also by suppression of eutectic nucleation(14). This produces a fine fibrous structure, highly twinned on the nanometer scale. While impurity modification is widely used in commercial modification practices due to its reproducibility, its use is also associated with a range of drawbacks, such as hot-tearing, porosity, and recycling limitations.(6; 9) Alternatively, quench modification may be used to obtain structural refinement while avoiding these drawbacks. Differing from the impurity modified alloys, which are heavily twinned and faceted on a very fine scale, quench modified structures are characterized by a low twin density and a microscopically smooth fibrous structure(12; 15). However, the quench-modification transition has been observed to occur gradually, over a wide velocity range(10), and the governing mechanisms are not completely understood, thus limiting our ability to effectively employ this morphological transition in the design of cast components.

Reported findings from a recent investigation of quench modification(10) reveal that the flake to fiber transition exhibits multiple stages, characterized by in-plane plate breakup at lower velocities and out-of-plane silicon rod formation at higher velocities. This hierarchical progression of plate breakup appears to be governed by the highly anisotropic silicon crystal structure which favors the development of low-energy high-stiffness $\{111\}$ planes and strongly restricts growth of silicon out of the plane of the plate. However, the mechanism behind the

destabilization of the silicon plate structure is not well understood, limiting our ability to predict the modification transition. In the work presented here, we employ a linear stability analysis to the flake-type silicon phase in the advancing Al-Si eutectic front in an attempt explain the plate breakup and out-of-plane rod development characteristic of the flake to fiber transition. Comparisons between the developed model and experiment are made with respect to solidification velocity, alloy composition, interfacial anisotropy, and instability length scale.

Experiments

Directional solidification experiments were performed with an Al-7 wt% Si and Al-20 wt% Si alloy in a Bridgman-type furnace over a range of growth velocities from 20 $\mu\text{m/s}$ to 1000 $\mu\text{m/s}$. Compositions were chosen to allow for comparison of a strongly hypoeutectic and strongly hypereutectic alloy. The axial temperature gradient during solidification was measured using a thermocouple embedded in a test sample and ranged from 7 K/mm at 10 $\mu\text{m/s}$ to 14 K/mm at 1000 $\mu\text{m/s}$. Samples were grown over a distance of 100 mm and then quenched to 25°C in a water-cooled bath containing a Ga-In-Sn alloy. Cross sections were obtained from each sample 10 mm behind the final quenched interface to obtain a representative eutectic structure and prepared for metallographic analysis. Specimens were polished and deep-etched by suspending in either a 2.5% HCl -1.5% HNO_3 -1% HF reagent or NaOH solution for one hour in an ultrasonic bath.

The Al-Si interface morphology and associated in-plane and out-of-plane instability development that we consider is shown schematically in Fig. 1. Morphological instabilities that develop parallel to the front edge of the advancing silicon plate can lead to the formation of in-plane protrusions, as shown in Fig. 2a, while more advanced formations can be seen in Fig. 2b-c, where protrusions have developed into in-plane rod structures. Similarly, instabilities developing on the sides of silicon plate faces are responsible for the formation of out-of-plane protrusions, a number of which are shown in Fig. 3. Characterization of silicon plate breakup was performed by measuring the average spacing of both the observed in-plane and out-of-plane silicon protrusions at each solidification velocity.

The majority of in-plane silicon plate breakup was found to occur at a similar scale for each solidification velocity, as shown in Fig. 2a. The predominance of this characteristic protrusion scale in the microstructure allowed for the measurement of an in-plane silicon protrusion spacing at all compositions and velocities studied. Measured in-plane protrusion spacings are shown in Fig. 4. Out-of-plane silicon protrusions, as shown in Fig. 3, were observed to occur in a wide variety of formations and at velocities as low as $50 \mu\text{m/s}$, but were much more common at velocities of $500 \mu\text{m/s}$ and above. Measured out-of-plane protrusion spacings are shown in Fig. 5. Both out-of-plane and in-plane protrusion spacings follow a spacing relationship near $\lambda^2 V = \text{constant}$, the microstructural scaling relationship. Alloy composition did not appear to be a significant factor on protrusion spacing for either protrusion type. However, out-of-plane silicon protrusion spacings were on average $1/3$ larger than in-plane protrusion spacings at each velocity, as shown in Fig. 6.

Instability Analysis

The regular nature of protrusion spacing observed in both alloys studied, along with the microstructural scaling, suggests the mechanism behind protrusion formation may be related to a Mullins-Sekerka type instability.(16) Mullins and Sekerka described the stability of an advancing planar solid-liquid interface by considering the behavior of an infinitesimal sinusoidal perturbation on the interface. We examine here whether the stability of interfacial perturbations plays a governing role in the morphology of silicon that develops as the plate structure breaks down in favor of a general rod structure. The central result obtained by Mullins and Sekerka gives the wavelength dependent amplification rate of a perturbation as

$$\frac{\dot{\delta}}{\delta} = \frac{V}{mG_c} \left(-\Gamma\omega^2 [b_c - (Vp/D)] - G [b_c - (Vp/D)] + mG_c [b_c - (V/D)] \right) \quad (1)$$

where

$$b_c = \frac{V}{2D} + \left[\left(\frac{V}{2D} \right)^2 + \omega^2 \right]^{\frac{1}{2}}$$

$$\dot{\delta} = \frac{\delta(t)}{dt}$$

$$\omega = 2\pi/\lambda$$

$$p = 1 - k$$

This result assumes the temperature gradients and thermal conductivities of the solid and liquid phases are equal. Descriptions of parameter symbols and material constants used in the present study can be found in Table 1. The composition gradient, G_c , must be calculated based upon the local diffusion field ahead of the interface.

Calculation of the diffusion profile in the Al-Si system is difficult due to the complex shape of the growth front, which we will now consider. The shape of an irregular eutectic interface has been modeled in a variety of forms by different researchers.(17–21) The shapes we will focus on here are those developed by Magnin and Kurz (MK) (20) and Guzik and Kopycinski (GK) (21).¹ Both interface shapes are calculated as functions of phase contact angle and phase depression/protrusion depth. Magnin and Kurz use a cubic function for the interface shape, while Guzik and Kopycinski use a biquadratic function. The two approaches are shown in Fig. 7 for two different silicon contact angles. The GK equation yields a larger depression depth in the aluminum phase and a larger protrusion in the silicon phase for a silicon contact angle of 65°, while the silicon phase shape and protrusion depth are fairly similar for an 85° silicon contact angle. The effective composition gradient was calculated in the direction normal to the interface. Thus, for a composition field $C(x, z)$, the composition gradient is

$$G_C(x, z) = \frac{\partial C(x, z)}{\partial x} \cos \phi + \frac{\partial C(x, z)}{\partial z} \sin \phi \quad (2)$$

where ϕ is the angle between the x -axis and the interface normal, as shown in Fig. 7. The

¹Note that Guzik and Kopycinski presented incorrect interface and depression depth equations for the aluminum (α) phase. The correct relations based upon their boundary conditions are graphed here in Fig. 7 and presented in the text as Eqs. 5 and 8.

value of ϕ can be determined from

$$\phi(x) = \arctan\left(\frac{-1}{\frac{dI(x)}{dx}}\right) \quad (3)$$

where $I(x)$ is the interface shape equation given by either MK or GK. Differentiating $I(x)$ from MK gives

$$\frac{dI(x)}{dx} = -\left(\frac{48\delta_\beta}{\lambda_E^3 f_\beta^3} + \frac{12 \tan \theta_\beta}{\lambda_E^2 f_\beta^2}\right) x^2 - \left(\frac{24\delta_\beta}{\lambda_E^2 f_\beta^2} + \frac{8 \tan \theta_\beta}{\lambda_E f_\beta}\right) x - \tan \theta_\beta \quad (4a)$$

$$-S_\beta \leq x < 0$$

$$\frac{dI(x)}{dx} = \left(\frac{48\delta_\alpha}{\lambda_E^3 f_\alpha^3} + \frac{12 \tan \theta_\alpha}{\lambda_E^2 f_\alpha^2}\right) x^2 - \left(\frac{24\delta_\alpha}{\lambda_E^2 f_\alpha^2} + \frac{8 \tan \theta_\alpha}{\lambda_E f_\alpha}\right) x + \tan \theta_\alpha \quad (4b)$$

$$0 \leq x < S_\alpha$$

and the corresponding equation from GK is

$$\frac{dI(x)}{dx} = \left(\frac{64\delta_\beta}{\lambda_E^4 f_\beta^4} - \frac{16 \tan \theta_\beta}{\lambda_E^3 f_\beta^3}\right) (x + S_\beta)^3 + \left(-\frac{16\delta_\beta}{\lambda_E^2 f_\beta^2} + \frac{2 \tan \theta_\beta}{\lambda_E f_\beta}\right) (x + S_\beta) \quad (5a)$$

$$-S_\beta \leq x < 0$$

$$\frac{dI(x)}{dx} = -\left(\frac{64\delta_\alpha}{\lambda_E^4 f_\alpha^4} + \frac{16 \tan \theta_\alpha}{\lambda_E^3 f_\alpha^3}\right) (x - S_\alpha)^3 + \left(\frac{16\delta_\alpha}{\lambda_E^2 f_\alpha^2} + \frac{2 \tan \theta_\alpha}{\lambda_E f_\alpha}\right) (x - S_\alpha) \quad (5b)$$

$$0 \leq x < S_\alpha$$

where λ_E represents the eutectic spacing and δ_α and δ_β are the depressions of the α and β phases, respectively. Note that δ_β will be negative for a protruding silicon phase in the MK analysis, while δ_β is defined to be positive for a protruding silicon phase by GK. For the present model, the eutectic spacing was estimated as

$$\lambda_E = 3.9 \cdot 10^{-8} V^{-0.5} \quad (6)$$

which was obtained by fitting the experimental relation, $\lambda_A = \sqrt{A/N}$, as defined in (10), to a $\lambda^2 V = \text{constant}$ relation. The volume fraction of each phase at eutectic composition was calculated assuming a density of solid aluminum of 2.56 g/cm³ at 580 °C (22) and a density of

solid silicon of 2.31 g/cm³ at 580°C. The depression depths in the MK solution are given by

$$\delta_i = \frac{2f_i^2}{48\Gamma_i + f_i^2\lambda_E^2 G} \left[\left(\frac{|m_i| C_0}{D} \left(\Pi_i - \frac{P}{f_i} \right) V - \frac{f_i \tan \theta_i}{24} G \right) \lambda_E^3 - \frac{2\Gamma_i}{f_i} (2 \tan \theta_i + \sin \theta_i) \lambda_E \right] \quad (7)$$

$$i = \alpha, \beta$$

and the depression depths in the GK solution are

$$\delta_\beta = \frac{15f_\beta^2}{7G\lambda_E^2 f_\beta^2 + 240\Gamma_\beta} \left[\left(\frac{m_\beta C_0 V}{D} \left(\Pi_\beta - \frac{P}{f_\beta} \right) + \frac{Gf_\beta \tan \theta_\beta}{30} \right) \lambda_E^3 + \frac{2\Gamma_\beta \lambda_E}{f_\beta} (\tan \theta_\beta + \sin \theta_\beta) \right] \quad (8a)$$

$$\delta_\alpha = \frac{15f_\alpha^2}{7G\lambda_E^2 f_\alpha^2 + 240\Gamma_\alpha} \left[\left(\frac{|m_\alpha| C_0 V}{D} \left(\Pi_\alpha - \frac{P}{f_\alpha} \right) - \frac{Gf_\alpha \tan \theta_\alpha}{30} \right) \lambda_E^3 - \frac{2\Gamma_\alpha \lambda_E}{f_\alpha} (\tan \theta_\alpha + \sin \theta_\alpha) \right] \quad (8b)$$

with

$$\Pi_i = \sum_{n=1}^{\infty} \frac{1}{(n\pi)^2} \sin(n\pi f_i), \quad i = \alpha, \beta$$

$$P = \sum_{n=1}^{\infty} \frac{1}{(n\pi)^3} \sin^2(n\pi f_\alpha)$$

Ideally, the composition field, $C(x, z)$, should be derived based on the complex Al-Si interface morphology, rather than the commonly used planar interface morphology solution presented by Jackson and Hunt.(17) However, Magnin *et al.*(23) noted that a reasonable approximation of the interface composition in an irregular eutectic alloy is given by the planar interface diffusion solution. This is due to the scale of variation in the interface position being significantly smaller than the diffusion distance at common solidification rates. Therefore, the composition field ahead of the Al-Si interface may be reasonably estimated using the planar diffusion solution, and the composition gradient is obtained from Eq. 2, which effectively superimposes the interface shape over the composition field, as shown in Fig. 8. According to the planar diffusion solution, the values of the composition gradients in the x and z directions

taken at $z = 0$ are equal to

$$\frac{\partial C}{\partial x}(x, z = 0) = \frac{-2VC_0}{\pi D} \sum_{n=1}^{\infty} \frac{1}{n} \sin(n\pi f_\alpha) \sin\left(\frac{2n\pi}{\lambda_E}x - n\pi f_\alpha\right) \quad (9a)$$

$$\frac{\partial C}{\partial z}(x, z = 0) = \frac{-VB_0}{D} + \frac{-2VC_0}{\pi D} \sum_{n=1}^{\infty} \frac{1}{n} \sin(n\pi f_\alpha) \cos\left(\frac{2n\pi}{\lambda_E}x - n\pi f_\alpha\right) \quad (9b)$$

with

$$B_0 = C_0^\alpha f_\alpha - C_0^\beta f_\beta \quad (10)$$

where C_0^α and C_0^β are phase diagram constants as defined in (17).

The wavelength amplification rate (Eq. 1) was used to determine two important instability spacing parameters: the marginal and maximum instability. Marginal instability refers to the smallest perturbation wavelength where the interface first becomes unstable, while maximum instability refers to the perturbation wavelength of maximum amplification rate. One method to obtain these instability parameters is to substitute the composition gradient (Eq. 2) into Eq. 1 and solve for the marginal or maximum instability spacing as a function of x along the interface ahead of the silicon phase. This is shown in Fig. 9 for both the MK and GK equations. Alternatively, the average composition gradient in the liquid at the interface can be calculated with

$$\bar{G}_C = \frac{2}{\lambda_E} \int_{-S_\beta}^0 G_C(x, z = 0) dx \quad (11)$$

Using this equation, a single average instability spacing can be obtained for a specified solidification velocity, which is also shown in Fig. 9. The marginal instability spacing at each point ahead of the silicon phase varies less than 25% from its average value.

It is also useful to consider the effect of silicon interfacial anisotropy in the model. The Gibbs-Thomson coefficient, Γ , can be expressed in terms of interfacial stiffness, $\Gamma = \frac{\tilde{\gamma}(\theta)}{\Delta S_f}$. The interfacial stiffness, $\tilde{\gamma}(\theta)$, is often calculated from a simple model which employs a generic parameter, ϵ_4 , to reflect four-fold interfacial anisotropy.(24; 25) This model states

$$\tilde{\gamma}(\theta) = \gamma_0 (1 - 15\epsilon_4 \cos 4\theta) \quad (12)$$

It can be seen that any value of ϵ_4 greater than 0.067 predicts a negative interfacial stiffness, which is not physically possible. Therefore, in the case of $\epsilon_4 > 0.067$, the equilibrium shape of

the system will be faceted. A similar model for interfacial anisotropy was employed recently by other researchers investigating planar morphological stability.(26; 27) The effect of anisotropy varies based upon the direction of growth with respect to crystallographic orientation (θ). For the case of instability formation out of the plane of the silicon plate, it was assumed that growth occurs in the direction of maximum stiffness, or $\theta = \frac{\pi}{4}$. In this case, the anisotropy-dependent Gibbs-Thomson coefficient can be specified as

$$\Gamma = \Gamma_0 (1 + 15\epsilon_4) \quad (13)$$

Therefore, relatively small changes in the anisotropy parameter can significantly alter the effective Gibbs-Thomson coefficient. A more exact model for the contributions of anisotropy could be obtained using a three dimensional anisotropy model(28). However, the current model is sufficient to demonstrate the potential effects of anisotropy, and was therefore chosen for simplicity.

The anisotropy of the solid Si/liquid Al-Si interface has been previously investigated by Sens *et. al.* (29) using a binary Al-95 wt% Si alloy. After heat treating for 500 hours at 600 °C and quenching, they observed {111} facet formation on the interface between the solid silicon matrix and liquid Al-Si intragranular inclusions. Furthermore, by measuring the distance between the center of inclusions and the inclusion peripheries they calculated the ratio of maximum anisotropy ($\sigma_{(001)}/\sigma_{(111)}$) to be 1.11 ± 0.02 . The value of ϵ_4 on a given plane can be calculated with $\epsilon_4 = \frac{\Delta-1}{\Delta+1}$ with $\Delta = \frac{r_{max}}{r_{min}}$ as specified in (30). Using $\Delta = 1.11$ gives $\epsilon_4^{max} = 0.05$. An attempt was made to verify this estimate of solid Si/liquid Al-Si interfacial anisotropy for the present study by holding a finely quenched Al-20 wt% Si alloy at 592 °C for 125 hours. This produced rounded octahedral silicon shapes roughly 20 μm in extent with faceting behavior consistent with {111} planar symmetry, as shown in Fig. 10. The $\langle 110 \rangle$ and $\langle 100 \rangle$ directions were measured from the base (100) plane on all three particles shown and the average value of $\epsilon_4^{100} = 0.086$ was obtained. Additionally, for the top particle in Fig. 10, the {111} distances were estimated, and a value of $\epsilon_4^{max} = 0.17$ was obtained. These measured values represent lower bounds for the solid Si/liquid Al-Si interfacial anisotropy, since it is not certain that the particles in Fig. 10 had reached equilibrium.

Model Results

Calculated instability spacings are shown in Figs. 11–13. Instability spacings are within the same order of magnitude as experimentally measured protrusion spacings. As shown in Fig. 11, the wavelength of marginal stability predicted by the model assuming no silicon anisotropy is $\approx 25\%$ of experimentally observed protrusion spacings while the wavelength of maximum instability is $\approx 40\%$ of experimentally observed protrusion spacings. Calculated spacings scale similarly with the microstructural scaling. It is also evident from these figures that increasing anisotropy can significantly increase the instability wavelength. Additionally, as evident in Fig. 12, the 85° silicon wetting angle results in $\approx 10\%$ higher spacings than the 65° wetting angle. Furthermore, there is little difference in the spacings obtained between the MK and GK equations. Alloy composition is shown to not have a significant effect on instability length scale in Fig. 13.

Discussion

From the present results, it can be asserted that the development of both in-plane and out-of-plane silicon protrusions, as observed and characterized herein, is key to the progress of the flake to fiber transition. The increase in out-of-plane protrusion formation noted at $500 \mu\text{m/s}$ in both the hypoeutectic and hypereutectic alloys examined in this study parallels the onset of out-of-plane silicon growth, denoted as the second stage of the flake to fiber transition, in the eutectic alloy described by Hosch *et al.*(10) While a limited number of intricate out-of-plane silicon protrusion formations were noted at velocities fully dominated by plate growth (Fig. 3b, d, and e), this may be explained by the presence of wide local variations within the diffusion field that are intrinsic to the irregular Al-Si structure. Further, the similarity of in-plane and out-of-plane silicon protrusions, along with the appearance of in-plane protrusions at lower velocities than out-of-plane protrusions, suggests in-plane plate breakup may be governed by the same instability mechanism that controls out-of-plane rod formation though operating on a different crystallographic plane. It is possible, therefore, that the different stages of the Al-Si flake to fiber transition are manifestations of a similar instability mechanism operating

on different crystallographic orientations throughout the velocity range of the transition.

As evident in Figs. 11–13, the perturbed interface model accurately predicts the length scale behavior of both in-plane plate breakup and out-of-plane plate breakup with respect to the effects of solidification velocity, alloy composition, and expected material anisotropy. Both experimental protrusion spacing and calculated perturbation wavelength approximate a $\lambda^2 V = \text{constant}$ relationship. Additionally, the lack of significant variation in protrusion spacing between the two compositions studied, as shown in Figs. 4–5, is in agreement with the small compositional effect on instability wavelength predicted by the perturbed interface model over the present composition range (Fig. 13). The slightly higher observed in-plane protrusion spacings for the hypoeutectic alloy shown in Fig. 4 may be a reflection of the effect of composition. Finally, for perturbations developing in the maximum interfacial stiffness direction, the model predicts increasing perturbation wavelengths with increasing anisotropy which is in agreement with the experimental observation of larger out-of-plane protrusion spacings in comparison to observed in-plane protrusion spacings. This is evident through consideration of the silicon flake crystallography, from which a larger anisotropy (interfacial stiffness) can be expected in directions normal to the plane of the plate as compared to in-plane directions normal to plate edges. Correspondingly, perturbations perpendicular to the plane of the plate that lead to out-of-plane protrusion formation will require a larger wavelength to be stable than perturbations parallel to the plane of the plate that lead to in-plane plate breakup.

By predicting a maximum perturbation amplification wavelength $\approx 1/3$ of experimentally observed protrusion spacings while assuming no anisotropy, the perturbed interface model reasonably predicts the length scale of protrusion formation throughout the velocity range studied considering the model’s inherent limitations. The perturbed interface model assumes steady state solidification conditions, an assumption which, by definition, breaks down once a stable perturbation is formed which destabilizes the advancing planar interface. Therefore, the final macroscopic protrusion spacing in the silicon structure can only be inferred from the contributory microscopic perturbation wavelength, and in practice, significant coarsening effects are operative during this phase of growth. As a result, the macroscopic protrusion length

scale that develops from microscopic perturbations is expected to be several times larger than the wavelength of the destabilizing perturbations.(31–34) Furthermore, the value of the silicon anisotropy coefficient for the solid silicon/liquid Al-Si interface has received limited study, potentially lowering the accuracy of predicted instability wavelengths when considering the significant influence of the anisotropy parameter (Fig. 11). The faceted nature of the silicon structure indicates a strong effect of anisotropy, however, which has the effect of pushing the predicted anisotropy wavelength closer to the observed protrusion length scale.

While the perturbed interface model adequately predicts the length scale behavior of in-plane and out-of-plane silicon protrusions that lead to the flake to fiber transition, it does not predict the onset of protrusion formation itself. Stable perturbation wavelengths were predicted for all experimental velocities, compositions, and anisotropy levels investigated herein, while it is apparent from experimental observations that the frequency of occurrence of silicon protrusions varies significantly with solidification velocity. In fact, it can be seen in Fig. 14 that the present perturbed interface model predicts a wavelength stability regime that extends to velocities several orders of magnitude lower than those encountered in the present experiment. Anisotropy is seen to have a mild effect on the size of the stability regime, with higher anisotropy decreasing the extent of the unstable region when the maximum stiffness direction is assumed for protrusion growth. This effect could explain the development of in-plane protrusions at lower velocities than out-of-plane protrusions assuming the in-plane directions are associated with lower anisotropy, however the shift of the stability regime occurs below $1 \mu\text{m/s}$, and thus does not associate favorably with the velocity range of the flake to fiber transition. Further study is needed to determine modeling constraints that predict the frequency of instability formation in addition to the length scale, which would provide a more complete description of the silicon microstructural transformation.

Conclusions

1. Out-of-plane silicon growth is vital to the formation of a modified fibrous structure at high solidification rates.

2. The in-plane and out-of-plane silicon plate breakup associated with the flake to fiber transition is initiated by a Mullins-Sekerka type instability.
3. The length scale of the instability mechanism responsible for silicon plate breakup and out-of-plane rod formation is adequately predicted by the perturbed interface stability model described herein.
4. The characteristic in-plane silicon plate breakup length scale is smaller than the corresponding length scale of out-of-plane protrusions due to the expected larger interfacial stiffness experienced by perturbations perpendicular to the plane of the plate, a phenomenon that is accurately predicted by the perturbed interface stability model.

Acknowledgements

This work was made possible by support from the National Science Foundation (NSF), Division of Materials Research, under Award No. 0237566.

References

- [1] Nabil Fat-Halla. Structure, mechanical properties and fracture of aluminum alloy a-356 modified with aluminum-strontium (al-5sr) master alloy. *Journal of materials science*, 22(3):1013, 1987.
- [2] N. Fat-Halla. Structural modification of aluminum-silicon eutectic alloy by strontium and its effect on tensile and fracture characteristics. *Journal of materials science*, 24(7):2488, 1989.
- [3] Mahmoud Fouad Hafiz, Nabil K. Fat-Halla, and Sherif B. Moshref. The effect of casting parameters on the modification process of a commercial hypoeutectic aluminum-silicon alloy. *Zeitschrift fr Metallkunde*, 81(1):70, 1990.
- [4] Mahmoud Hafiz and Toshiro Kobayashi. Tensile properties influencing variables in eutectic al-si casting alloys. *Scripta metallurgica et materialia*, 31(6):701, 1994.

- [5] M. Zuo, X. F. Liu, and Q. Q. Sun. Effects of processing parameters on the refinement of primary si in a390 alloys with a new al-si-p master alloy. *Journal of Materials Science*, 44(8):1952–1958, 2009.
- [6] M. Garat, G. Laslaz, S. Jacob, P. Meyer, P. H. Guerin, and R. Adam. State-of-art use of sb-, na- and sr-modified al-si casting alloys. *Transactions of the Amercian Foundrymen's Society*, 100:821–832, 1992.
- [7] Shu Zu Lu and A. Hellawell. The mechanism of silicon modification in aluminum-silicon alloys: impurity induced twinning. *Metallurgical transactions. A, Physical metallurgy and materials science*, 18A(10):1721, 1987.
- [8] D. C. Jenkinson and L. M. Hogan. Modification of aluminum-silicon alloys with strontium. *Journal of Crystal Growth*, 28(2):187, 1975.
- [9] A. K. Dahle, K. Nogita, S. D. McDonald, C. Dinnis, and L. Lu. Eutectic modification and microstructure development in alsil alloys. *Materials Science and Engineering A*, 413-414:243–248, 2005.
- [10] T. Hosch, L. G. England, and R. E. Napolitano. Analysis of the high growth-rate transition in al-si eutectic solidification. *Journal of Materials Science*, 44(18):4892–4899, 2009.
- [11] M. G. Day and A. Hellawell. Microstructure and crystallography of aluminum-silicon eutectic alloys. *Proceedings - Royal Society. Mathematical, physical and engineering sciences*, 305(1483):473, 1968.
- [12] Shu Zu Lu and Angus Hellawell. Growth mechanisms of silicon in aluminum-silicon alloys. *Journal of Crystal Growth*, 73(2):316, 1985.
- [13] R. E. Napolitano and L. G. England. High-rate growth-mode transitions in al-si eutectics. *Solidification of Aluminum Alloys*, pages 445–451, 2004.
- [14] Y. H. Cho, H. C. Lee, K. H. Oh, and A. K. Dahle. Effect of strontium and phosphorus on eutectic al-si nucleation and formation of -al₅fesi in hypoeutectic al-si foundry alloys.

- Metallurgical and Materials Transactions; A; Physical Metallurgy and Materials Science*, 39(10):2435, 2008.
- [15] L. M. Hogan and M. Shamsuzzoha. Crystallography of the flake-fiber transition in the aluminum-silicon eutectic. *Materials forum*, 10(4):270, 1987.
 - [16] W. W. Mullins and R. F. Sekerka. Stability of a planar interface during solidification of a dilute binary alloy. *Journal of Applied Physics*, 35:445–451, 1964.
 - [17] K. A. Jackson and J. D. Hunt. Lamellar and rod eutectic growth. *Transactions of the Metallurgical Society of AIME*, 236(8):1129, 1966.
 - [18] T. Sato and Y. Sayama. Completely and partially co-operative growth of eutectics. *Journal of Crystal Growth*, 22:259–271, 1974.
 - [19] D. J. Fisher and W. Kurz. A theory of branching limited growth of irregular eutectics. *Acta Metallurgica*, 28(6):777, 1980.
 - [20] P. Magnin and W. Kurz. An analytical model of irregular eutectic growth and its application to fe - c. *Acta Metallurgica*, 35(5):1119–1128, 1987.
 - [21] E. Guzik and D. Kopycinski. Modeling structure parameters of irregular eutectic growth: modification of magnin-kurz theory. *Metallurgical and Materials Transactions; A; Physical Metallurgy and Materials Science*, 37A(10):3057, 2006.
 - [22] T. Magnusson and L. Arnberg. Density and solidification shrinkage of hypoeutectic aluminum-silicon alloys. *Metallurgical and Materials Transactions a-Physical Metallurgy and Materials Science*, 32(10):2605–2613, 2001.
 - [23] P. Magnin, J. T. Mason, and R. Trivedi. Growth of irregular eutectics and the aluminum-silicon system. *Acta metallurgica et materialia*, 39(4):469, 1991.
 - [24] R. E. Napolitano, S. Liu, and R. Trivedi. Experimental measurement of anisotropy in crystal-melt interfacial energy. *Interface Science*, 10(2-3):217–232, 2002.

- [25] J. R. Morris and R. E. Napolitano. Developments in determining the anisotropy of solid-liquid interfacial free energy. *Jom*, 56(4):40–44, 2004.
- [26] Z. J. Wang, J. C. Wang, and G. C. Yang. Effects of surface tension anisotropy on interfacial instability in directional solidification. *Crystal Research And Technology*, 44(1):43–53, 2009.
- [27] M. W. Chen, M. Lan, L. Yuan, Y. Y. Wang, Z. D. Wang, and J. J. Xu. The effect of anisotropic surface tension on the morphological stability of planar interface during directional solidification. *Chinese Physics B*, 18(4):1691–1699, 2009.
- [28] R. E. Napolitano and S. Liu. Three-dimensional crystal-melt wulff-shape and interfacial stiffness in the al-sn binary system. *Physical Review B*, 70(21):11, 2004.
- [29] H. Sens, N. Eustathopoulos, and D. Camel. Anisotropy of solid si-liquid (al,si) interfacial-tension in the binary and sr-doped al-si eutectic alloy. *Journal of Crystal Growth*, 98(4):751–758, 1989.
- [30] S. Liu, R. E. Napolitano, and R. Trivedi. Measurement of anisotropy of crystal-melt interfacial energy for a binary al-cu alloy. *Acta Materialia*, 49(20):4271–4276, 2001.
- [31] R. Trivedi and K. Somboonsuk. Pattern formation during the directional solidification of binary systems. *Acta Metall.*, 33(6):1061–1068, 1985.
- [32] J. A. Warren and J. S. Langer. Prediction of dendritic spacing in a directional solidification experiment. *Phys. Rev. E.*, 47(4):2702–2712, 1993.
- [33] W. Losert, B. Q. Shi, and H. Z. Cummins. Evolution of dendritic patterns during alloy solidification: From the initial instability to the steady state. *Proceedings of the National Academy of Sciences of the United States of America*, 95:439–442, 1998.
- [34] W. Losert, B. Q. Shi, and H. Z. Cummins. Evolution of dendritic patterns during alloy solidification: Onset of the initial instability. *Proceedings of the National Academy of Sciences of the United States of America*, 95:431–438, 1998.

Figure 1 Al-Si interface schematic demonstrating both the development of in-plane (right) and out-of-plane (left) silicon protrusions. Both types of protrusion formation may develop separately, together, or not at all. The type of instability that forms on any given plate will be governed by the local flake growth conditions, which are widely variable and difficult to predict due to the inherent irregularity of the Al-Si structure.

- [35] J. L. Murray and A. J. McAllister. The al-si (aluminum-silicon) system. *Bulletin of Alloy Phase Diagrams*, 5(1):74, 1984.
- [36] P. Magnin and R. Trivedi. Eutectic growth: A modification of the jackson and hunt theory. *Acta Metall Mater*, 39(4):453–467, 1991.

Table 1 Definition of symbols. *Composition parameters converted from wt% silicon to mol/m³ for calculation of gradient.

Symbol	Description	Value	Units	Ref/Eqn
b_c	instability constant	–	m ⁻¹	Eq. 1
B_0	diffusion equation coefficient	–	wt%*	Eq. 10
C	liquid composition	–	wt%*	Eq. 9
C_0	phase diagram constant	98.38	wt%*	(35)
C_0^α	phase diagram constant	11	wt%*	(35)
C_0^β	phase diagram constant	87.38	wt%*	(36)
C_E	eutectic composition	12.6	wt%*	(35)
D	liquid interdiffusion coefficient	$4.3 \cdot 10^{-9}$	m ² /s	(36)
f_α	volume fraction of phase α	0.8622	–	–
f_β	volume fraction of phase β	0.1378	–	–
G	temperature gradient	7500	K/m	–
G_c	composition gradient	–	wt%/m*	Eq. 2
$\bar{G}_C$	average composition gradient	–	wt%/m*	Eq. 11
I	Al-Si interface position	–	m	Eq. 4
k	partition coefficient	–	–	Eq. 1
m, m_β	slope of the liquidus	15.7	K/wt%*	(36)
m_α	slope of the aluminum liquidus	-7.5	K/wt%*	(36)
p	function of partition coefficient	–	–	Eq. 1
P	function of volume fraction	–	–	Eq. 7
S_f	entropy of fusion	–	J/m ³ K	–
S_i	half width of phase i	–	m	–
α	denotes aluminum phase	–	–	–
β	denotes silicon phase	–	–	–
δ	perturbation amplitude	–	m	Eq. 1
$\dot{\delta}$	rate of change of δ	–	m/s	Eq. 1
δ_i	depth of depression in phase i	–	m	Eq. 7
ϵ_4	interfacial anisotropy parameter	0–0.1	–	–
γ_0	silicon isotropic surface energy	$1.7 \cdot 10^{-7}$	J/m ²	(36)
$\tilde{\gamma}$	silicon interfacial stiffness	–	J/m ²	Eq. 12
Γ	Gibbs-Thomson coefficient	–	m K	Eq. 13
Γ_0, Γ_β	isotropic Gibbs-Thomson coefficient	$1.7 \cdot 10^{-7}$	m K	(36)
Γ_α	Gibbs-Thomson coefficient	$1.96 \cdot 10^{-7}$	m K	(36)
λ	perturbation wavelength	–	m	–
λ_E	eutectic spacing	–	m	Eq. 6
ω	perturbation frequency	–	m ⁻¹	Eq. 1
ϕ	angle from x -axis to interface normal	–	degrees	Eq. 3
Π	function of volume fraction	–	–	Eq. 7
θ	crystal growth direction	45	degrees	–
θ_α	wetting angle of phase α	30	degrees	(36)
θ_β	wetting angle of phase β	65, 85	degrees	(21)

Figure 2 Images showing examples of in-plane silicon plate instability. This instability leads to the development of in-plane protrusions which are shown in (a) and indicated by white arrows, and eventually results in the development of in-plane rod structures, as shown in (b) and (c). Images are from an Al-20 wt% Si alloy (a, b) and an Al-7 wt% Si alloy (c) and were solidified at $200\ \mu\text{m/s}$ (a) and $50\ \mu\text{m/s}$ (b, c). All samples prepared by deep etching in a 2.5% HCl -1.5% HNO_3 -1% HF reagent.

Figure 3 Observed out of plane instability formations in an Al-7 wt% Si alloy (c) and Al-20 wt% Si alloy (a, b, d-f): a) Columnar instability, b) Columnar instability with additional row instability, c) Grid instability, d) Separate grid and columnar instabilities on same plate, e) Advanced grid instability displaying inter-connecting bands, and f) instability formation at multiple levels of advancement. Solidification velocities include $50 \mu\text{m/s}$ (b), $100 \mu\text{m/s}$ (d, e), $200 \mu\text{m/s}$ (a), and $1000 \mu\text{m/s}$ (c, f). Samples were prepared by deep-etching in a $2.5\%\text{HCl}-1.5\%\text{HNO}_3-1\%\text{HF}$ reagent (a-e) or NaOH solution (f).

Figure 4 Comparison of in-plane protrusion spacings measured on a hypoeutectic (7 wt%) and hypereutectic (20 wt%) Al-Si alloy. Error bars represent the 95% confidence interval of the mean.

Figure 5 Comparison of out-of-plane protrusion spacings measured on a hypoeutectic (7 wt%) and hypereutectic (20 wt%) Al-Si alloy. Error bars represent the 95% confidence interval of the mean.

Figure 6 Comparison of protrusion spacings in an Al-20 wt% Si alloy for in-plane and out-of-plane protrusions. Error bars represent the 95% confidence interval of the mean.

Figure 7 Al-Si interface shapes as specified by Magnin and Kurz (MK) (20) and Guzik and Kopycinski (GK) (21) for two different silicon contact angles. Calculated at $1000 \mu\text{m/s}$.

Figure 8 Comparison of a planar interface composition contour map and the Al-Si interface shape.

Figure 9 The marginal instability spacing calculated as a function of x in front of the silicon phase for both the Magnin-Kurz and Guzik-Kopycinski interface shapes. Calculations are for a solidification velocity of $1000 \mu\text{m/s}$ and a silicon phase contact angle of 85° . Also shown is the marginal stability spacing calculated using the average composition gradient ahead of the silicon phase.

Figure 10 Near-equilibrium silicon shapes of an Al-20 wt% Si alloy held at 592 °C for 125 hours. Silicon particles approximate an octahedral shape, with triangular facets consistent with faceting on {111} silicon planes. Particles were extracted after rapid quenching and deep-etching in a 2.5% HCl-1.5% HNO₃-1% HF reagent.

Figure 11 Comparison of experimental silicon protrusion spacings and calculated minimum and maximum instability spacings with multiple values of anisotropy. Calculated values are based on the Magnin-Kurz(20) interface shape at a silicon phase contact angle of 85° .

Figure 12 Effect of anisotropy and wetting angle on the calculated minimum and maximum instability wavelengths at a $1000 \mu\text{m/s}$ solidification velocity. Results were obtained using the Magnin-Kurz (20) interface shape.

Figure 13 Effect of alloy composition on calculated marginal and maximum amplification rate instability spacings.

Figure 14 Planar interface stability range as a function of perturbation wavelength and solidification velocity. Inside the unstable region perturbations will grow with time and destabilize the planar interface. Higher anisotropy serves to decrease the extent of the instability range to a minor degree.

METALLURGICAL ANALYSIS OF FLOW-LINE INDICATIONS IN MG-BASED ALLOY CASTINGS

A paper to be submitted to Journal of Cast Metals Research

T. Hosch and R. E. Napolitano

Abstract

Additions of heavy elements such as zinc, zirconium, and rare earth elements to magnesium casting alloys can significantly improve high temperature performance. However, these elements can lead to macrosegregational effects in castings, which are detectable by radiographic scans. The effects of these flow-line indications on alloy mechanical properties are examined here in three magnesium-based casting alloys. Preliminary results found instances of micromechanical effects as a result of the flow-line indications, though no evidence was found to suggest that the flow-lines measurably impact bulk material properties.

Introduction

Many magnesium-based castings for aerospace applications have requirements for one hundred percent radiographic inspection. A frequent observation resulting from such inspection is an indication commonly known as a “flow line”, which is believed to be due to chemical inhomogeneities arising from solute partitioning and flow-induced segregation in the mushy zone. Since the radiographic contrast of a flow-line indication depends on X-ray absorption, the relationship between radiographic contrast and the magnitude of chemical segregation depends strongly on the atomic number of the alloying elements present. Indeed, lighter elements such

as Al and Zn do not promote readily detectable flow lines. However, heavier elements, such as Ce, Nd, Y, and other rare earths (ORE) give rise to strong flow-line contrast during radiographic inspection. Despite the differences in the observable radiographic contrast associated with similar levels of segregation, acceptability limits have not taken into account the specific partitioning, density, and X-ray absorption characteristics of the segregating elements. This calls into question the use of uniform acceptability limits with regard to the heavy elements common in the newer generation Mg-based alloys. To examine this issue, the investigation reported here is a preliminary study into the chemical and microstructural inhomogeneities associated with flow line indications and their influence on the mechanical properties of the cast material.

Scope

The investigation reported here includes the analysis of segregational flow lines, detected by radiographic analysis, in three different cast magnesium alloys, WE43B, EV31A, and ZE41. The purpose of this investigation is to characterize the chemical nature of the flow lines present in each alloy and to examine their effect on the microstructure and mechanical properties of the alloy. Microstructural analysis was performed using optical and scanning electron microscopy (SEM). Microchemical analysis and measurement of chemical segregation was performed using energy dispersive spectroscopy (EDS). Mechanical properties were examined with local microhardness measurements, uniaxial tension testing, and three-point-bend fracture analysis.

Test Specimens

Test samples of three alloy types were supplied by Wellman Dynamics¹, with nominal compositions as listed in Table 1. An undisclosed post-casting heat treatment was performed for the WE43B alloy. The EV31 alloy was heat treated to a T6 condition, involving a solution treatment at 520°C for 8 hours followed by a still air quench, a two-hour stress relief at 510°C (after welding), a polymer quench, and a final aging treatment at 200°C for 16 hours. The

¹Fansteel Corp., Creston, Iowa

ZE41 alloy was heat treated to a T5 condition, involving aging at 330°C for 2 hours. Each test sample was cut from a thin (≈ 6 mm) section of a large (approximately 225 kg) casting, shown by radiographic analysis to contain segregational flow lines. Radiographic flow line indications for each test sample are shown in Figs. 1–2. For each alloy, a total of four specimens were provided, including two flat “dog-bone” specimens, for uniaxial tension testing, and two additional specimens for metallographic analysis. Specimen identification numbers (assigned by Wellman Dynamics) are listed in Table 1.

Table 1 Nominal compositions of test specimens, provided by Wellman Dynamics.

Alloy	Mg	Y	Nd	Zr	Zn	Gd	ORE*	Tension	Metall.
WE43B	Bal.	4.07	2.22	0.82	0.02	-	1.02	V17B/V17C	V19/V36
EV31A	Bal.	-	2.88	0.68	0.36	1.48	0.17	C/I	A/D
ZE41	Bal.	-	-	0.73	4.33	-	1.19	1/2	3A/3B

* ORE denotes total of other rare earth elements.

Microstructural Analysis

Metallographic analysis, scanning electron microscopy, and energy dispersive spectroscopy were used to characterize the microstructure and chemical segregation patterns in the metallurgical test specimens in the vicinity of the observed flow-line indications. Representative microstructures for each test alloy, as revealed by backscattered electron contrast, are shown in Fig. 3. The microstructures of the three alloys each exhibit distinct features which persist from the solidification process. For the WE43B alloy, the most distinctive features in the microstructure are the light-colored areas indicating solute segregation on a scale consistent with the dendritic morphology within the mushy zone. The EV31A alloy exhibits a well-defined grain structure, evidence of the long high-temperature solution treatment that was performed, permitting evolution of a grain structure from the dendritic structure of the cast alloy. Within the well-defined grains, however, evidence of dendritic segregation during solidification is clearly visible in the form of the spatial distribution of precipitates. For the

Figure 1 X-ray radiographs of the six tensile specimens showing flowline indications.

Figure 2 X-ray radiographs of the six coupon specimens showing flowline indications.

ZE41 alloy, which was subjected only to the intermediate-temperature two-hour aging treatment, the grain boundary structure is not well established, and the remnants of the dendrite arm structure remain distinctive. In the following sections, the microstructure of each alloy is examined in more detail, with particular attention given to microsegregation profiles, phase distribution, and other nonuniform microstructural features that may be associated with the presence of the observed *flow-lines*.

Alloy WE43B

The most notable features in the WE43B structure in Fig. 3(a) are the small light areas which are evenly dispersed throughout the structure (observed in both V19 and V36 specimens). The EDS analysis results, summarized in Table 2, reveal significant Zr enrichment in these lighter regions, with Zr content increasing to roughly seven times the nominal level in the surrounding darker areas. The inhomogeneity of the Zr concentration is revealed by the linescan shown in Fig. 4, where the zirconium content can be seen to increase mildly but significantly within the lighter region. Indeed, the scale and pattern of the zirconium-enriched

Figure 3 Representative microstructures of the test alloys (backscattered electron images): (a) WE43B (left), (b) EV31A (middle), and (c) ZE41 (right).

regions throughout the sample suggests that the regions arise from dendritic segregation during solidification.

Considering now the relationship between the structure, chemistry, and the observed flow-line, we note that the flow-lines in samples V19 and V36 are readily visible upon inspection of the optical micrographs, shown in Fig. 5. Moreover, the flow-lines are strongly correlated with a visible change in the spatial distribution of the Zr-enriched regions. However, EDS composition profiles, measured over a distance of several millimeters across the flow-line, do not reveal any overall macroscopic variation in composition associated with this change in segregation pattern, as shown in Fig. 5(a). Indeed, no significant compositional variation across the flow-line region was detected.

Table 2 EDS analysis of alloy WE43B.

Elem.	<i>Dark Areas</i>		<i>Light Areas</i>	
	V19	V36	V19	V36
Mg	93.2	93.2	92.1	91.9
Y	4.0	3.8	3.5	3.3
Zr	0.3	0.4	2.2	2.8
Nd	2.5	2.5	2.2	2.0

Examining the influence of the structural variation further, microhardness profiles were

Figure 4 EDS linescan showing the variation in Zr content in the WE43B alloy (V19).

measured across the sample surface to determine if the flow lines contributed to any deleterious local mechanical effects. The results for sample V36, shown in Fig. 5(a) do not indicate any significant variation in hardness across the flow-line. However, microhardness results for sample V19, in Fig. 5(b) show a hardness difference of about 7 VHN from one side of the flow line to the other. Here, the properties of the flow-line region itself are not affected, but the flow-line serves as a boundary between regions with distinctly different properties.

Alloy EV31A

The EV31A alloy exhibits a grain structure that is well defined. Representative backscattered electron images of the EV31A microstructure are shown in Fig. 3(b), indicating that there are three distinct phases in the microstructure that are well resolved by the atomic-weight sensitivity of the BSE contrast. First, there is the (dark) primary phase which grew dendritically from the liquid. Second, there is a blocky grain boundary allotriomorph morphology with rather large particles, likely formed by coarsening and coalescence of the interdendritic

Figure 5 Optical micrographs of alloy WE43B with corresponding micro-hardness and composition profiles.

phases during the high-temperature solution treatment. Finally, there is a distinct pattern of intragranular precipitation at the center of the grains with precipitate free zones at the grain peripheries. A cruciform pattern of these precipitates can be observed on the left side of Fig. 3(b), indicating clearly the cored segregation pattern that developed during primary dendritic solidification. Phase compositions, measured by EDS analysis, are listed in Table 3. The dark matrix phase is the Mg(hcp) primary solution phase, containing the indicated amounts of the listed solute elements.

With respect to the primary solidification, none of the elements are particularly strong partitioners, except for Zr which appears to have been much more concentrated in the Mg(hcp) solution phase, prior to precipitation of the Zr-rich plates, than in the pre-existing liquid. This segregation of Zr in the primary Mg(hcp) phase gives rise to the pattern of fine Zr-rich plate precipitates observed in Fig. 3(b) and can be understood by considering the peritectic invariant exhibited by the Mg-Zr binary system, where the solubility shifts from 0.6 wt.% in the liquid to approximately 3.8 wt.% in the Mg(hcp) phase upon cooling through the peritectic at 654°C (1209°F). During the low temperature aging treatment, however, the Zr solubility is approximately 0.2 wt.%, consistent with the measured concentration in the matrix (Mg) phase. It should also be noted that the resolution of the EDS microchemical probe is approximately 0.001 mm. Therefore, the composition values listed for the ultra-fine plate precipitates can be considered to reflect the average composition in that area. The overall chemical segregation and relative phase compositions are illustrated further by the chemical dot maps shown in Fig. 6, where Nd-rich grain boundary particles and the increased Zr associated with the fine intragranular precipitates are evident.

Table 3 EDS analysis of alloy EV31A (sample A). (Values in wt. percent.)

Element	Overall	Primary	G.B. Ppt. 1	G.B. Ppt. 2	Plate Ppt.
Mg	Bal.	95.0	62.6	65.0	87.2
Zn	0.36	0.3	1.7	1.5	2.0
Zr	0.68	0.2	0.2	0.2	5.1
Nd	2.88	2.8	33.4	31.6	3.4
Gd	1.48	1.7	2.1	1.7	2.2

Figure 6 Backscattered electron image (left) of EV31A (sample A) and corresponding dot maps, showing Nd-rich grain boundary particles (center) and Zr-rich intragranular plates (right).

A detailed view of an intragranular precipitate colony in alloy EV31A is shown in Fig. 7, revealing a high density of ultra-fine (100nm thickness) plates. Since the density, size, and distribution of these precipitates are the principal factors governing the mechanical properties of the cast alloy, it is important to determine whether the presence of the flow-line has any significant effect on the precipitate distribution. Referring to Fig. 8, we note that the flow line is difficult to detect visually. Indeed, the flow line is marked only by a slight increase in the incidence of intergranular plate colonies. Another notable feature is the absence of grain boundary precipitates within the flow line region which extends from the indicated boundary out of the field of view to the left. The grain boundary precipitates are clearly apparent on the right side of the flow-line. Microhardness measurements, taken across this flow-line boundary, do not indicate any change in hardness across this boundary, as shown in Fig. 8(a). Similar results were obtained for sample EV31A-D, as shown in Fig. 8(b).

Alloy ZE41

A representative backscattered electron image, showing the microstructure of the ZE41 alloys is shown in Fig. 3(c). There are two distinct phases identifiable in the microstructure, based on the atomic-weight contrast of the backscattered electron image. First, there is the primary dendritic phase, exhibiting a clear globular or secondary dendrite arm morphology. This Mg solution phase has a low average molecular weight and appears dark in the image.

Figure 7 A high-magnification secondary electron image, showing a colony of Zr-rich fine plate precipitates in alloy EV31A (sample A).

Second, there is an interdendritic phase which occupies a considerable portion of the interdendritic channels, appearing to be nearly continuous in some places. EDS measurements of phase compositions are summarized in Table 4. The results indicate interdendritic precipitates are extremely rich in Zn and Ce and strongly enriched in the other rare earth elements (La, Nd). In addition, the etching pattern reveals a composition profile within the primary phase, where the center of the dendritic regions are enriched in Zn and Zr. This is graphically illustrated by the qualitative dot maps shown in Fig. 9.

Table 4 EDS analysis of alloy ZE41 (sample 3B). (Values in wt. percent.)

<i>Element</i>	<i>Primary Phase</i>		<i>Interdendritic Phase</i>
	<i>Inner</i>	<i>Outer</i>	
Mg	91.9	95.5	38.7
Zn	5.0	3.9	39.0
Zr	3.1	0.5	0.2
La	-	-	6.2
Ce	-	-	13.2
Nd	-	-	1.1

Examining now the microstructural effects of the flow-line, Fig. 10 shows that there is a

Figure 8 Optical micrograph showing the structure of alloy EV31A and corresponding microhardness measurements. Note that the scale of the microhardness measurements is compressed when compared with that of the optical image. Only the flow line location is consistent.

Figure 9 Electron backscattered image (top) for alloy ZE41 (sample 3A) and corresponding dot maps showing the distribution of elements in the different phases.

Figure 10 Optical micrographs for alloy ZE41 (sample 3B) demonstrating a clear change in grain size across the sample. This change coincides with the location of the flow line, which is roughly in the middle of the upper image.

significant change in grain size across the flow-line in alloy ZE41 (sample 3B). Quantitative grain size measurements are shown in Fig. 11, where the grain size profile has been measured using a circle-intercept method and plotted across the flow-line boundary. Fig. 11 shows, however, that the measured gradient in grain size does not correspond to any measurable variation in microhardness (VHN) across this same region. Finally, we note here that composition and grain size variation was not observed in the flow-line region of the other ZE41 sample (sample 3A), as shown in Fig. 12. Additional testing is required to more definitively quantify and understand the direct correlation between the flow-line and structural/chemical gradients.

Tension Testing and Fracture Analysis

The mechanical properties of each alloy were investigated through standard uniaxial tension testing, using a screw-driven load frame in displacement control with a constant crosshead

Figure 11 Microhardness and grain size measurements for alloy ZE41 (sample 3B). There is a noticeable change in grain size at the flow line, though this change in grain size does not significantly affect the measured hardness.

speed of 0.2 mm/min. Elongation was measured using an extensometer attached to the sample. Two specimens of each alloy were tested, and the results are summarized in Fig. 13 and Table 5. Each tension specimen (provided by Wellman Dynamics) was sectioned from a casting so that it contains a flow-line within the test gage length. However, in no case did any specimen fracture at the indicated flow-line location. Images of the tensile specimen fracture surfaces from the WE43B, EV31A, and ZE41 alloys are shown in Figs. 14, 15, and 16, respectively. Both transgranular and intergranular fracture modes are observed, but no fracture surfaces exhibit any variation in appearance that would indicate a change in fracture mechanism associated with the presence of a flow-line. Some grain boundary precipitates are evident on the fracture surface, however, which may suggest that the fracture behavior would be influenced by the flow-line, which was shown in Fig. 8 to affect the distribution of grain boundary precipitates.

An additional test was performed to investigate effect of the flow-line on the local fracture mechanism. Rectangular bars were cut from the “metallurgical analysis” specimens provided by Wellman Dynamics. A small groove was machined into each specimen directly at the flow-line location. Each specimen was then loaded to fracture in a three-point bend geometry, forcing the fracture to occur at the flow line. Fracture surfaces from this test are shown in Fig.

Figure 12 Microhardness, grain size, and composition measurements for alloy ZE41 (sample 3A). There is no noticeable change in hardness, grain size, or composition across the sample.

Figure 13 Uniaxial tension test results for all specimens.

Table 5 Summary of tensile results for all samples.

Sample	E (GPa)	$\sigma_{0.2}$ (MPa)	UTS (MPa)	EL (%)
WE43-V17B	46	165	225	3.2
WE43-V17C	45	180	230	1.9
EV31A-C	45	155	250	2.7
EV31A-I	47	170	255	2.3
ZE41-1	48	165	225	3.9
ZE41-2	42	160	235	5.8

Figure 14 Representative fracture surfaces of alloy WE43B (samples V17B and V17C).

17. In all cases, there is no substantial difference between the flow line fracture surface and the fracture surfaces from the tension specimen fracture surfaces shown in Figs. 14–16.

Conclusions

This investigation into the influence of “flow-lines” on the microstructure, chemical segregation, and mechanical properties of Mg-based castings involved preliminary microstructural analysis, measurement of the variation of local composition and microhardness, tension testing, and a cursory examination of fracture surfaces. The results show that, in some cases, there may be measurable variation in microstructural, microchemical, and micromechanical properties associated with the presence of a “flow-line”, as indicated by radiographic inspection. However,

Figure 15 Representative fracture surfaces of alloy EV31A (samples C and I).

Figure 16 Representative fracture surfaces of alloy ZE41 (samples 1 and 2).

Figure 17 Representative fracture surfaces observed after fracture by three-point bending with a pre-machined groove at the flow-line location. The appearance of the fracture surface is similar to the tensile specimen fracture surfaces for all three test alloys.

there is no evidence to suggest that any of these effects would have a deleterious impact on the macroscopic (bulk) mechanical properties of the alloy component. It must be clearly stated here that this is a *preliminary* finding, and that more systematic and rigorous testing must be performed before any general statements can be made definitively. Such work would include more comprehensive tension test experiments, utilizing a larger statistical sampling of specimens removed from bulk castings with clearly defined flow-line indications. Fracture toughness ($J - R$ curve and J_{IC}) testing is also recommended for quantification of fracture behavior. In such a research program, mechanical testing would be followed by (i) measurement of composition profiles using an electron probe micro-analyzer, (ii) phase identification using X-ray diffraction, and (iii) microchemical analysis of fracture surfaces using energy/wavelength dispersive spectroscopy. An essential feature of this experimental work would be careful selection of test castings and test specimen locations within the castings, based on X-ray inspection methods and casting simulation. Finally, rigorous thermodynamic modeling is recommended to facilitate computation of expected multicomponent solidification paths and the associated segregation profiles. This systematic investigation could lead to an alloy-sensitive set of tolerances for radiographic flow-line indications that would make the necessary allowances for the differences in X-ray absorption behavior associated with the different alloying elements in Mg-based casting alloys. Indeed, the establishment of nonuniform tolerances is likely to emerge as a critical issue in the development of new Mg-based casting alloys.

ADDITIONAL ANALYSES OF THREE-DIMENSIONAL MICROSTRUCTURES

Tim Hosch

Introduction

This section details additional methods of characterization of three-dimensional microstructures that have not been incorporated into a complete manuscript.

Liquid structure quantification using a minimum spanning tree analysis

The arrangement of atoms in a liquid phase is significantly different than in solids, with liquid structures generally displaying no long range order. However, recent experimental evidence has suggested that liquid and supercooled metal structures may contain ordered atom clusters in the form of icosahedral or other three-dimensional structures. Understanding the structure of liquids and supercooled metals is important when dealing with glass forming systems, such as CuZr. However, it is difficult to measure the actual atomic arrangement of a liquid in any real system due to the scales involved. A more feasible approach is to simulate the behavior of a supercooled/liquid system using molecular dynamics. With this approach, exact atomic positions can be simulated for systems of thousands of atoms. Molecular dynamics simulations of liquid structures can then be compared to the structure of solid systems, or idealized structures containing clusters of icosahedral or other three dimensional structures in order to extract any underlying structural similarities. This section details initial attempts at applying a minimum spanning tree analysis to liquid phase structures, solid phase structures,

and synthetic icosahedral structures in order to extract meaningful quantitative descriptions of the liquid phase system.

Minimum Spanning Tree

The minimum spanning tree (MST) of a set of points is the set of lines that connects all points to each other with the shortest (or lowest value) path possible, as shown in Fig. 1. Common applications of the minimum spanning tree are forming a network or planning airline routes, in which case the value of a connection between two points could be set equal to the cost of connecting those two points, and then the minimum spanning tree would specify the lowest cost method of connecting all points. The minimum spanning tree can also be applied to atoms, in which case a connection between two atoms specifies a nearest neighbor link between the two. In this manner, the minimum spanning tree of a set of atoms represents a measure of local structure. This local structure can then be quantified through statistics obtained from the minimum spanning tree, and these statistics will serve to describe to some extent the local structure of a set of atoms.

Data to be obtained from a minimum spanning tree of atoms includes segment length, angles between segments, junction type, and segment type. Junction type specifies the number of connections there are to a specific atom in the minimum spanning tree, while segment types specifies what two junctions an individual segment connects. For example, if atom A is connected to three other atoms in a minimum spanning tree, then atom A will have a junction type of 3, which will be referred to as J3 herein. Then, if one of the atoms connected to A, called atom B, is not connected to any other atoms, then atom B will have a junction type of 1 and the segment between atom A and atom B has a segment type of 1-3, which is referred to as S1_3 for the present analysis.

Algorithm

Brute force calculation of the MST of a set of points can take up to $O(n^3)$ time complexity, which makes calculation of systems over a few thousand points prohibitively long, even on

Figure 1 Example of a minimum spanning tree constructed between a set of 11 points.

modern processors. Therefore, the use of a more efficient algorithm for calculation of the MST was required. Consider a graph of points (or vertices) with V number of vertices and a set of possible connections between them known as edges, with number E . E is at most $V(V - 1)/2$ if all connections between points are possible (a complete graph). In the case where only a relatively small amount of points are connected, then two commonly used and efficient algorithms for calculating the MST are Prim's algorithm (1) and Kruskal's algorithm (2). Prim's algorithm builds a spanning tree by starting with an arbitrary vertex and adds the minimum value vertex to the tree one at a time. Kruskal's algorithm starts with the minimum weight edge and adds edges of minimum weight to the tree until complete. These algorithms have time complexities near $O(E \log(V))$, which is reasonable in cases of sparsely populated graphs (E on the scale of V), but is significantly less efficient for a complete graph, where E is on the scale of V^2 . This is problematic for the present case, where we are concerned with calculation of the minimum spanning tree of a set of points, because applying these algorithms requires construction of a complete graph of all points. A more efficient approach is to first calculate the Delaunay triangulation of the set of points, which generates a more

sparsely populated graph which may be used for efficient calculation of the MST by either Prim's or Kruskal's algorithm. An example of an algorithm to efficiently calculate a Delaunay triangulation in three dimensions is presented by Cignoni *et. al.* (3). However, implementation of this algorithm was deemed more advanced and time consuming than necessary for the present analysis. Therefore, an original, simple, and efficient algorithm for calculation of the MST directly from a set of points was developed and is presented here.

The basis of the developed algorithm is the creation of a PointGrid, which is a data structure that compartmentalizes points on the basis of their location. As shown in Fig. 2, points are separated into grid boxes based upon their location. After creation of the PointGrid, and given the coordinates of an arbitrary point, it is trivial to calculate which grid box the point lies in, and which grid boxes are adjacent to the point. Each grid box can be checked in turn whether it contains a point, and in this manner only a few neighboring points need to be checked to determine the nearest neighbor of any point in the PointGrid. Determination of neighboring points in a PointGrid thus does not depend on of the number of overall points.

Pseudocode for the operation of the PointGrid algorithm for calculation of the MST is shown in Fig. 3. The basic process is to create two PointGrids, one containing points that are not yet added to the MST (NonMSTGrid), and the other containing points that have been added to the MST (MSTGrid). An arbitrary point is first added to the MST, then the NonMSTGrid is searched to find the nearest neighbor points to the one just added to the MST. Each point to point distance is called a segment. The segments between MST points and non MST points are stored in a min binary heap and sorted by the length of the segment. The FindNearest function in the PointGrid adds the segments between the point being searched and its nearest neighbors in the grid. The points nearestPoint_MST and nearestPoint_NonMST are then returned from the segment at the top of the binary heap (the shortest segment), and nearestPoint_NonMST is added to the MST if it has not already been added. After each point is added, its neighbors in the NonMST PointGrid are added to the binary heap. Then, after removing each segment from the top of the heap, the nearest neighbors to the point in MSTGrid which was part of the segment are found and added to the heap. This process

Figure 2 Schematic of PointGrid structure, which separates points into individual grid boxes based upon their location.

repeats until all points have been added to the MST. A full listing of the C++ code used to calculate the MST is presented in Appendix A.

Structures

The MST was calculated for two types of structures in the present analysis: idealized structures created for the analysis, and structures calculated using molecular dynamics (MD) simulations.¹ Additionally, an attempt to simulate noise was made in the calculation of idealized structures. Noise was applied by calculating a random, normally distributed offset distance along with two random angles to determine the offset direction. The offset distance, r , was calculated as the inverse of the cumulative distribution function with a random number chosen as the value, a mean of 0, and a standard deviation of $x\%$ of the average spacing of the structure. The rotation angle, θ , was chosen randomly in the range $0 \leq \theta < 2\pi$. The tilt angle, ϕ , was calculated from $\phi = \arcsin(R)$ where R is a random number from 0 to 1. This offset was independently calculated and applied to each atom in the simulated structure for a range

¹Molecular dynamics simulations were performed at Ames Laboratory

```

Var Type      Var Name
Point:        nearestPoint
PointList:    points
PointGrid:    NonMSTGrid, MSTGrid

Function CalculateMST(points)
  Create NonMSTGrid(points)
  Create MSTGrid
  NonMSTGrid.Remove(points[0])
  MSTGrid.Add(points[0])
  NonMSTGrid.FindNearest(points[0])
  while NonMSTGrid != Empty
    nearestPoint_MST = NonMSTGrid.NearestPointMST
    nearestPoint_NonMST = NonMSTGrid.NearestPointNonMST
    if nearestPoint_NonMST is not in MSTGrid
      NonMSTGrid.Remove(nearestPoint_NonMST)
      MSTGrid.Add(nearestPoint_NonMST)
      NonMSTGrid.FindNearest(nearestPoint_NonMST)
  NonMSTGrid.FindNearest(nearestPoint_MST)

```

Figure 3 Pseudocode for the PointGrid algorithm used for calculation of the MST.

of deviations from 0 to 50%. All structures were normalized to a similar volumetric length scale using the formula: $\lambda_{norm} = \left(\frac{V}{N}\right)^{1/3}$ where $V = (x + \lambda_{mean})(y + \lambda_{mean})(z + \lambda_{mean})$, N is the number of atoms in the structure, and x , y , and z are the respective maximum dimensions of the system in each direction.

Idealized structures created for the calculation of MST include BCC, FCC, HCP, SC, and a completely random structure. Additionally, an icosahedral structure was constructed by stringing individual icosahedra in a simple cubic type arrangement. The top and bottom atoms of adjacent icosahedra are shared, while a separation of one atomic distance is maintained on the sides, shown in Fig. 12.

Structures simulated from MD include an FCC copper structure simulated at temperatures over the range 200–1400 K, a BCC zirconium structure over the range 1000–2200 K, and a CuZr system (50% Cu, 50% Zr by atomic percent) over the range 100–1800 K. The CuZr system was rapidly quenched from 1800 K to each temperature in separate simulations and undergoes a glass transition near 750 K. This allows for the comparison of liquid structures, solid structures, and supercooled liquid (glass) structures.

Results and Analysis

Simulated BCC and FCC structures at deviations from 0 to 50% were compared to BCC and FCC structures calculated from MD simulations. Comparisons were made between the ideal and MD simulated structures for the MST mean segment length, mean angle, junction type frequency, and junction type length and are shown in Figs. 4–9. The percent deviation range over which the ideal structures most closely approximate the simulated MD structures was estimated for each MST statistic and is listed in Table 1. While there is some variance in agreement between the different measurements, the ideal results appear to agree most closely with the MD simulated results from about 10–15% deviation. This suggests that the effect of varying temperature on the atomic structure is somewhat similar to the effect of adding random noise to the positions of atoms in an ideal structure, and in the present case the temperature range of 200–1400 K for the copper FCC system and 1000–2000 K for the zirconium BCC system is similar to a noise level of 10–15%.

Comparisons of the relationship between mean and standard deviation of both segment lengths and segment angles for the ideal and MD simulated structures studied are shown in Figs. 10 and 11, respectively. In Fig. 10, it is apparent that the ideal and simulated structures display a significantly different relationship between the mean and standard deviation of segment length. The simulated structures are significantly below their ideal counterparts, and instead are close to the location of the less closely packed simple cubic and icosahedral structures. All structures approach the top left of the graph with increasing noise/temperature. In turn, the completely random structure is at the top left of the graph and its MST behavior does not change with applied noise (as the structure is already fully random). The mean-standard deviation relation of segment angles in Fig. 11 displays significantly different behavior. In this case, none of the simulated structures line up with the ideal structures. Instead, the standard deviation of segment angles is much higher for the simulated structures. Again, all structures generally approach the position of the randomized structure with increasing noise/temperature. These results indicate that despite some similarities between simulated and ideal structures there are also significant differences which can be detected by the MST analysis. Further, the

Table 1 Percentage deviation range over which each MST statistic category is most similar for the ideal structures versus the MD simulated structures.

	Length Mean	Angle Mean	Junction Freq.	Junction Length
FCC	9–14%	9–18%	18–22%	9–14%
BCC	12–14%	16–18%	0–10%	11–14%

MST analysis detects significant differences in the segment angle relationships between the BCC, FCC, and CuZr structures, though differences in segment lengths are less pronounced between each simulated structure.

Normalization length, segment length mean, and angle mean MST statistics for the MD simulated CuZr system are shown in Fig. 13. From the results, an apparent transition is notable around 700 K, which coincides closely to the glass transition temperature calculated for this system. However, the temperature of the transition varies between the three measurements presented in Fig. 13. The transition in normalization length occurs below 650 K, the transition in segment length mean occurs at 700 K, and the transition in angle mean occurs at 750 K. The normalization length transition is related to the changing volume of the structure, which is likely affected by differential thermal expansion values of the glassy versus liquid structures. However, the transitions in MST angle and segment mean begin to appear before the volume-induced normalization length transition, perhaps suggesting some local structural rearrangement related to the glass transition is occurring before the difference in thermal expansion becomes noticeable. The length scale normalization was removed from the segment length mean results as shown in Fig. 14 in order to verify that the segment length transition is not dependent on the normalization. The same transition is seen at 700 K, though overall the data contains a greater degree of scatter due to slightly different structure volumes. It is also interesting to note that despite decreasing volume at lower temperatures, the average MST segment length increases with decreasing temperature. This is likely due to the decreased randomness in the structure having a large effect on increasing the average MST segment length.

The relationships between standard deviation and mean for both the segment length and segment angle MST measurements are displayed in Figs. 15–16. The transition in the segment length data in Fig. 15 is difficult to detect, though faintly visible as a slight transition in slope in addition to a jump in mean segment length at and below 700 K. The transition in segment angle data in Fig. 16, however, is very clear, and results in two disparate groupings of data points. Above the transition at 750 K, the angle mean decreases slowly with decreasing temperature, while below 750 K the angle mean displays a sudden decrease and begins to increase with decreasing temperature. These changes in the angle of MST junctions seem to reflect a difference in local atomic structure between the glassy and liquid states.

As shown in Fig. 17, the dominant junction frequency at all temperatures in the quenched CuZr structure is J1, and the dominance of J1 increases at lower temperatures at the expense of J2 frequency. When the segment length data are separated by junction type, the glass transition only shows up strongly in J3-type junctions, as shown in Fig. 18. The transition is not apparent in either J1 or J2-type junctions, and may be weakly present, if at all, in J4 and J5-type junctions. Examination of segment length separated by segment type reveals similar behavior for any segments containing a J3 junction, as shown in Fig. 19. Structural changes associated with the glass transition appear to be localized to the J3-type junctions of the MST.

The MST segment frequency separated by atom type is displayed in Fig. 20 while the corresponding segment lengths are shown in Fig. 21. It is apparent that Cu-Cu segments display the glass transition noted previously in J3-containing segments while CuZr segments do not. Interestingly, Zr-Zr segments seem to display an opposite transition around the glass transition temperature, with segment lengths increasing more rapidly with decreasing temperature. However, this could be a result of the decreasing frequency of Zr-Zr segments at lower temperatures; near the glass transition temperature, the frequency of Zr-Zr segments drops below 1%, and further decreases to near 0.1% at 100 K, which could reduce the amount to which those segments reflect the true local structure. The effect changing segment frequency has on segment length can be investigated by examining CuZr systems of varying %Cu, the results of which are shown in Figs. 22 and 23. While the frequency of CuZr segments decreases

with increasing copper composition as shown in Fig. 22, the length of CuZr segments starts to increase above a linear trend, shown in Fig. 23. The length of Zr-Zr segments, which also decrease in frequency with increasing copper composition, does not appear to undergo a similar nonlinear increase, though there is relatively limited data above 50% copper. These results may then support the finding that diminishing segment frequency increases segment length. Additionally, it can be seen in Fig. 24 that only the MST angles between copper atoms display a meaningful trend with temperature, while the angles between zirconium atoms do not display such a trend. This is further support that, similar to the importance of J3-type MST junctions, the glass transition is only displayed in copper MST junctions and not zirconium junctions.

Conclusion

- The ideal structures used in the present analysis best approximate the MD simulated structures when a range of 10–15% noise has been added. However, even at this level of noise, the ideal structures are significantly different than the simulated structures, a distinction which can be elucidated by the MST analysis applied herein.
- An MST analysis is effective in indicating the glass transition temperature in the CuZr system by detecting differences in local atomic arrangement. These differences show up in the MST junction angle and MST segment length. The transition is indicated by both a slope change in the relation of segment length mean with CuZr system temperature, and by a change in the behavior of the MST angle mean with temperature. Additionally, the system volumetric length scale indicates the glass transition temperature, though at a lower temperature than the MST transition indications.
- The important atoms in the MST analysis for detection of the CuZr glass transition temperature are atoms of junction type J3 and segments of type Cu-Cu. A more in depth investigation of the use of the MST analysis should focus on isolating these elements in the CuZr structure in order to more fully quantify the glass transition and its effect on the local atomic arrangement.

Figure 4 MST segment length mean comparison for ideal and MD simulated BCC and FCC structures.

Figure 5 MST angle mean comparison for ideal and MD simulated BCC and FCC structures.

Figure 6 MST junction frequency comparison for ideal and simulated BCC structures. Solid lines represent ideal structures while dashed lines are MD simulated structures.

Figure 7 MST junction frequency comparison for ideal and simulated FCC structures. Solid lines represent ideal structures while dashed lines are MD simulated structures.

Figure 8 MST junction length comparison for ideal and simulated BCC structures. Solid lines represent ideal structures while dashed lines are MD simulated structures.

Figure 9 MST junction length comparison for ideal and simulated FCC structures. Solid lines represent ideal structures while dashed lines are MD simulated structures.

Figure 10 Segment length mean versus standard deviation of different structures. The CuZr structures are simulated using MD at temperatures from 100–1800 K, BCC MD are simulated from 1000–2200 K, and FCC MD structures are simulated from 200–1400 K. The rest of the structures were generated with 0–50% noise added.

Figure 11 Segment angle mean versus standard deviation of different structures. The CuZr structures are simulated using MD.

Figure 12 Schematic of packing arrangement adopted to approximate an icosahedral structure.

Figure 13 Normalization length, segment length mean, and angle mean MST statistics obtained from a MD simulated CuZr structure quenched from 1800 K to a range of temperatures. Clear transition points are evident in the data, though the actual transition temperature value includes temperatures of 650 K (in black), to 700 K (in red), and 750 K (in orange).

Figure 14 MST segment length mean from a MD simulated CuZr structure without length scale normalization. The data display the same transition in slope at 700 K noted in the normalized data, though the data also display increased scatter.

Figure 15 MST segment length standard deviation versus mean of a MD simulated CuZr structure. The glass transition around 700 K is fairly faint but noticeable.

Figure 16 MST angle standard deviation versus mean of a MD simulated CuZr structure. The glass transition is readily apparent at 750 K, with the angle standard deviation versus mean relationship showing significantly different trends above and below this temperature.

Figure 17 MST junction type frequency of a MD simulated CuZr structure. Junction frequency decreases with junction type number from J1 up through J5. Additionally, the dominance of J1-type junctions increases at lower temperatures at the expense of J2 frequency.

Figure 18 MST segment length separated by junction type for a MD simulated CuZr structure. The glass transition at 700 K is only readily apparent in the J3-type junctions. The J3 junction data are shown separately at the right for clarity. The transition does not show up at all for J1 or J2 junctions, and may be faintly apparent if at all in the J4 and J5 junction types.

Figure 19 MST segment length separated by segment type and showing only the first three J3-type segments for a MD simulated CuZr structure. The glass transition near 700 K shows up in all three segment types. This transition does not show up in any segments not containing a J3 junction.

Figure 20 Frequency of MST segment types separated by junction atoms in a MD simulated CuZr structure. Copper to copper segments are significantly more frequent than zirconium to zirconium connections, which is likely due to the larger zirconium atom.

Figure 21 Length of MST segment types separated by junction atoms in a MD simulated CuZr structure. Due to atom size, Zr-Zr segments are the longest, followed by CuZr, and Cu-Cu segments are the shortest. Note the transition in the Cu-Cu data and the increase in Zr-Zr segment length at lower temperatures.

Figure 22 Frequency of MST segment types separated by junction atoms in a MD simulated CuZr structure of varying composition at 300 K. With increasing fraction of copper the prevalence of Zr-Zr segments plunges to near 0 at 50% copper.

Figure 23 Length of MST segment types separated by junction atoms in a MD simulated CuZr structure of varying composition at 300 K. Length of Zr-Zr segments increases linearly until disappearing above 57 atomic %Cu, while the length of CuZr segments begins to increase rapidly above 57% copper.

Figure 24 MST angle mean as a function of temperature in a MD simulated CuZr structure. The angles between copper atoms display a similar trend with temperature as the overall MST data, while the angles between zirconium atoms do not appear to display any meaningful trend.

Local microstructure characterization through use of the radial distribution function

The radial distribution function (RDF) is commonly used in systems of atoms to determine how, on average, atoms are packed around each other. It describes the average density of particles as function of distance away from a central particle. When averaged among all atoms in a system, the RDF thus provides a measure of average system spacing. Calculation of the RDF is done as:

$$g(r) = \frac{dN}{dV} \frac{V}{N} \quad (1)$$

where $g(r)$ is the RDF, N is the total number of particles, and V is the volume of the system. Similarly, in two dimensions the RDF is

$$g(r) = \frac{dN}{dA} \frac{A}{N} \quad (2)$$

where A represents the area of the system. In the case of two dimensions, a system may be split up into concentric circular sections about a particular atom and a small distance apart, dr . In this case, dA represents the difference in area between neighboring circular sections, and dN represents the difference in atom count between the neighboring sections. Therefore,

$$dA = A_1 - A_0 \quad (3)$$

$$A_1 = \pi (r + dr)^2 \quad (4)$$

$$A_0 = \pi r^2 \quad (5)$$

The expansion of A_1 is

$$A_1 = \pi (r^2 + 2rdr + dr^2) \quad (6)$$

This gives

$$dA = \pi (2rdr + dr^2) \quad (7)$$

and the RDF is

$$g(r) = \frac{dN}{\pi (2rdr + dr^2)} \frac{A}{N} \quad (8)$$

The value of dr can be arbitrarily chosen and is referred to as the “bin” size.

For the present analysis, instead of atoms, the RDF was applied to two dimensional microstructure sections of the irregular eutectic Al-Si system in an attempt to more accurately calculate the average system spacing. Samples of Al-13wt% Si were directionally solidified in a Bridgman-type furnace at velocities of 10 $\mu\text{m/s}$ to 950 $\mu\text{m/s}$. Samples were sectioned at least 10 mm behind the final quench interface to obtain a representative microstructural sample, polished, and then imaged with either an optical microscope or SEM depending on microstructural scale. Microstructural images were converted to black and white “masks” to allow digital image processing and calculation of the RDF.

Calculation of the RDF

When calculating the RDF for the Al-Si system, there will be some noise added due to the discretization of silicon particle positions into standard images containing pixels in a rectangular grid. One influence of this discretization is that the frequency of certain pixel distances is higher than others. Consider, for example, the calculation of the RDF around the central pixel of an all white square image 5 pixels by 5 pixels in size. This is shown in Fig. 25, with each pixel enlarged to a box and the square of the distance from the central pixel written inside each pixel. It is seen that the frequency of the distance square 5 is twice that of any of the other distances. Also, 4 pixels will be contained within a circle of diameter 1 around the center pixel as compared to 8 pixels within a circle of diameter 2. This will cause the RDF to be fairly jagged for short distances, though this effect will smooth out at larger distances.

The RDF was calculated individually for each silicon particle by calculating the distance from each pixel in the silicon particle to the pixels of the surrounding silicon particles. As shown in Fig. 26, the RDF is effective in determining the local spacing around a particle. The three concentric circles in Fig. 26 determined by the RDF effectively pick out the locations of nearby silicon particles. The average eutectic spacing around an object can then be determined by measuring the location of the first peak in the RDF for that particle. This process was performed for anywhere from 200 to 2000 objects for a particular solidification velocity in order to obtain sufficient data to determine the distribution of local spacings in the irregular

structure.

Results

Cumulative probability distributions as a function of spacing for a range of solidification velocities are shown in Fig. 27. The general relationship of decreased average particle spacing with increased velocity is evident in the results, and is clearly shown in Fig. 28 as well for three different cumulative probability levels. However, also clear from Fig. 27 is the large range of spacings encountered in the structures. For example, the top 5% of spacings in the 250 $\mu\text{m/s}$ structure are similar to the average spacing in the 50 $\mu\text{m/s}$ structure, while the bottom 5% of spacings in the 50 $\mu\text{m/s}$ structure are similar to the average spacing in the 250 $\mu\text{m/s}$ structure. With this inherent variation in the Al-Si structure, it is evident why the flake to fiber transition occurs over a range of solidification velocities rather than at a discrete velocity. In fact, if one were to assume that a modified fibrous structure begins to form when local solidification conditions elicit a eutectic spacing of roughly 1.7 μm , then the spacing distribution results presented in Fig. 27 would suggest that the microstructure is 5% modified at 50 $\mu\text{m/s}$, 50% modified at 250 $\mu\text{m/s}$, and 95% modified at 950 $\mu\text{m/s}$, a progression which reasonably approximates that which may be observed visually and quantitatively in the Al-Si structure (see Hosch *et. al.*(4)).

Calculation of the critical diameter function in an Al-Si alloy

The critical diameter metric was described by England in (5) and Napolitano and England in (6). The critical diameter is defined as the minimum diameter of an infinite cylinder aligned along the growth direction which provides a representative measure of the overall volume fraction of phases. More specifically, the critical diameter is the cylindrical diameter above which the standard deviation of the volume fraction falls below 15% of the mean.

The critical diameter was recalculated here for a set of directionally solidified Al-13wt% Si samples over a velocity range of 10 $\mu\text{m/s}$ to 2000 $\mu\text{m/s}$. The critical diameter measurements are shown in Fig. 29. A general relationship of $d_f = 260V^{-0.6}$ in μm and $\mu\text{m/s}$ was observed,

8	5	4	5	8
5	2	1	2	5
4	1	0	1	4
5	2	1	2	5
8	5	4	5	8

Figure 25 Schematic of pixel distance frequency around a central pixel. Each box represents a pixel of an image, and the numbers in each pixel represent the square of the distance between that pixel and the central pixel. Each distance occurs four times, except for 5, which occurs eight times.

Figure 26 Example calculation of the radial distribution function (RDF) demonstrating its effectiveness in determining average local spacing. The three peaks of the RDF function in this example correspond to the concentric circles drawn around the object being measured.

Figure 27 Cumulative probability distributions for the average spacing around a silicon particle for a range of solidification velocities.

Figure 28 Relationship of spacing versus velocity for the average particle spacing determined at three different cumulative probability values.

Figure 29 Critical diameter as a function of velocity. The measurement contains significant scatter and it is difficult to determine an overall trend.

though there appears to be a significant amount of variance in the critical diameter measurements. Due to this variance, it is difficult to tell whether this measurement indicates a change in morphology over the flake to fiber transition range.

References

- [1] R. C. Prim. Shortest connection networks and some generalizations. *Bell System Technical Journal*, 36:1389–1401, 1957.
- [2] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7:48–50, 1956.
- [3] P. Cignoni, C. Montani, and Scopigno R. Dewall: A fast divide and conquer delaunay triangulation algorithm in e d. *Computer-Aided Design*, 30(5):333–341, 1998.
- [4] T. Hosch, L. G. England, and R. E. Napolitano. Analysis of the high growth-rate transition in al-si eutectic solidification. *Journal of Materials Science*, 44(18):4892–4899, 2009.

- [5] L. G. England. High-rate growth-mode transitions in al-si eutectics. Master's thesis, Iowa State University, 2004.
- [6] R. E. Napolitano and L. G. England. High-rate growth-mode transitions in al-si eutectics. *Solidification of Aluminum Alloys*, pages 445–451, 2004.

CONCLUSION

The recent push for increased energy efficiency in the automotive and aerospace industries has placed extra importance on the continued development and application of lightweight structural materials such as irregular eutectic Al-Si and magnesium casting alloys. In this study, the local solidification conditions, mechanisms, and tensile properties associated with the flake to fiber growth mode transition in Al-Si eutectic alloys were investigated using bridgman type gradient-zone directional solidification. Tensile testing highlighted the importance of obtaining a modified silicon structure in Al-Si alloys, with an attendant increase in ductility, tensile reproducibility, and tensile strength. The solidification mechanisms contributing to silicon modification were investigated with quantitative image analysis of two-dimensional sections and careful observation of deep-etched sections showing the three-dimensional microstructure. The transition was found to occur in two stages: an initial stage occurring from 100–500 $\mu\text{m/s}$ that is dominated by in-plane plate breakup and rod formation within the plane of the plate, and a second stage occurring from 500–950 $\mu\text{m/s}$ where the onset of out-of-plane silicon rod growth leads to the formation of an irregular fibrous structure. Several microstructural parameters were investigated in an attempt to quantify this transition, and it was found that the particle aspect ratio is effective in objectively identifying the onset and completion velocity of the flake to fiber transition. Furthermore, despite descriptions of flake to fiber onset and completion velocities, a range of silicon growth mechanisms were found to be present at all velocities studied (10–2000 $\mu\text{m/s}$), and the flake to fiber transition, as reported, simply reflects those velocities at which the dominant growth mechanism changes.

The appearance of out-of-plane silicon protrusions midway through the flake to fiber transition suggests that the occurrence of three-dimensional out-of-plane silicon growth is vital

to the formation of the modified fibrous microstructure. Further investigation of these silicon protrusions was performed by adapting a perturbed-interface stability analysis to the Al-Si system. It was concluded that both in-plane and out-of-plane plate breakup is initiated by a similar Mullins-Sekerka type instability though operating on a different crystallographic plane. The characteristic in-plane silicon plate breakup length scale is smaller than the corresponding length scale of out-of-plane protrusions due to the expected larger interfacial stiffness experienced by perturbations perpendicular to the plane of the plate, a phenomenon that is accurately predicted by the perturbed interface stability model. Furthermore, measurements of silicon equilibrium shape particles suggested the minimum anisotropy of the solid Si/liquid Al-Si system is on the order of $\epsilon_4^{001} \approx 0.1$, and incorporation of this silicon anisotropy into the perturbed interface model was found to improve prediction of the instability length scale.

The benefit of the extreme lightweight nature of magnesium alloys comes at a cost of decreased high temperature performance. Additions of heavier elements, such as zinc, zirconium, and rare earth elements, can significantly improve the high temperature performance of magnesium alloys, though this introduces additional problems, including the appearance of segregational flow-lines. The effect of flow-line indications in WE43B, EV31A, and ZE41 alloys were investigated here in order to determine the severity of their effect on mechanical properties. The results of a preliminary investigation show that, in some cases, there may be measurable variation in microstructural, microchemical, and micromechanical properties associated with the presence of a flow-line. However, there is no evidence to suggest that any of these effects would have a deleterious impact on the macroscopic (bulk) mechanical properties of the alloy component. Further studies are suggested that would allow for a more rigorous and comprehensive determination of the effects of flow-line indications, as well as developing a set of tolerances to guide the use and development of new magnesium-based casting alloys.

A minimum spanning tree (MST) analysis was found to be effective in determination of some aspects of local atomic structure. A set of ideal structures generated with a normally distributed noise parameter was found to best approximate molecular-dynamics simulated structures when a range of 10–15% noise was added. However, even at this level of noise, the

ideal structures are significantly different than the simulated structures, a distinction which was elucidated by the MST analysis. The MST analysis is also effective in detecting the glass transition temperature in the CuZr system, as indicated by both a slope change in the segment length as a function of temperature, and by a change in the behavior of the MST angle mean as a function of temperature. The important atoms in the MST analysis for detection of the CuZr glass transition temperature were found to be atoms with three nearest neighbors in the MST (junction type J3) and segments of type Cu-Cu. A more in depth investigation of the use of the MST analysis should focus on isolating these elements in the CuZr structure in order to more fully quantify the glass transition and its effect on the local atomic arrangement.

The radial distribution function (RDF) was found to effectively identify the average local eutectic spacing around an individual silicon particle as defined on a two-dimensional microstructural section. The RDF was used to define the cumulative probability of a particular eutectic spacing value as a function of solidification velocity. Comparison of the frequency of eutectic spacings in different solidification velocities found 5% of eutectic spacings were below an arbitrarily selected $1.7 \mu\text{m}$ at $50 \mu\text{m/s}$, 50% were below it at $250 \mu\text{m/s}$, and 95% were below it at $950 \mu\text{m/s}$, a progression which reasonably approximates the modified fraction of the silicon microstructure. This microstructural variation can explain the wide range of velocities over which the flake to fiber transition has been observed to occur in Al-Si eutectic alloys.

Calculation of the critical diameter, which is defined as the cylindrical diameter above which the standard deviation of the volume fraction falls below 15% of the mean, was performed on the Al-Si eutectic system over a velocity range of $10 \mu\text{m/s}$ to $2000 \mu\text{m/s}$. A general relationship of $d_f = 260V^{-0.6}$ in μm and $\mu\text{m/s}$ was observed, though there appears to be a significant amount of variance in the critical diameter measurements. Due to this variance, it is difficult to tell if this measurement truly indicates a change in morphology over the flake to fiber transition range.

APPENDIX

This appendix provides a listing of the code used to generate many of the measurements and results herein.

Minimum spanning tree

The following code was used to calculate the minimum spanning tree of a set of points. Input files are required to be in a plain text file where each point is represented by a separate line in the format *lf lf [lf] [d]* where *lf* is a long float (or double) specifying the point location, and *d* is an integer specifying the type of the atom. The final two parameters are optional, depending on whether the MST is two or three dimensional, but if the type specifier *d* is used, then all three *lf* parameters must be used as well (*i.e.*, two dimensional MST analysis with atom type specification is not allowed).

The minimum spanning tree application was programmed in C++ and compiled using Microsoft Visual Studio 2008 as a console application.

The current algorithm is highly optimized for point sets with a relatively uniform distribution, which makes it optimal for atom systems. The algorithm does not handle adding the final atoms to the MST efficiently, however. When, for example, only the last percent of atoms are left to add to the MST, the algorithm severely slows down because it is still checking for nearest neighbors from the now densely populated MST PointGrid to the sparsely populated nonMST PointGrid. This could be improved by changing the algorithm near the end of MST calculation to work from the few remaining unadded MST points instead of the large number of added MST points.

Main

This is the code specifying the main function, or entry point, of the MST application.

```
// MST.cpp : Defines the entry point for the console application.
#include "stdafx.h"
#include <cmath>
#include <cstdlib>
#include <cstdio>
#include <iostream>
#include <new>
#include <string>
#include <vector>
#include "point.h"
#include "pointlist.h"
#include "branchlist.h"
#include "MSTcalc.h"
#include "MST_Settings.h"
#include <conio.h>
using namespace std;
int MultiMST(MST_Settings &settings);
vector<string>* GetFileNames(string fname);
int main(int argc, char *argv[])
{
    printf("_____\\n___Minimum_Spanning_\\n___Tree_Calculation____\\n_____\\n\\n");
    string appPath = argv[0];
    int strPos = appPath.find_last_of("\\");
    appPath = appPath.substr(0, strPos + 1);
    MST_Settings settings = MST_Settings(appPath);
    settings.ReadSettings();
    settings.VerifySettings();
    MultiMST(settings);
    _getch();
}
int MultiMST(MST_Settings &settings)
{
    string analysis = "stats_Combined";
    string INPUT_FILE = settings.directory + "MST_FILENAMES.txt";
    vector<string> *structureFiles = GetFileNames(INPUT_FILE);
    int numFiles = structureFiles->size();
    MSTcalc* mstCalc = new MSTcalc(analysis.c_str());
    cout << "\\n***_Beginning_Analysis_***\\n";
    for (int i = 0; i < numFiles; i++)
    {
        cout << "\\nAnalyzing_file_" << i + 1 << "_of_" << numFiles << ":\n" <<
            settings.directory << structureFiles->at(i) << ".txt\\n\\n";
        mstCalc->ConstructMST(settings.directory.c_str(), structureFiles->at(i)
            ).c_str(), settings.gridSize, settings.recalcMST);
    }
    delete mstCalc;
    cout << "\\nAnalysis_finished.\\n";
}
```

```

    delete structureFiles;
    return 0;
}
vector<string>* GetFileNames(string fname)
{
    char BUFFER[255];
    FILE *fp;
    if(fopen_s(&fp, fname.c_str(), "r") != 0)
    {
        printf("\nCannot open input file . . . Enter file names to read in .
        MST_FILENAMES_in_selected_directory.\n");
        return new vector<string>;
    }
    vector<string> *strArray = new vector<string>;
    while (fgets(BUFFER, 255, fp) != NULL)
    {
        if (BUFFER[0] != '\\\' && BUFFER[1] != '\\\'') // allow commenting out of
            file names using //
        {
            strArray->push_back(BUFFER);
            if (strArray->back().at(strArray->back().size() - 1) == '\\n')
                strArray->back() = strArray->back().substr(0, strArray->back().
                    size() - 1);
        }
    }
    fclose(fp);
    return strArray;
}

```

MST_Settings

This class interfaces with the user to load, set up, and save settings pertaining to the calculation of the MST.

```

// MST_Settings.h
#ifndef MST_SETTINGS_H
#define MST_SETTINGS_H
#include <string>
#include <iostream>
#include <cstdlib>
#include <cstdio>
using namespace std;
class MST_Settings
{
private:
    string appPath;
    string GetDirectory();
    string GetDirectoryFromUser();
    double GetGridSize();
    double GetGridSizeFromUser();

```

```

    bool  GetYesNoFromUser(string prompt);
    void  GetNewSettings();
    bool  CheckDirectory();
public:
    MST_Settings(void) {}
    MST_Settings(string applicationPath): appPath(applicationPath) {}
    ~MST_Settings(void) {}
    string directory;
    double gridSize;
    bool  recalcMST;    // Specifies whether to analyze old mst files or
                       // create new ones
    void  ReadSettings();
    bool  SaveSettings();
    void  VerifySettings();
};
#endif

// MST_Settings.cpp
#include "StdAfx.h"
#include "MST_Settings.h"
void MST_Settings::ReadSettings()
{
    string SETTINGS_FILE = appPath + "MST_SETTINGS.txt";
    double grid = -1;
    bool  recalc = true;
    int  recalcTemp = 1;
    char BUFFER[255];
    FILE *fp;
    if (fopen_s(&fp, SETTINGS_FILE.c_str(), "r") == 0)
    {
        fgets(BUFFER, 255, fp); // DIRECTORY
        directory = BUFFER;
        fgets(BUFFER, 255, fp); // GRID SIZE
        if (BUFFER != NULL)
        {
            sscanf_s(BUFFER, "%lf", &grid);
            fgets(BUFFER, 255, fp); // RECALCULATE MST
            if (BUFFER != NULL)
                sscanf_s(BUFFER, "%d", &recalcTemp);
        }
        fclose(fp);
    }
    if (!CheckDirectory())
    {
        cout << "\nLast directory used: \" + directory.substr(0, directory.
            length() - 1) + "\" is invalid.\nPress enter to continue.";
        directory = GetDirectoryFromUser();
    }
    if (grid > 0 && grid < 1000)
        gridSize = grid;
    else
        gridSize = 1.6;
    if (recalcTemp == 0)

```



```

        recalcMST = false;
        SaveSettings();
    }
    bool MST_Settings::SaveSettings()
    {
        string SETTINGS_FILE = appPath + "MST_SETTINGS.txt";
        FILE *fp;
        if (fopen_s(&fp, SETTINGS_FILE.c_str(), "w") != 0)
        {
            return false;
        }
        int recalcINT;
        if (recalcMST)
            recalcINT = 1;
        else
            recalcINT = 0;
        fprintf(fp, "%s", directory.c_str());
        fprintf(fp, "\n%f", gridSize);
        fprintf(fp, "\n%d", recalcINT);
        fclose(fp);
        return true;
    }
    void MST_Settings::VerifySettings()
    {
        bool verify = false;
        while (!verify)
        {
            string recalcString;
            if (recalcMST)
                recalcString = "Calculate_New";
            else
                recalcString = "Analyze_Old";
            cout << "\n-----\nCurrent_MST_Settings:\n
            -----\n\nDirectory:\t" <<
            directory.substr(0, directory.length() - 1) <<
            "\nGrid_Size:\t" << gridSize << "\nCalculation:\t" << recalcString
            << "\n\n-----\n\n";
            verify = GetYesNoFromUser("Are_these_settings_ok?");
            if (!verify)
                GetNewSettings();
        }
        SaveSettings();
    }
    void MST_Settings::GetNewSettings()
    {
        directory = GetDirectory();
        gridSize = GetGridSize();
        cout << "\n";
        recalcMST = GetYesNoFromUser("Do_you_want_to_calculate_new_MST_files?(Y/N)");
    }
    // Asks user for directory which contains MST_FILENAMES.txt and loops
    until a valid directory with such a file is found.

```

```

string MST_Settings::GetDirectoryFromUser()
{
    bool dirFound = false;
    string dirName;
    cin.ignore(1000, '\n');
    while (!dirFound)
    {
        cout << "\nInput_directory_which_contains_\nMST_FILENAMES.txt\n\n";
        nDirectory:;
        getline(cin, dirName);
        directory = dirName;
        dirFound = CheckDirectory();
        if (!dirFound)
        {
            bool analyzeAll = GetYesNoFromUser("\nMST_FILENAMES.txt_not_found,_
            analyze_all_.txt_files_in_directory?");
            cin.ignore(1000, '\n');
            if (analyzeAll)
            {
                string nameMaker = "\"" + appPath + "MST_Namemaker.CL" + "\"";
                string arguments = "\"" + directory + "\"";
                string command = nameMaker + "_" + arguments;
                system(("\" + nameMaker + "_" + "\"" + directory + "\"").c_str());
                ;
                dirFound = CheckDirectory();
                if (!dirFound)
                    cout << "Error_creating_MST_FILENAMES.txt";
            }
        }
    }
    return directory;
}

double MST_Settings::GetGridSize()
{
    if (gridSize > 0 && gridSize < 1000)
    {
        char BUFFER[255];
        cout << "\n";
        sprintf_s(BUFFER, "Current_grid_size: %lf. _\nUse_this_grid_size?(Y/N
        )", gridSize);
        string prompt = BUFFER;
        if (!GetYesNoFromUser(prompt))
            gridSize = GetGridSizeFromUser();
    }
    else
        gridSize = GetGridSizeFromUser();
    return gridSize;
}

double MST_Settings::GetGridSizeFromUser()
{
    bool sizeFound = false;
    double size = -1;
    string sizeString;

```

```

cin.ignore(1000, '\n');
cout << "\nGrid size is specified as a fraction of average spacing.\n";
cout << "Use smaller grid sizes for uniformly distributed structures,\n
        nand larger grid sizes for more variable structures.\n";
while (!sizeFound)
{
    cout << "Input new grid size (1.5-15 recommended, 1.6 default): ";
    getline(cin, sizeString);
    sscanf_s(sizeString.c_str(), "%lf", &size);
    if (size > 0 && size < 1000)
        sizeFound = true;
}
return size;
}
bool MST_Settings::GetYesNoFromUser(string prompt)
{
    bool response = true;
    string input = "";
    while (input != "Y" && input != "N" && input != "y" && input != "n")
    {
        printf("%s", prompt.c_str());
        cin >> input;
        input = input.substr(0, 1);
    }
    if (input == "N" || input == "n")
        response = false;
    return response;
}
// Analyzes the supplied string to see if it is a valid directory, and if
// not calls GetDirectoryFromUser; returns a valid directory
string MST_Settings::GetDirectory()
{
    if (directory == "") // No directory in settings file, make user enter
        a valid one
    {
        cout << "No directory found. You must input a valid directory. Press
            Enter to continue.";
        directory = GetDirectoryFromUser();
    }
    else
    { // Previous directory found in settings file, ask user if they want to
        enter a different one
        string prompt = "\nLast directory used: " + directory.substr(0,
            directory.length() - 1) + "\n\nUse this directory? (Y/N) ";
        if (!GetYesNoFromUser(prompt))
            directory = GetDirectoryFromUser();
    }
    return directory;
}
bool MST_Settings::CheckDirectory()
{
    bool dirOK = false;
    if (directory.length() > 0)

```

```

{
    if (directory.substr(directory.length()-1) == "\n") // Remove null
        termination if necessary
        directory = directory.substr(0, directory.length()-1);
    if (directory.length() > 0 && directory.substr(directory.length()-1)
        != "\\")
        directory.append("\\");
}
string fName = directory + "MST_FILENAMES.txt";
FILE *fp;
if (fopen_s(&fp, fName.c_str(), "r") == 0)
{
    dirOK = true;
    fclose(fp);
}
return dirOK;
}

```

MSTcalc

This class performs calculation and analysis of MST files.

```

// MSTcalc.h
#ifndef MSTCALCH
#define MSTCALCH
#include "branchlist.h"
#include "pointlist.h"
#include "PointGrid.h"
#include <iostream>
#include <cmath>
#include <string>
#include <map>
#include <vector>
#include <ctime>
using namespace std;
const static int BUFFER_SIZE = 256;
struct MeanAndStdDev
{
    double mean;
    double stdDev;
    MeanAndStdDev(): mean(0), stdDev(0) {}
    MeanAndStdDev(double mn, double stdD): mean(mn), stdDev(stdD) {}
};
class MSTcalc
{
private:
    branchlist *branches;
    map<point, vector<point>> neighborPoints;
    map<pointlist*, bool> inMST;
    map<string, map<double, double>> junctFreqMap;
    map<string, map<pair<int, int>, double>> segFreqMap;

```

```

map<string, map<double, MeanAndStdDev>> junctMnStdMap;
map<string, map<pair<int, int>, MeanAndStdDev>> segMnStdMap;
map<string, map<int, double>> junctPtTypeFraction; //
    Fraction of each type of junction (atomic fraction)
map<string, map<pair<int, int>, double>> segPtTypeFraction; //
    Fraction of each point type (AA's vs AB's vs BB's)
map<string, map<int, MeanAndStdDev>> mnStdOnePtTypeMap; //
    length mn-std stored by one pt type
map<string, map<pair<int, int>, MeanAndStdDev>> mnStdTwoPtTypeMap;
    // length mn-std stored by neighboring pt types
map<string, map<int, MeanAndStdDev>> mnStdPtTypeAngMap; //
    angle mn-std stored by one pt type
map<string, map<int, MeanAndStdDev>> junctMnStdOnePtTypeMap; //
    junction# mn-std stored by one pt type
map<string, map<int, map<double, double>>> junctFreqOnePtTypeMap;
    // junct freq stored by one pt type
map<string, map<pair<int, int>, map<pair<int, int>, double>>>
    segFreqTwoPtTypeMap; // how many 12's, 23's, etc are there for AA's,
    AB's, BB's
map<string, map<pair<int, int>, map<pair<int, int>, MeanAndStdDev>>>
    segMnStdTwoPtTypeMap; // mean and std of lengths of segment types
    separated by pt type
map<double, bool> junctTypes;
map<pair<int, int>, bool> segTypes;
map<int, bool> ptTypes;
map<pair<int, int>, bool> ptPairTypes;
vector<int> pointTypes;
point ptlistDimensions;
int startingPoints;
double normLength;
bool multiAnalysis; // indicates that a combined statistics file named
    analysisFile should be saved
bool firstRun;
bool multiTypes;
char sampleName[BUFFER.SIZE];
char analysisName[BUFFER.SIZE];
char inFile[BUFFER.SIZE];
char mstFile[BUFFER.SIZE];
char statFile[BUFFER.SIZE];
char analysisFile[BUFFER.SIZE];
char analFreqFile[BUFFER.SIZE];
char analTypeFile[BUFFER.SIZE]; // lengths/angles segregated by point
    type
void ConstructMST(pointlist *unmarked);
void ConstructMSTnew(pointlist *unmarked, double gridSize);
bool CreateFiles(const char* directory, const char* sampleName, bool
    newAnalysis);
void AnalyzeMST();
void FindNeighborPoints();
void GetTypes();
void AnalyzeLengths();
void AnalyzeAngles();
void OutputResults();

```

```

    double CalcMean(const double *doubleArray, int numElements) const;
    double CalcStdev(const double *doubleArray, double mean, int numElements
        ) const;
    template <class T>
    map<T, double>* CalcFreq(const vector<T> &dataArray) const;
public:
    MSTcalc(void): firstRun(true), multiAnalysis(false), branches(NULL) {}
    MSTcalc(const char* analysisName): firstRun(true), multiAnalysis(true),
        branches(NULL)
    { strcpy_s(this->analysisName, BUFFER_SIZE, analysisName);}
    ~MSTcalc(void);
    bool ConstructMST(const char* directory, const char* sampleName, double
        gridSize, bool newAnalysis);
};
#endif

// MSTcalc.cpp
#include "StdAfx.h"
#include "MSTcalc.h"
MSTcalc::~MSTcalc(void)
{
    if (multiAnalysis)
    {
        OutputResults();
        junctFreqMap.clear();
        segFreqMap.clear();
    }
}
// Constructs the relevant file names based on the input information and
// checks that the files can be opened
bool MSTcalc::CreateFiles(const char *directory, const char *sampleName,
    bool newAnalysis)
{
    strcpy_s(this->sampleName, sampleName);
    sprintf_s(inFile, BUFFER_SIZE, "%s%.txt", directory, sampleName);
    sprintf_s(mstFile, BUFFER_SIZE, "%smst_%.txt", directory, sampleName);
    sprintf_s(statFile, BUFFER_SIZE, "%sstats_%.txt", directory, sampleName
    );
    FILE* ofp;
    if (firstRun)
    {
        firstRun = false;
        if (multiAnalysis)
        {
            sprintf_s(analysisFile, BUFFER_SIZE, "%s%.txt", directory,
                analysisName);
            sprintf_s(analFreqFile, BUFFER_SIZE, "%s%.Freq.txt", directory,
                analysisName);
            if (fopen_s(&ofp, analysisFile, "w") != 0) // clears analysisFile
                on first run for subsequent appending
            {
                printf("\nCannot open output file <%s>.\n", analysisFile);
                perror("Error:");
            }
        }
    }
}

```

```

    return false;
}
fprintf(ofp, "Multiple_MST_analysis_results_compilation\n\nFile\
    tVolume\tNorm-Length\tLength_Norm-Std_Dev.\tLength_Norm-Mean\t");
fprintf(ofp, "Angle(Deg)_Std_Dev.\tAngle(Deg)_Mean\tJType_Std_Dev.\
    tJType_Mean");
fclose(ofp);
if (fopen_s(&ofp, analFreqFile, "w") != 0) // clears analFreqFile
{
    printf("\nCannot_open_output_file <%s>.\n", analFreqFile);
    return false;
}
fclose(ofp);
if (multiTypes)
{
    sprintf_s(analTypeFile, BUFFER_SIZE, "%s%s_Type.txt", directory,
        analysisName);
    if (fopen_s(&ofp, analTypeFile, "w") != 0) // clears analTypeFile
    {
        printf("\nCannot_open_output_file <%s>.\n", analTypeFile);
        return false;
    }
    fclose(ofp);
}
}
}
if (newAnalysis)
{
    if (fopen_s(&ofp, inFile, "r") != 0)
    {
        printf("\nCannot_open_input_file <%s>.\n", inFile);
        return false;
    }
    fclose(ofp);
}
char readType[BUFFER_SIZE];
if (newAnalysis)
    strcpy_s(readType, BUFFER_SIZE, "w");
else
    strcpy_s(readType, BUFFER_SIZE, "r");
if (fopen_s(&ofp, mstFile, readType) != 0)
{
    printf("\nCannot_open_file <%s>.\n", mstFile);
    return false;
}
fclose(ofp);
if (fopen_s(&ofp, statFile, "w") != 0)
{
    printf("\nCannot_open_output_file <%s>.\n", statFile);
    return false;
}
fclose(ofp);
if (multiAnalysis && (fopen_s(&ofp, analysisFile, "a") != 0))

```

```

{
    printf("\nCannot open output file <%s>.\n", analysisFile);
    return false;
}
fclose(ofp);
return true;
}
// Constructs the MST for the data in the supplied file and saves output
// to outFile. Returns true if construction was successful, false
// otherwise.
bool MSTcalc::ConstructMST(const char *directory, const char *sampleName,
    double gridSize, bool newAnalysis)
{
    if (!CreateFiles(directory, sampleName, newAnalysis))
        return false;
    if (!newAnalysis)
    {
        branches = new branchlist(mstFile);
        if (branches->countPts() < 1)
            return false;
    }
    else
    {
        fprintf(stdout, "Reading points from file.\n");
        pointlist* unmarked = new pointlist(inFile); // Create pointlist from
            inFile
        fprintf(stdout, "Removing duplicates.\n");
        unmarked->remdup(); // Remove duplicate points
        if (unmarked->count() < 1)
            return false;
        ConstructMSTnew(unmarked, gridSize);
        unmarked = NULL; // all points in unmarked are deleted in
            ConstructMST, so clear the remnant pointer here
        fprintf(stdout, "\nPrinting branchlist to file ...");
        branches->printall(mstFile);
    } // the MST branchlist is now stored in branches
    AnalyzeMST();
    branches->del(); // clean up
    neighborPoints.clear();
    ptlistDimensions = point();
    return true;
}
// Optimized method to calculate MST which depends on an associative grid
// array to rapidly identify potential neighboring atoms
void MSTcalc::ConstructMSTnew(pointlist *unmarked, double gridSize)
{
    clock_t startTime = clock();
    fprintf(stdout, "Creating PointGrid.\n");
    inMST.clear();
    pointlist *nonMSTpts = unmarked;
    ptlistDimensions = nonMSTpts->maxDimensions();
    int numPts = nonMSTpts->count();
    float rootnum = (float)pow(numPts, 1.0/3.0);

```



```

int ptsPerSide = (int)((float)(pow(numPts, 1.0/3.0)) + 0.99999);
double avgSpacing = (ptsPerSide - 1) / max(max(ptlistDimensions.x,
    ptlistDimensions.y), ptlistDimensions.z) * gridSize;
pointlist *inMSTpts = new pointlist(nonMSTpts);
PointGrid nonMSTGrid = PointGrid(nonMSTpts, avgSpacing); // create
    nonMSTGrid with all points in it
pointlist *firstPtcpy = new pointlist(nonMSTpts);
nonMSTGrid.RemovePoint(*nonMSTpts); // remove the first MST point
    from the grid
nonMSTpts = nonMSTpts->removeLocal(nonMSTpts); // remove first point
    from list, it will be deleted by nonMSTGrid
map<point, pointlist*> ptToBrMap;
map<point, pointlist*>::iterator ptToBrIT;
pointlist *currentPoint;
branchlist *currentBranch;
fprintf(stdout, "Beginning tree construction.\n");
inMST[inMSTpts] = true; // add first MST point to
    pointlist map
int addedPts = 1;
int progressIncrement = max(numPts / 100, 1);
branches = new branchlist(inMSTpts);
ptToBrMap[(point)*inMSTpts] = branches->start;
bool neighborsFound = nonMSTGrid.AddNearest(inMSTpts); // analyze
    points near the first MST point
while (addedPts < numPts && neighborsFound)
{
    PointGrid::Segment *mstCandidate = nonMSTGrid.PopClosest(); // pt1 is
        from inMSTpts, pt2 is from nonMSTGrid
    if (inMST.find(mstCandidate->pt2) == inMST.end())
    { // if mstCandidate is not already in the MST then it will be the
        next MST point. Must move point from nonMSTGrid to inMSTpts
        inMST[mstCandidate->pt2] = true;
        pointlist *ptCpy = inMSTpts->add(mstCandidate->pt2); // store pt to
            inMSTpts (creates new pointlist, so store this to ptCpy)
        nonMSTGrid.RemovePoint(mstCandidate->pt2);
        mstCandidate->pt2 = ptCpy; // reset reference in
            segment to the new location of the point
        addedPts++;
        if (addedPts < numPts)
            nonMSTGrid.AddNearest(ptCpy); // Add the neighbors for
                the new MST point
        currentPoint = ptToBrMap.find((point)*mstCandidate->pt1)->second;
        if (currentPoint->next == NULL) // new MST point is at end of a
            branch
        {
            ptToBrMap[(point)*mstCandidate->pt2] = currentPoint->add(
                mstCandidate->pt2);
        }
        else // new MST point is at start of new branch
        {
            currentBranch = branches->add(mstCandidate->pt1);
            ptToBrMap[(point)*mstCandidate->pt2] = currentBranch->start->add(
                mstCandidate->pt2);
        }
    }
}

```

```

    }
    if ((addedPts-1)%progressIncrement == 0) // Progress update
        fprintf(stdout, "Tree_construction_is_%d_percent_complete\n", (int)
            (((double)addedPts)/((double)numPts)*100));
    }
    // I just removed the nearest neighbor to pt1, so must add its
    // neighbors to the heap
    if (addedPts < numPts)
        neighborsFound = nonMSTGrid.AddNearest(mstCandidate->pt1);
    delete mstCandidate;
}
clock_t timeDiff = clock() - startTime;
fprintf(stderr, "Tree_construction_is_complete_in_%f_seconds.\n\n", (
    float)(timeDiff/1000.0));
fprintf(stderr, "Cleaning_up_memory...");
inMSTpts->del();
nonMSTpts->del();
inMST.clear();
}
// Perform analyses on the completed MST branchlist
void MSTcalc::AnalyzeMST()
{
    if (ptlistDimensions == point()) // default ptlistDimensions value
        indicates the dimensions have not been calculated
        ptlistDimensions = branches->maxDimensions();
    startingPoints = branches->countPts();
    bool extendedInfo = false;
    if (startingPoints > 999)
        extendedInfo = true;
    if (extendedInfo) fprintf(stdout, "\nFinding_neighbor_points...");
    FindNeighborPoints();
    if (extendedInfo) fprintf(stdout, "\nExamining_point_types...");
    GetTypes();
    if (extendedInfo) fprintf(stdout, "\nCalculating_segment_lengths...");
    AnalyzeLengths();
    if (extendedInfo) fprintf(stdout, "\nCalculating_segment_angle,
        frequency, and_junction_data...");
    AnalyzeAngles();
    fprintf(stdout, "\nMST_statistics_output_to:\n%s\n", statFile);
}
// Populate neighborPoints map from branches branchlist
void MSTcalc::FindNeighborPoints()
{
    neighborPoints.clear();
    branchlist *currentBr = branches;
    pointlist *currentPt = branches->start;
    while (currentBr != NULL)
    {
        currentPt = currentBr->start;
        while (currentPt->next != NULL)
        {
            neighborPoints[*currentPt].push_back(*currentPt->next); // add next
            point to current point neighbors
        }
    }
}

```

```

        neighborPoints[*currentPt->next].push_back(*currentPt); // add
            current point to next point neighbors
        currentPt = currentPt->next;
    }
    currentBr = currentBr->next;
}
}
// Fills the vector pointTypes with the different types of points that are
    in the MST
void MSTcalc::GetTypes()
{
    pointTypes.clear();
    multiTypes = false;
    map<int, bool> pointTypeMap;
    for (map<point, vector<point>>::const_iterator it = neighborPoints.begin()
        (); it != neighborPoints.end(); ++it)
        pointTypeMap[it->first.type] = true;
    for (map<int, bool>::const_iterator it = pointTypeMap.begin(); it !=
        pointTypeMap.end(); ++it)
        pointTypes.push_back(it->first);
    if (pointTypes.size() > 1)
        multiTypes = true;
}
// Analyze length data and append standard deviation and mean data to
    statFile
void MSTcalc::AnalyzeLengths()
{
    double* mstLengths = new double[startingPoints-1];
    map<int, vector<double>> mstLengthsAllMap;
    map<pair<int, int>, vector<double>> mstLengthsMap; // allows
        segregation of segment lengths by neighboring point types
    double currentLength = 0;
    int numlengths = 0;
    branchlist* curbranch = branches;
    pointlist* current;
    while(curbranch != NULL) // Create array of lengths
    {
        current = curbranch->start;
        while(current->next != NULL)
        {
            currentLength = current->DistanceFrom(current->next);
            mstLengths[numlengths] = currentLength;
            if (multiTypes)
            {
                int smallType = min(current->type, current->next->type);
                int bigType = max(current->type, current->next->type);
                mstLengthsMap[make_pair(smallType, bigType)].push_back(
                    currentLength);
                mstLengthsAllMap[current->type].push_back(currentLength);
                mstLengthsAllMap[current->next->type].push_back(currentLength);
            }
            numlengths++;
            current = current->next;
        }
    }
}

```

```

    }
    curbranch = curbranch->next;
}
double mean = CalcMean(mstLengths, numlengths);
double sigma = CalcStdev(mstLengths, mean, numlengths);
delete [] mstLengths;
double size, root;
char sizeString[BUFFER_SIZE];
if (ptlistDimensions.z > 0)
{
    size = (ptlistDimensions.x+mean)*(ptlistDimensions.y+mean)*(
        ptlistDimensions.z+mean);
    //size = ptlistDimensions.x * ptlistDimensions.y * ptlistDimensions.z;
    root = 1.0/3.0;
    strcpy_s(sizeString, BUFFER_SIZE, "Volume:");
}
else
{
    size = (ptlistDimensions.x+mean)*(ptlistDimensions.y+mean); // NOTE:
        Seeing what it does to not include the spacing parameter in size
    //size = ptlistDimensions.x * ptlistDimensions.y;
    root = 0.5;
    strcpy_s(sizeString, BUFFER_SIZE, "Area:");
}
double norm = pow(size/startingPoints, root);
normLength = norm;
// Store length by point type data to memory for later file output
string sampleNameStr = string(sampleName);
mnStdOnePtTypeMap[sampleNameStr];
map<string, map<int, MeanAndStdDev>>::iterator mnStdAllIT =
    mnStdOnePtTypeMap.find(sampleNameStr);
mnStdTwoPtTypeMap[sampleNameStr];
map<string, map<pair<int, int>, MeanAndStdDev>>::iterator mnStdIT =
    mnStdTwoPtTypeMap.find(sampleNameStr);
if (multiTypes) // Calculate mean and standard deviation of lengths and
    store by point type, if there are different types
{
    double tempMean, tempStd;
    for (map<int, vector<double>>::const_iterator it = mstLengthsAllMap.
        begin(); it != mstLengthsAllMap.end(); ++it)
    {
        tempMean = CalcMean(&it->second[0], it->second.size());
        tempStd = CalcStdev(&it->second[0], tempMean, it->second.size());
        mnStdAllIT->second[it->first] = MeanAndStdDev(tempMean/normLength,
            tempStd/normLength);
    }
    for (map<pair<int, int>, vector<double>>::const_iterator it =
        mstLengthsMap.begin(); it != mstLengthsMap.end(); ++it)
    {
        tempMean = CalcMean(&it->second[0], it->second.size());
        tempStd = CalcStdev(&it->second[0], tempMean, it->second.size());
        mnStdIT->second[it->first] = MeanAndStdDev(tempMean/normLength,
            tempStd/normLength);
    }
}

```



```

segmentTypes.reserve(startingPoints);
double pi = 3.1415926535897932384;
int counter = 0;
string sampleNameStr = string(sampleName);
mnStdPtTypeAngMap[sampleNameStr];
map<string, map<int, MeanAndStdDev>>::iterator mnStdAngIT =
    mnStdPtTypeAngMap.find(sampleNameStr);
map<int, vector<double>> mstAnglesMap; // store angles/junctions/
    segments separately by point type
map<int, vector<double>> jTypeByPt;
map<pair<int, int>, vector<pair<int, int>>> sTypeByPt; // store 12's,
    23's separately by AA's, BB's...
map<pair<int, int>, map<pair<int, int>, vector<double>>> sTypeLenByPt;
    // map<ptType, map<segType, vector<lengths>>>
map<int, int> numPtTypeJunct; // num A's, B's, ...
map<pair<int, int>, int> numPtTypeSeg; // num AA's, AB's, BB's, ...
map<point, bool> pointAnalyzed;
for (map<point, vector<point>>::iterator it = neighborPoints.begin(); it
    != neighborPoints.end(); it++)
{ // loop through all points in neighborPoints
    int numNeighbors = it->second.size();
    int numNeighborsLess = numNeighbors - 1;
    junctionTypes.push_back(numNeighbors);
    map<int, vector<double>>::iterator angleMapIT = mstAnglesMap.find(it->
        first.type);
    if (multiTypes)
    {
        jTypeByPt[it->first.type].push_back(numNeighbors);
        numPtTypeJunct[it->first.type]++;
        if (angleMapIT == mstAnglesMap.end())
        {
            mstAnglesMap[it->first.type];
            angleMapIT = mstAnglesMap.find(it->first.type);
        }
    }
    pointAnalyzed[it->first] = true;
    for (int i = 0; i < numNeighbors; i++)
    {
        bool newSeg = false;
        if (pointAnalyzed.find(it->second[i]) == pointAnalyzed.end())
            newSeg = true; // Avoid double counting segments
        int smallJunct, bigJunct, tempJunct, smallType, bigType;
        pair<int, int> junctPair, typePair;
        double dist;
        tempJunct = neighborPoints[it->second[i]].size();
        smallJunct = min(tempJunct, numNeighbors); // Want the smaller
            junction number to be first
        bigJunct = max(tempJunct, numNeighbors);
        if (multiTypes)
        {
            smallType = min(it->first.type, it->second[i].type);
            bigType = max(it->first.type, it->second[i].type);
            typePair = make_pair(smallType, bigType);

```

```

    if (newSeg)
        numPtTypeSeg[typePair]++;
}
dist = it->first.DistanceFrom(&it->second[i]);
junctLengthsMap[numNeighbors].push_back(dist);
if (bigJunct < 5) // Limit the segment types up to J4, so only 12,
13, 14, 22, 23, 24, 33, 34, 44 = 9 segments
{
    if (newSeg)
    {
        junctPair = pair<int, int>(smallJunct, bigJunct);
        segmentTypes.push_back(junctPair);
        segLengthsMap[junctPair].push_back(dist);
        if (multiTypes && bigJunct < 4) // Limit segment types stored by
atom type to J3, so only 5 segments
        {
            sTypeByPt[typePair].push_back(junctPair);
            sTypeLenByPt[typePair][junctPair].push_back(dist);
        }
    }
}
for (int j = i+1; j < numNeighbors; j++)
{
    angles.push_back((it->first.AngleBetween(it->second[i], it->second
[j]))*180/pi); // adds angle to vector, converts to degrees
    if (multiTypes)
        angleMapIT->second.push_back(angles.back());
}
}
}
double meanAng = CalcMean(&angles[0], angles.size());
double stdevAng = CalcStdev(&angles[0], meanAng, angles.size());
double meanJunct = CalcMean(&junctionTypes[0], junctionTypes.size());
double stdevJunct = CalcStdev(&junctionTypes[0], meanJunct,
junctionTypes.size());
map<double, MeanAndStdDev> jMnStdTemp;
for (map<double, vector<double>>::const_iterator it = junctLengthsMap.
begin(); it != junctLengthsMap.end(); it++)
{
    double mean = CalcMean(&it->second[0], it->second.size());
    double stdDev = CalcStdev(&it->second[0], mean, it->second.size());
    jMnStdTemp[it->first] = MeanAndStdDev(mean/normLength, stdDev/
normLength);
}
junctMnStdMap[sampleNameStr] = jMnStdTemp;

map<pair<int, int>, MeanAndStdDev> sMnStdTemp;
for (map<pair<int, int>, vector<double>>::const_iterator it =
segLengthsMap.begin(); it != segLengthsMap.end(); it++)
{
    double mean = CalcMean(&it->second[0], it->second.size());
    double stdDev = CalcStdev(&it->second[0], mean, it->second.size());

```

```

    sMnStdTemp[it->first] = MeanAndStdDev(mean/normLength, stdDev/
        normLength);
}
segMnStdMap[sampleNameStr] = sMnStdTemp;
map<double, double> *jMap = (CalcFreq(junctionTypes));
map<pair<int, int>, double> *sMap = (CalcFreq(segmentTypes));
junctFreqMap[sampleNameStr] = *jMap;
segFreqMap[sampleNameStr] = *sMap;
if (multiTypes)
{
    // Point type junction and segment fraction (#A's, B's, then #AA's, #
    // AB's, #BB's)
    for (map<int, int>::const_iterator it = numPtTypeJunct.begin(); it !=
        numPtTypeJunct.end(); ++it)
        junctPtTypeFraction[sampleNameStr][it->first] = it->second / (double
            )startingPoints;
    segPtTypeFraction[sampleNameStr];
    int numSegs = startingPoints - 1;
    map<string, map<pair<int, int>, double>>::iterator segPtTypeIT =
        segPtTypeFraction.find(sampleNameStr);
    for (map<pair<int, int>, int>::const_iterator it = numPtTypeSeg.begin
        ()); it != numPtTypeSeg.end(); ++it)
        segPtTypeIT->second[it->first] = it->second / (double)numSegs;
    // Angle mean and std by one point type
    for (map<int, vector<double>>::const_iterator it = mstAnglesMap.begin
        ()); it != mstAnglesMap.end(); ++it)
    {
        if (it->second.size() > 0)
        {
            double tempMean = CalcMean(&it->second[0], it->second.size());
            double tempStd = CalcStdev(&it->second[0], tempMean, it->second.
                size());
            mnStdAngIT->second[it->first] = MeanAndStdDev(tempMean, tempStd);
        }
    }
}
// Junction freq and mean and std by one point type. This is not
// junction length, but just the junction number analysis.
junctMnStdOnePtTypeMap[sampleNameStr];
map<string, map<int, MeanAndStdDev>>::iterator jMnStdPtTypeIT =
    junctMnStdOnePtTypeMap.find(sampleNameStr);
junctFreqOnePtTypeMap[sampleNameStr];
map<double, double> *jMap2;
map<string, map<int, map<double, double>>>::iterator jFreqPtTypeIT =
    junctFreqOnePtTypeMap.find(sampleNameStr);
if (multiTypes)
{
    for (map<int, vector<double>>::const_iterator it = jTypeByPt.begin();
        it != jTypeByPt.end(); ++it)
    {
        double tempMean = CalcMean(&it->second[0], it->second.size());
        double tempStd = CalcStdev(&it->second[0], tempMean, it->second.size
            ());
    }
}

```



```

    jMnStdPtTypeIT->second[it->first] = MeanAndStdDev(tempMean, tempStd)
    ;
    jMap2 = CalcFreq(it->second);
    jFreqPtTypeIT->second[it->first] = *jMap2;
}
}
// Segment freq by type
segFreqTwoPtTypeMap[sampleNameStr];
map<string, map<pair<int, int>, map<pair<int, int>, double>>>::iterator
    sFreqPtTypeIT = segFreqTwoPtTypeMap.find(sampleNameStr);
segMnStdTwoPtTypeMap[sampleNameStr];
map<string, map<pair<int, int>, map<pair<int, int>, MeanAndStdDev>>>::
    iterator sMnStdPtTypeIT = segMnStdTwoPtTypeMap.find(sampleNameStr);
map<pair<int, int>, double> *sMap2;
if (multiTypes)
{
    for (map<pair<int, int>, vector<pair<int, int>>>::const_iterator it =
        sTypeByPt.begin(); it != sTypeByPt.end(); ++it)
    {
        sMap2 = CalcFreq(it->second);
        sFreqPtTypeIT->second[it->first] = *sMap2;
    }
    for (map<pair<int, int>, map<pair<int, int>, vector<double>>>::
        const_iterator it = sTypeLenByPt.begin(); it != sTypeLenByPt.end();
        ++it)
    { // iterate through every point type
        sMnStdPtTypeIT->second[it->first];
        map<pair<int, int>, map<pair<int, int>, MeanAndStdDev>>::iterator
            sMnStdIT_2 = sMnStdPtTypeIT->second.find(it->first); // find pt
            type pair iterator
        for (map<pair<int, int>, vector<double>>::const_iterator itSeg = it
            ->second.begin(); itSeg != it->second.end(); ++itSeg)
        { // iterate through every segment type
            double tempMean = CalcMean(&itSeg->second[0], itSeg->second.size()
                );
            double tempStd = CalcStdev(&itSeg->second[0], tempMean, itSeg->
                second.size());
            sMnStdIT_2->second[itSeg->first] = MeanAndStdDev(tempMean/
                normLength, tempStd/normLength);
        }
    }
}
}
FILE* ofp;
if(fopen_s(&ofp, statFile, "a") != 0)
{
    printf("\nCannot open statistics output file <%s>.\n", statFile);
    return;
}
fprintf(ofp, "\nSegment junction angle statistics (degrees):");
fprintf(ofp, "\n\tStandard deviation:\t%f", stdevAng);
fprintf(ofp, "\n\tMean:\t\t\t%f\n\n", meanAng);
fprintf(ofp, "\nSegment junction number statistics:");
fprintf(ofp, "\n\tStandard deviation:\t%f", stdevJunct);

```



```

    for (map<pair<int, int>, double>::const_iterator it2 = it->second.
        begin(); it2 != it->second.end(); it2++)
    { // store all unique segment types from segFreqMap into segmentTypes
        segTypes[it2->first] = true;
    }
    vector<double> junctTypeVect; // this is not implemented in the first
        set of stats
    for (map<double, bool>::const_iterator it = junctTypes.begin(); it !=
        junctTypes.end(); it++)
        junctTypeVect.push_back(it->first);
    // JUNCTION TYPE
    fprintf(ofp, "\nJunction_type_frequencies:_Junction_type_number_is_
        determined_from_number_of_neighboring_points_in_the_MST)\n");
    fprintf(ofp, "File\t");
    // print junction type headings
    for (map<double, bool>::const_iterator it = junctTypes.begin(); it !=
        junctTypes.end(); it++)
        fprintf(ofp, "%d_Freq\t", (int)(it->first + 0.5)); // Junction type
        frequencies
    for (map<double, bool>::const_iterator it = junctTypes.begin(); it !=
        junctTypes.end(); it++)
        fprintf(ofp, "%d_Len_Std\t", (int)(it->first + 0.5)); // Junction
        type std dev
    for (map<double, bool>::const_iterator it = junctTypes.begin(); it !=
        junctTypes.end(); it++)
        fprintf(ofp, "%d_Len_Mean\t", (int)(it->first + 0.5)); // Junction
        type mean
    // print junction type data for all files
    for (map<string, map<double, double>::const_iterator jFileIT =
        junctFreqMap.begin(); jFileIT != junctFreqMap.end(); jFileIT++)
    { // loop through all file name statistic rows
        fprintf(ofp, "\n%s\t", jFileIT->first.c_str());
        for (map<double, bool>::const_iterator jTypeIT = junctTypes.begin();
            jTypeIT != junctTypes.end(); jTypeIT++)
        { // loop through all junction types and output frequency
            map<double, double>::const_iterator jFreqFound = jFileIT->second.
                find(jTypeIT->first);
            if (jFreqFound == jFileIT->second.end())
                fprintf(ofp, "0\t"); // item not found, output 0
            else
                fprintf(ofp, "%f\t", jFreqFound->second); // item found, output
                    frequency
        }
    }
    // J# length std dev
    for (map<double, bool>::const_iterator jTypeIT = junctTypes.begin();
        jTypeIT != junctTypes.end(); jTypeIT++)
    { // loop through all junction types and output std dev and mean
        map<double, MeanAndStdDev>::const_iterator jMnStdFound =
            junctMnStdMap[jFileIT->first].find(jTypeIT->first);
        if (jMnStdFound == junctMnStdMap[jFileIT->first].end())
            fprintf(ofp, "0\t"); // item not found, output 0
        else

```

```

        fprintf(ofp, "%f\t", jMnStdFound->second.stdDev); // item found,
            output std dev
    }
    // J# length mean
    for (map<double, bool>::const_iterator jTypeIT = junctTypes.begin();
        jTypeIT != junctTypes.end(); jTypeIT++)
    { // loop through all junction types and output std dev and mean
        map<double, MeanAndStdDev>::const_iterator jMnStdFound =
            junctMnStdMap[jFileIT->first].find(jTypeIT->first);
        if (jMnStdFound == junctMnStdMap[jFileIT->first].end())
            fprintf(ofp, "0\t"); // item not found, output 0
        else
            fprintf(ofp, "%f\t", jMnStdFound->second.mean); // item found,
                output mean
    }
}
fprintf(ofp, "\n");
// SEGMENT TYPE
fprintf(ofp, "\n\nSegment_type_frequencies:(Segment_type_is_determined_
    from_junction_type_at_both_ends_of_segment)\n\n");
fprintf(ofp, "File\t");
// print segment type headings
for (map<pair<int, int>, bool>::const_iterator it = segTypes.begin(); it
    != segTypes.end(); it++)
    fprintf(ofp, "%d_%d_Freq\t", it->first.first, it->first.second); //
        segment type frequency
for (map<pair<int, int>, bool>::const_iterator it = segTypes.begin(); it
    != segTypes.end(); it++)
    fprintf(ofp, "%d_%d_Len_Std\t", it->first.first, it->first.second);
    // segment type std dev length
for (map<pair<int, int>, bool>::const_iterator it = segTypes.begin(); it
    != segTypes.end(); it++)
    fprintf(ofp, "%d_%d_Len_Mean\t", it->first.first, it->first.second);
    // segment type mean length
// print segment type data for all file names
for (map<string, map<pair<int, int>, double>>::const_iterator sFileIT =
    segFreqMap.begin(); sFileIT != segFreqMap.end(); sFileIT++)
{ // loop through all file name statistic rows
    fprintf(ofp, "\n%s\t", sFileIT->first.c_str());
    for (map<pair<int, int>, bool>::const_iterator sTypeIT = segTypes.
        begin(); sTypeIT != segTypes.end(); sTypeIT++)
    { // loop through all segment types
        map<pair<int, int>, double>::const_iterator sFreqFound = sFileIT->
            second.find(sTypeIT->first);
        if (sFreqFound == sFileIT->second.end())
            fprintf(ofp, "0\t"); // item not found, output 0
        else
            fprintf(ofp, "%f\t", sFreqFound->second); // item found, output
                frequency
    }
}
// Segment type length std dev
for (map<pair<int, int>, bool>::const_iterator sTypeIT = segTypes.
    begin(); sTypeIT != segTypes.end(); sTypeIT++)

```

```

{ // loop through all junction types and output std dev and mean
  map<pair<int, int>, MeanAndStdDev>::const_iterator sMnStdFound =
    segMnStdMap[sFileIT->first].find(sTypeIT->first);
  if (sMnStdFound == segMnStdMap[sFileIT->first].end())
    fprintf(ofp, "0\t0\t"); // item not found, output 0
  else
    fprintf(ofp, "%f\t", sMnStdFound->second.stdDev); // item found,
    output segment length std dev
}
for (map<pair<int, int>, bool>::const_iterator sTypeIT = segTypes.
  begin(); sTypeIT != segTypes.end(); sTypeIT++)
{ // loop through all junction types and output std dev and mean
  map<pair<int, int>, MeanAndStdDev>::const_iterator sMnStdFound =
    segMnStdMap[sFileIT->first].find(sTypeIT->first);
  if (sMnStdFound == segMnStdMap[sFileIT->first].end())
    fprintf(ofp, "0\t0\t"); // item not found, output 0
  else
    fprintf(ofp, "%f\t", sMnStdFound->second.mean); // item found,
    output segment length mean
}
}
fprintf(ofp, "\n");
fclose(ofp);
if (!multiAnalysis)
  return;
// -----
// SEPARATE BY POINT TYPE
if(fopen_s(&ofp, analTypeFile, "w") != 0)
{
  printf("\nCannot open statistics output file <%s>.\n", analTypeFile);
  return;
}
// CREATE HEADING VECTORS
// -----
// Store all the different types of point in ptTypes
for (map<string, map<int, double>>::const_iterator it =
  junctPtTypeFraction.begin(); it != junctPtTypeFraction.end(); it++)
  for (map<int, double>::const_iterator it2 = it->second.begin(); it2 !=
    it->second.end(); it2++)
    { // store all unique junction types from junctFreqMap into junctTypes
      ptTypes[it2->first] = true;
    }
}
vector<int> ptTypeVect;
vector<pair<int, int>> ptPairTypeVect;

// Store different types of point pairs in ptPairTypes
for (map<int, bool>::const_iterator it = ptTypes.begin(); it != ptTypes.
  end(); ++it)
  ptTypeVect.push_back(it->first);
for (unsigned int i = 0; i < ptTypeVect.size(); ++i)
  for (unsigned int j = i; j < ptTypeVect.size(); ++j)
    ptPairTypes[make_pair(ptTypeVect[i], ptTypeVect[j])] = true;

```

```

for (map<pair<int, int>, bool>::const_iterator it = ptPairTypes.begin();
      it != ptPairTypes.end(); ++it)
    ptPairTypeVect.push_back(it->first);
// PRINT HEADINGS
// -----
// POINT TYPE AND PAIR FRACTIONS
fprintf(ofp, "\nPoint_type_frequencies:\n");
fprintf(ofp, "File\t");
// print headings for point type frequencies and mst length and angle
// data
for (unsigned int i = 0; i < ptTypeVect.size(); ++i) // Point type
    frequency heading
    fprintf(ofp, "T%d_Freq\t", ptTypeVect[i]);
for (unsigned int i = 0; i < ptPairTypeVect.size(); ++i) // Point pair
    type frequency heading
    fprintf(ofp, "T%d_%d_Freq\t", ptPairTypeVect[i].first, ptPairTypeVect[
        i].second);
for (unsigned int i = 0; i < ptTypeVect.size(); ++i) // Point type
    length std dev
    fprintf(ofp, "T%d_Len_Std\t", ptTypeVect[i]);
for (unsigned int i = 0; i < ptTypeVect.size(); ++i) // Point type
    length mean
    fprintf(ofp, "T%d_Len_Mean\t", ptTypeVect[i]);
for (unsigned int i = 0; i < ptPairTypeVect.size(); ++i) // Point pair
    type length std dev
    fprintf(ofp, "T%d_%d_Len_Std\t", ptPairTypeVect[i].first,
        ptPairTypeVect[i].second);
for (unsigned int i = 0; i < ptPairTypeVect.size(); ++i) // Point pair
    type length mean heading
    fprintf(ofp, "T%d_%d_Len_Mean\t", ptPairTypeVect[i].first,
        ptPairTypeVect[i].second);
for (unsigned int i = 0; i < ptTypeVect.size(); ++i) // Point type
    angle std
    fprintf(ofp, "T%d_Angle_Std\t", ptTypeVect[i]);
for (unsigned int i = 0; i < ptTypeVect.size(); ++i) // Point type
    angle mean
    fprintf(ofp, "T%d_Angle_Mean\t", ptTypeVect[i]);
// PRINT DATA
// -----
// POINT TYPE AND PAIR FRACTIONS
for (map<string, map<int, double>>::const_iterator fileIT =
      junctPtTypeFraction.begin(); fileIT != junctPtTypeFraction.end(); ++
      fileIT)
{ // loop through all files
    fprintf(ofp, "\n%s\t", fileIT->first.c_str());
    // Atom type frequency: T0 Freq
    map<string, map<int, double>>::iterator jTfracIT = junctPtTypeFraction
        .find(fileIT->first);
    for (unsigned int i = 0; i < ptTypeVect.size(); i++)
    { // loop through all point types
        map<int, double>::iterator foundIT = jTfracIT->second.find(
            ptTypeVect[i]); // search for point type
        if (foundIT != jTfracIT->second.end())

```

```

    fprintf(ofp, "%f\t", foundIT->second); // Found it
else
    fprintf(ofp, "0\t"); // Couldn't find it, output 0
}
// Atom-Atom type frequency: T0_1 Freq
map<string, map<pair<int, int>, double>>::iterator sTfracIT =
    segPtTypeFraction.find(fileIT->first);
for (unsigned int i = 0; i < ptPairTypeVect.size(); i++)
{ // loop through all point pair types
    map<pair<int, int>, double>::iterator foundIT = sTfracIT->second.
        find(ptPairTypeVect[i]); // search for point type
    if (foundIT != sTfracIT->second.end())
        fprintf(ofp, "%f\t", foundIT->second); // Found it
    else
        fprintf(ofp, "0\t"); // Couldn't find it, output 0
}
// Atom type length std dev: T0 Len Std
map<string, map<int, MeanAndStdDev>>::iterator mSOneIT =
    mnStdOnePtTypeMap.find(fileIT->first);
for (unsigned int i = 0; i < ptTypeVect.size(); i++)
{ // loop through all point types
    map<int, MeanAndStdDev>::iterator foundIT = mSOneIT->second.find(
        ptTypeVect[i]); // search for point type
    if (foundIT != mSOneIT->second.end())
        fprintf(ofp, "%f\t", foundIT->second.stdDev); // Found it
    else
        fprintf(ofp, "0\t"); // Couldn't find it, output 0
}
// Atom type length mean: T0 Len Mean
for (unsigned int i = 0; i < ptTypeVect.size(); i++)
{ // loop through all point types
    map<int, MeanAndStdDev>::iterator foundIT = mSOneIT->second.find(
        ptTypeVect[i]); // search for point type
    if (foundIT != mSOneIT->second.end())
        fprintf(ofp, "%f\t", foundIT->second.mean); // Found it
    else
        fprintf(ofp, "0\t"); // Couldn't find it, output 0
}
// Atom-Atom type length std dev: T0_1 Len Std
map<string, map<pair<int, int>, MeanAndStdDev>>::iterator mSTwoIT =
    mnStdTwoPtTypeMap.find(fileIT->first);
for (unsigned int i = 0; i < ptPairTypeVect.size(); i++)
{ // loop through all point types
    map<pair<int, int>, MeanAndStdDev>::iterator foundIT = mSTwoIT->
        second.find(ptPairTypeVect[i]); // search for point type
    if (foundIT != mSTwoIT->second.end())
        fprintf(ofp, "%f\t", foundIT->second.stdDev); // Found it
    else
        fprintf(ofp, "0\t"); // Couldn't find it, output 0
}
// Atom type length mean: T0_1 Len Mean
for (unsigned int i = 0; i < ptPairTypeVect.size(); i++)
{ // loop through all point types

```

```

    map<pair<int, int>, MeanAndStdDev>::iterator foundIT = mSTwoIT->
        second.find(ptPairTypeVect[i]); // search for point type
    if (foundIT != mSTwoIT->second.end())
        fprintf(ofp, "%f\t", foundIT->second.mean); // Found it
    else
        fprintf(ofp, "0\t"); // Couldn't find it, output 0
}
// Angle std dev: T0 Angle Std
map<string, map<int, MeanAndStdDev>>::iterator mSAngIT =
    mnStdPtTypeAngMap.find(fileIT->first);
for (unsigned int i = 0; i < ptTypeVect.size(); i++)
{ // loop through all point types
    map<int, MeanAndStdDev>::iterator foundIT = mSAngIT->second.find(
        ptTypeVect[i]); // search for point type
    if (foundIT != mSAngIT->second.end())
        fprintf(ofp, "%f\t", foundIT->second.stdDev); // Found it
    else
        fprintf(ofp, "0\t"); // Couldn't find it, output 0
}
// Angle mean: T0 Angle Mean
for (unsigned int i = 0; i < ptTypeVect.size(); i++)
{ // loop through all point types
    map<int, MeanAndStdDev>::iterator foundIT = mSAngIT->second.find(
        ptTypeVect[i]); // search for point type
    if (foundIT != mSAngIT->second.end())
        fprintf(ofp, "%f\t", foundIT->second.mean); // Found it
    else
        fprintf(ofp, "0\t"); // Couldn't find it, output 0
}
}
// POINT TYPE JUNCTION-SEGMENT ANALYSIS
fprintf(ofp, "\n\nPoint_type_junction_analysis:\n");
fprintf(ofp, "File\t");
// print headings for junction and segment analysis
for (unsigned int i = 0; i < ptTypeVect.size(); ++i) // Point type
    junction number std dev
    fprintf(ofp, "J%d_Std\t", ptTypeVect[i]);
for (unsigned int i = 0; i < ptTypeVect.size(); ++i)
    for (unsigned int j = 0; j < junctTypeVect.size(); ++j) // Point
        type junction number freq
        fprintf(ofp, "T%d_J%d_Freq\t", ptTypeVect[i], (int)(junctTypeVect[j]
            + 0.5));
string limitedSegs[5] = {"1_2", "1_3", "2_2", "2_3", "3_3"};
pair<int, int> limitedSegPairs[5] = {make_pair(1, 2), make_pair(1, 3),
    make_pair(2, 2), make_pair(2, 3), make_pair(3, 3)};
for (unsigned int i = 0; i < ptPairTypeVect.size(); ++i)
    for (int j = 0; j < 5; j++) // Segment frequency by point pair type
        fprintf(ofp, "T%d_%d_S%s_Freq\t", ptPairTypeVect[i].first,
            ptPairTypeVect[i].second, limitedSegs[j].c_str());
for (unsigned int i = 0; i < ptPairTypeVect.size(); ++i)
    for (int j = 0; j < 5; j++) // Segment std by point pair type
        fprintf(ofp, "T%d_%d_S%s_Std\t", ptPairTypeVect[i].first,
            ptPairTypeVect[i].second, limitedSegs[j].c_str());

```



```

for (unsigned int i = 0; i < ptPairTypeVect.size(); ++i)
    for (int j = 0; j < 5; j++) // Segment mean by point pair type
        fprintf(ofp, "T%d_%d_S%s_Mean\t", ptPairTypeVect[i].first,
            ptPairTypeVect[i].second, limitedSegs[j].c_str());
// POINT TYPE JUNCTION-SEGMENT DATA
// Mean junction number by type
for (map<string, map<int, double>>::const_iterator fileIT =
    junctPtTypeFraction.begin(); fileIT != junctPtTypeFraction.end(); ++
    fileIT)
{ // loop through all files
    fprintf(ofp, "\n%s\t", fileIT->first.c_str());
    map<string, map<int, MeanAndStdDev>>::iterator jmSOneIT =
        junctMnStdOnePtTypeMap.find(fileIT->first);
    for (unsigned int i = 0; i < ptTypeVect.size(); i++)
    { // loop through all point types
        map<int, MeanAndStdDev>::iterator foundIT = jmSOneIT->second.find(
            ptTypeVect[i]); // search for point type
        if (foundIT != jmSOneIT->second.end())
            fprintf(ofp, "%f\t", foundIT->second.stdDev); // Found it
        else
            fprintf(ofp, "0\t"); // Couldn't find it, output 0
    }
    // Atom type then junction type: T0 J1
    map<string, map<int, map<double, double>>::iterator jFOneIT =
        junctFreqOnePtTypeMap.find(fileIT->first);
    for (unsigned int i = 0; i < ptTypeVect.size(); i++)
    { // loop through all point types
        map<int, map<double, double>>::iterator foundIT = jFOneIT->second.
            find(ptTypeVect[i]); // search for point type
        if (foundIT != jFOneIT->second.end())
        {
            for (unsigned int j = 0; j < junctTypeVect.size(); ++j) // Found
                type
            { // loop through all junction types
                map<double, double>::iterator foundIT_2 = foundIT->second.find(
                    junctTypeVect[j]); // search for junction type
                if (foundIT_2 != foundIT->second.end())
                    fprintf(ofp, "%f\t", foundIT_2->second); // Found junction
                else
                    fprintf(ofp, "0\t"); // Couldn't find junction,
                        output 0
            }
        }
    }
    else
    {
        for (unsigned int j = 0; j < junctTypeVect.size(); ++j)
        { // loop through all junction types
            fprintf(ofp, "0\t"); // Couldn't find type, output
                0 for all junctions
        }
    }
}
// Atom-Atom type then segment type frequency: T0-0 S1-2 Freq

```

```

map<string , map<pair<int , int> , map<pair<int , int> , double>>>::
    iterator sFTwoIT = segFreqTwoPtTypeMap.find(fileIT->first);
for (unsigned int i = 0; i < ptPairTypeVect.size(); i++)
{ // loop through all point types
    map<pair<int , int> , map<pair<int , int> , double>>::iterator foundIT =
        sFTwoIT->second.find(ptPairTypeVect[i]); // search for point
        type
    if (foundIT != sFTwoIT->second.end())
    {
        for (unsigned int j = 0; j < 5; ++j) // Found type
        { // loop through limitedSeg segments
            map<pair<int , int> , double>::iterator foundIT_2 = foundIT->
                second.find(limitedSegPairs[j]); // search for segment
                type
            if (foundIT_2 != foundIT->second.end())
                fprintf(ofp, "%f\t", foundIT_2->second); // Found segment
            else
                fprintf(ofp, "0\t"); // Couldn't find segment,
                output 0
        }
    }
else
    {
        for (unsigned int j = 0; j < 5; ++j)
        { // loop through all junction types
            fprintf(ofp, "0\t"); // Couldn't find segment,
            output 0 for all segments
        }
    }
}
// Atom-Atom type then segment type Std: T0_0 S1_2 Std
map<string , map<pair<int , int> , map<pair<int , int> , MeanAndStdDev>>>::
    iterator smSTwoIT = segMnStdTwoPtTypeMap.find(fileIT->first);
for (unsigned int i = 0; i < ptPairTypeVect.size(); i++)
{ // loop through all point types
    map<pair<int , int> , map<pair<int , int> , MeanAndStdDev>>::iterator
        foundIT = smSTwoIT->second.find(ptPairTypeVect[i]); // search for
        point type
    if (foundIT != smSTwoIT->second.end())
    {
        for (unsigned int j = 0; j < 5; ++j) // Found type
        { // loop through limitedSeg segments
            map<pair<int , int> , MeanAndStdDev>::iterator foundIT_2 = foundIT
                ->second.find(limitedSegPairs[j]); // search for
                segment type
            if (foundIT_2 != foundIT->second.end())
                fprintf(ofp, "%f\t", foundIT_2->second.stdDev); // Found
                segment
            else
                fprintf(ofp, "0\t"); // Couldn't find segment,
                output 0
        }
    }
}

```

```

    else
    {
        for (unsigned int j = 0; j < 5; ++j)
        { // loop through all junction types
            fprintf(ofp, "0\t"); // Couldn't find segment,
                                // output 0 for all segments
        }
    }
}
// Atom-Atom type then segment type Mean: T0_0 S1_2 Mean
for (unsigned int i = 0; i < ptPairTypeVect.size(); i++)
{ // loop through all point types
    map<pair<int, int>, map<pair<int, int>, MeanAndStdDev>>::iterator
        foundIT = smSTwoIT->second.find(ptPairTypeVect[i]); // search for
        point type
    if (foundIT != smSTwoIT->second.end())
    {
        for (unsigned int j = 0; j < 5; ++j) // Found type
        { // loop through limitedSeg segments
            map<pair<int, int>, MeanAndStdDev>::iterator foundIT_2 = foundIT
                ->second.find(limitedSegPairs[j]); // search for
                segment type
            if (foundIT_2 != foundIT->second.end())
                fprintf(ofp, "%f\t", foundIT_2->second.mean); // Found segment
            else
                fprintf(ofp, "0\t"); // Couldn't find segment,
                                // output 0
        }
    }
}
else
{
    for (unsigned int j = 0; j < 5; ++j)
    { // loop through all junction types
        fprintf(ofp, "0\t"); // Couldn't find segment,
                            // output 0 for all segments
    }
}
}
}
fclose(ofp);
}
// Calculates the mean (average) of the specified number of elements in
// the supplied array starting from the first element, returns 0 if
// numElements is 0
double MSTcalc::CalcMean(const double *doubleArray, int numElements) const
{
    double sum = 0;
    for (int i = 0; i < numElements; i++)
        sum += doubleArray[i];
    if (numElements == 0)
        return 0;
    else
        return sum/numElements;
}

```

```

}
// Calculates the standard deviation of the specified number of elements
// in the supplied array given the supplied mean starting from the first
// element
double MSTcalc::CalcStdev(const double *doubleArray, double mean, int
numElements) const
{
    double sum = 0;
    double diffFromMean = 0;
    for(int i=0; i<numElements; i++)
    {
        diffFromMean = mean - doubleArray[i];
        sum += diffFromMean * diffFromMean;
    }
    if (numElements == 0)
        return 0;
    else
    {
        double sigma = pow(sum/numElements, 0.5);
        return sigma;
    }
}
template<class T>
// Calculates the frequency each element occurs in the first specified
// number of elements in the supplied array of any data type. Each unique
// element is returned in a map paired with its frequency of occurrence (
// from 0 to 1). The map is allocated on the heap.
map<T, double>* MSTcalc::CalcFreq(const vector<T> &dataArray) const
{
    map<T, int> freqMap;
    int numElements = dataArray.size();
    for (int i = 0; i < numElements; i++)
        freqMap[dataArray[i]]++;
    int freqMapCount = freqMap.size();
    map<T, double> *freqResults = new map<T, double>;
    for (map<T, int>::const_iterator it = freqMap.begin(); it != freqMap.end
        (); it++)
    {
        freqResults->insert(pair<T, double>(it->first, it->second/(double)
            numElements));
    }
    return freqResults;
}

```

branchlist

This class stores a linked list of pointers to pointlist data structures, which are in turn a linked list of points. This class also provides methods for printing all its branchlists to file, and for determining the min and max boundary points for the entire branchlist.

```

// branchlist.h
#ifndef BRANCHLIST_H
#define BRANCHLIST_H
#include <iostream>
#include "pointlist.h"
#include <map>
    pointlist;
    using namespace std;
class branchlist{
public:
    pointlist *start;
    branchlist *next;
    branchlist *last; // applies only to first branchlist in bunch
    branchlist(): start(NULL), next(NULL), last(NULL) {}
    branchlist(pointlist *P): start(new pointlist(P)), next(NULL), last(NULL)
        {}
    branchlist(char *fname);
    ~branchlist();
    branchlist* add(pointlist *newbranch);
    void printall(char *fname);
    void printall(FILE *fp);
    int count();
    int countPts();
    void del();
    point* minX();
    point* maxX();
    point* minY();
    point* maxY();
    point* minZ();
    point* maxZ();
    point maxDimensions();
    double treelength();
};

```

```

#endif

```

```

// branchlist.cpp
#include "stdafx.h"
#include "pointlist.h"
#include "branchlist.h"
#include <cstdlib>
branchlist::branchlist(char *fname): start(NULL), next(NULL), last(NULL)
{
    // File format has branch[x] where x is branch number followed by all of
    // the points in the branch, then atom type(optional), then the next
    // branch[x], etc...
    // branch[1]
    // x1 y1 z1 (type)
    // x2 y2 z2 (type)
    // . . .
    // xN yN zN (type)
    //

```

```

//  branch[2]
//  . . .
FILE *fp;
if(fopen_s(&fp, fname, "r") != 0)
{
    printf("\nCannot open input file.\n");
    exit(1);
}
bool firstLine = true;
char BUFFER[255];
double X, Y, Z;
int type, numInputs;
bool newBranch = true;
branchlist *current = this;
while(fgets(BUFFER, 255, fp) != NULL)
{
    if (BUFFER[0] == 'b')
    { // now reading branch
        newBranch = true;
        while(fgets(BUFFER, 255, fp)[0] != 10)
        {
            numInputs = sscanf_s(BUFFER, "%lf %lf %lf %d", &X, &Y, &Z, &type);
            if (numInputs < 4)
                type = 0;
            if (newBranch)
            {
                if (this->start != NULL)
                    current = add(&pointlist(X, Y, Z, type));
                else
                    start = new pointlist(X, Y, Z, type);
                newBranch = false;
            }
            else
                current->start->add(&point(X, Y, Z, type));
        }
    }
}
fclose(fp);
branchlist::~~branchlist()
{
    start->del();
}
// deletes the branchlist
void branchlist::del()
{
    branchlist* current = next;
    branchlist* nextbranch;
    while(current!=NULL)
    {
        nextbranch = current->next;
        delete current;
        current = nextbranch;
    }
}

```

```

    }
    delete this;
}
// Adds a new branch to the end of the branchlist (you must call this only
// on the first branchlist in the list to maintain proper operation),
// Should probably adapt a head node to this class for rapid branchlist
// adding, but not necessary now since branchlist's are relatively few
branchlist* branchlist::add(pointlist *P)
{
    branchlist *newbranch = new branchlist(P);
    if (last != NULL)
        last->next = newbranch;
    else
        next = newbranch;
    last = newbranch;
    return newbranch;
}
void branchlist::printall(char *fname)
{
    FILE* ofp;
    if(fopen_s(&ofp, fname, "w") != 0)
    {
        printf("\nCannot open output file <%s>.\n", fname);
        return;
    }
    printall(ofp);
    fclose(ofp);
}
void branchlist::printall(FILE *fp)
{
    branchlist* curbranch = this;
    int branchNum = 1;
    while(curbranch != NULL)
    {
        fprintf(fp, "branch[%d]\n", branchNum++);
        curbranch->start->printall(fp);
        curbranch = curbranch->next;
    }
}
point* branchlist::minX()
{
    branchlist *current = this;
    point* pt = start->minX();
    point* test;
    while (current->next != NULL)
    {
        current = current->next;
        test = current->start->minX();
        if (test->x < pt->x)
            pt = test;
    }
    return pt;
}

```

```

point* branchlist::maxX()
{
    branchlist *current = this;
    point* pt = start->maxX();
    point* test;
    while (current->next != NULL)
    {
        current = current->next;
        test = current->start->maxX();
        if (test->x > pt->x)
            pt = test;
    }
    return pt;
}
point* branchlist::minY()
{
    branchlist *current = this;
    point* pt = start->minY();
    point* test;
    while (current->next != NULL)
    {
        current = current->next;
        test = current->start->minY();
        if (test->y < pt->y)
            pt = test;
    }
    return pt;
}
point* branchlist::maxY()
{
    branchlist *current = this;
    point* pt = start->maxY();
    point* test;
    while (current->next != NULL)
    {
        current = current->next;
        test = current->start->maxY();
        if (test->y > pt->y)
            pt = test;
    }
    return pt;
}
point* branchlist::minZ()
{
    branchlist *current = this;
    point* pt = start->minZ();
    point* test;
    while (current->next != NULL)
    {
        current = current->next;
        test = current->start->minZ();
        if (test->z < pt->z)
            pt = test;
    }
}

```



```

    }
    return pt;
}
point* branchlist::maxZ()
{
    branchlist *current = this;
    point* pt = start->maxZ();
    point* test;
    while (current->next != NULL)
    {
        current = current->next;
        test = current->start->maxZ();
        if (test->z > pt->z)
            pt = test;
    }
    return pt;
}
// Return the dimensions of the smallest cube that envelopes all points in
// the branchlist
point branchlist::maxDimensions()
{
    double DX = maxX()->x - minX()->x;
    double DY = maxY()->y - minY()->y;
    double DZ = maxZ()->z - minZ()->z;
    return point(DX, DY, DZ);
}
// returns number of points in list
int branchlist::count()
{
    branchlist *current;
    current = this;
    int count=0;
    while(current!=NULL)
    {
        count++;
        current = current->next;
    }
    return(count);
}
// returns number of unique points in branchlist
int branchlist::countPts()
{
    map<point, bool> pointCount;
    branchlist *currentBranch = this;
    pointlist *currentPoint = start;
    while (currentBranch != NULL)
    {
        currentPoint = currentBranch->start;
        while (currentPoint != NULL)
        {
            pointCount[currentPoint] = true;
            currentPoint = currentPoint->next;
        }
    }
}

```

```

        currentBranch = currentBranch->next;
    }
    return pointCount.size();
}
double branchlist::treelength()
{
    double length = 0;
    branchlist *curbranch;
    pointlist *current;
    curbranch = this;
    while(curbranch != NULL)
    {
        current = curbranch->start;
        while(current->next != NULL)
        {
            length += current->DistanceFrom(current->next);
            current = current->next;
        }
        curbranch = curbranch->next;
    }
    return(length);
}

```

pointlist

This class defines a doubly linked list of points with a controlling head node structure. It also defines methods for calculating boundary points inside the pointlist.

```

// pointlist.h
// pointlist.h

#ifndef POINTLIST_H
#define POINTLIST_H
#include <stdio>
#include <map>
#include "point.h"
#include "HeadNode.h"
class HeadNode;
class pointlist : public point{
public:
    pointlist *next, *prev;
    HeadNode* headNode;
public:
    pointlist():
        next(NULL), prev(NULL), headNode(NULL) {};
    pointlist(point *P):
        point(P), next(NULL), prev(NULL), headNode(
            NULL) {};
    pointlist(double X, double Y):point(X, Y),
        next(NULL), prev(NULL),
        headNode(NULL) {};
    pointlist(double X, double Y, double Z):point(X, Y, Z),
        next(NULL),
        prev(NULL), headNode(NULL) {};

```

```

pointlist(double X, double Y, double Z, int type):point(X, Y, Z, type),
    next(NULL), prev(NULL), headNode(NULL) {};
pointlist(char *fname);
~pointlist();
pointlist* add(point *P) { return add(&pointlist(P));}
pointlist* add(pointlist *newpt);
pointlist* addLocal(pointlist *newpt);
pointlist* padd(point *P) { return padd(&pointlist(P));}
pointlist* padd(pointlist *newpt);
int count();
pointlist* remove(pointlist *Q);
pointlist* removeLocal(pointlist *Q);
void del();
int remdup();
void printall(FILE *fp);
void printall(char *fname);
pointlist* furthest(pointlist *Q);
pointlist* nearest(pointlist *Q);
pointlist* nearest();
pointlist* minX();
pointlist* maxX();
pointlist* minY();
pointlist* maxY();
pointlist* minZ();
pointlist* maxZ();
point    maxDimensions();
};
#endif

// pointlist.cpp
#include "stdafx.h"
#include "point.h"
#include "pointlist.h"
#include "cmath"
#include <cstdlib>
pointlist::pointlist(char *fname): next(NULL), prev(NULL), headNode(NULL)
{
    FILE *fp;
    if(fopen_s(&fp, fname, "r") != 0)
    {
        printf("\nCannot open input file.\n");
        exit(1);
    }
    double X, Y, Z;
    int type;
    bool threeDim = false;
    bool atoms = false; // atoms implies three dimensions, not going to
        allow atoms in two dimensions
    char tester[255];
    fgets(tester, 255, fp); // Read first line of input to determine if file
        points are two or three dimensional
    int numInputs = sscanf_s(tester, "%lf %lf %lf %d", &X, &Y, &Z, &type);
    if (numInputs == 3)

```

```

{
    threeDim = true;
    z = Z;
}
else if (numInputs == 4)
{
    threeDim = true;
    atoms = true;
    z = Z;
    this->type = type;
}
x = X;
y = Y;
if (threeDim)
{
    if (atoms)
        while( fscanf_s(fp, "%lf %lf %lf %d", &X, &Y, &Z, &type) != EOF)
        {
            add(&point(X, Y, Z, type));
        }
    else
        while( fscanf_s(fp, "%lf %lf %lf", &X, &Y, &Z) != EOF)
        {
            add(&point(X, Y, Z));
        }
}
else
    while( fscanf_s(fp, "%lf %lf", &X, &Y) != EOF)
    {
        add(&point(X, Y));
    }
fclose(fp);
}
pointlist::~~pointlist()
{}
// add point to beginning of linked list of points
pointlist* pointlist::padd(pointlist *P)
{
    if (headNode == NULL)
    { // This is the first pointlist* of a new linked list of points, so add
      a HeadNode
      headNode = new HeadNode(this);
    }
    return headNode->AddPointToBeg(P);
}
// create new point on the heap and add it to end of linked list of points
// and return a pointer to the new point
pointlist* pointlist::add(pointlist *P)
{
    if (headNode == NULL)
    { // This is the first pointlist* of a new linked list of points, so add
      a HeadNode
      headNode = new HeadNode(this);

```

```

    }
    return headNode->AddPoint(P);
}
// Add the supplied point to the pointlist without creating a new point.
// The supplied point will still be deleted if pointlist::del() is called
pointlist* pointlist::addLocal(pointlist *newpt)
{
    if (headNode == NULL)
    { // This is the first pointlist* of a new linked list of points, so add
      a HeadNode
      headNode = new HeadNode(this);
    }
    return headNode->AddPointLocal(newpt);
}
// Remove point Q from the list, delete it from the heap, and return the
// first point in the list
pointlist* pointlist::remove(pointlist *Q)
{
    pointlist *firstpt = this;
    HeadNode *headPtr = Q->headNode;
    if (headPtr != NULL)
    {
        firstpt = Q->headNode->RemovePoint(Q);
        if (headPtr->GetCount() == 0) // list is now empty, so delete headNode
        {
            firstpt = NULL;
            delete headPtr;
        }
    }
    else
    { // head node has not been created yet, so only one point in list and
      must handle deletion here
        delete Q;
        firstpt = NULL;
    }
    return firstpt;
}
pointlist* pointlist::removeLocal(pointlist *Q)
{
    pointlist *firstpt = this;
    if (Q->headNode != NULL)
    {
        firstpt = Q->headNode->RemovePointLocal(Q);
        if (Q->headNode->GetCount() == 0)
        {
            firstpt = NULL;
            delete Q->headNode;
        }
    }
    else
        firstpt = NULL;
    return firstpt;
}

```

```

// delete the pointlist
void pointlist::del()
{
    if (prev == NULL)
        delete headNode; // point is at beginning of list, so do not need
                           headNode
    pointlist *current = this->next;
    pointlist *prev = NULL;
    while(current!=NULL)
    {
        delete current->prev;
        prev = current;
        current = current->next;
    }
    delete prev;
}
// remove duplicate points
int pointlist::remdup()
{
    std::map<point, bool> *points = new std::map<point, bool>;
    pointlist *currentPoint = this;
    pointlist *nextPoint;
    point tempPt;
    int count = 0;
    while(currentPoint != NULL)
    {
        nextPoint = currentPoint->next;
        tempPt = (point)currentPoint;
        if (points->find(tempPt) == points->end()) // if no duplicate already
                                                    in list, then store point
            points->insert(std::pair<point, bool>(tempPt, true));
        else // if point is a duplicate, then delete it
        {
            currentPoint->remove(currentPoint);
            count++;
        }
        currentPoint = nextPoint;
    }
    delete points;
    return count;
}
// returns number of points in list
int pointlist::count()
{
    return this->headNode->GetCount();
}
// print list to file
void pointlist::printall(FILE *fp)
{
    pointlist *current;
    current = this;
    while(current != NULL)

```

```

    { fprintf(fp, "%.14lf\t%.14lf\t%.14lf\t%d\n", current->x, current->y,
      current->z, current->type);
      current = current->next;
    }
    fprintf(fp, "\n");
}
void pointlist::printall(char *fname)
{
    FILE *fp;
    if(fopen_s(&fp, fname, "w") != 0)
    {
        printf("\nCannot open output file <%s>.\n", fname);
        exit(1);
    }
    printall(fp);
    fclose(fp);
}
// Find furthest pt from Q in list
pointlist* pointlist::furthest(pointlist *Q)
{
    pointlist *furthest = this;
    pointlist *current = this;
    double maxdist = furthest->DistanceFromSquared(Q);
    double testdist;
    while(current != NULL)
    {
        testdist = current->DistanceFromSquared(Q);
        if(testdist > maxdist)
        {
            furthest = current;
            maxdist = testdist;
        }
        current = current->next;
    }
    return furthest;
}
// Find nearest pt to Q in list
pointlist* pointlist::nearest(pointlist *Q)
{
    pointlist *closest = this;
    pointlist *current = this->next; // start current on second point
    double mindist = this->DistanceFromSquared(Q); // set min dist as
        distance between first two points
    double testdist;
    while(current != NULL)
    {
        if (current != Q) // Don't check distance if current and Q reference
            the same point in memory
        {
            testdist = current->DistanceFromSquared(Q);
            if(testdist < mindist)
            {
                closest = current;
            }
        }
    }
}

```

```

        mindist = testdist;
    }
    current = current->next;
}
}
return closest;
}
pointlist* pointlist::nearest()
{
    return(nearest(this));
}
// find the point with the minimum X value
pointlist* pointlist::minX()
{
    double min = this->x;
    pointlist *current = this->next;
    pointlist *retpt = this;
    while(current != NULL)
    {
        if(current->x < min)
        {
            min = current->x;
            retpt = current;
        }
        current = current->next;
    }
    return(retpt);
}
// find the point with the maximum X value
pointlist* pointlist::maxX()
{
    double max = this->x;
    pointlist *current;
    current = this->next;
    pointlist *retpt;
    retpt = this;
    while(current != NULL)
    {
        if(current->x > max)
        {
            max = current->x;
            retpt = current;
        }
        current = current->next;
    }
    return(retpt);
}
// find the point with the maximum Y value
pointlist* pointlist::minY()
{
    double min = this->y;
    pointlist *current;
    current = this->next;

```



```

    pointlist *retpt;
    retpt = this;
    while(current != NULL)
    {
        if(current->y < min)
        {
            min = current->y;
            retpt = current;
        }
        current = current->next;
    }
    return(retpt);
}

// find the point with the maximum Y value
pointlist* pointlist::maxY()
{
    double max = this->y;
    pointlist *current;
    current = this->next;
    pointlist *retpt;
    retpt = this;
    while(current != NULL)
    {
        if(current->y > max)
        {
            max = current->y;
            retpt = current;
        }
        current = current->next;
    }
    return(retpt);
}

// find the point with the maximum Z value
pointlist* pointlist::minZ()
{
    double min = this->z;
    pointlist *current;
    current = this->next;
    pointlist *retpt;
    retpt = this;
    while(current != NULL)
    {
        if(current->z < min)
        {
            min = current->z;
            retpt = current;
        }
        current = current->next;
    }
    return(retpt);
}

// find the point with the maximum Z value

```

```

pointlist* pointlist::maxZ()
{
    double max = this->z;
    pointlist *current;
    current = this->next;
    pointlist *retpt;
    retpt = this;
    while(current != NULL)
    {
        if(current->z > max)
        {
            max = current->z;
            retpt = current;
        }
        current = current->next;
    }
    return(retpt);
}
// Return the dimensions of the smallest cube that envelopes all points in
// the pointlist
point pointlist::maxDimensions()
{
    double DX = maxX()->x - minX()->x;
    double DY = maxY()->y - minY()->y;
    double DZ = maxZ()->z - minZ()->z;
    return point(DX, DY, DZ);
}

```

HeadNode

This class stores pointers to the first and last points in a pointlist, and also controls adding and removing points to and from the list.

```

// HeadNode.h
#ifndef HEADNODEH
#define HEADNODEH
#include "pointlist.h"
class pointlist;
class HeadNode
{
private:
    pointlist* firstNode;
    pointlist* lastNode;
    int count;
public:
    HeadNode(pointlist* initialNode): firstNode(initialNode), lastNode(
        initialNode), count(1) {}
    ~HeadNode() {}
    pointlist* GetFirstPoint() const { return firstNode;}
    pointlist* GetLastPoint() const { return lastNode;}
}

```

```

    int GetCount() const { return count;}
    pointlist* AddPoint(pointlist* newPoint);
    pointlist* AddPointLocal(pointlist *newPoint);
    pointlist* AddPointToBeg(pointlist* newPoint);
    pointlist* RemovePoint(pointlist* oldPoint);
    pointlist* RemovePointLocal(pointlist *oldPoint);
};
#endif
// HeadNode.cpp
#include "stdafx.h"
#include "HeadNode.h"
// Creates a new point on the heap and adds it to the end of the list
pointlist* HeadNode::AddPoint(pointlist* newPoint)
{
    pointlist* newPtLst = new pointlist(newPoint);
    return AddPointLocal(newPtLst);
}
// Adds the supplied point to the end of the list without creating a new
// point. This will transfer the point if it already belongs to another
// list
pointlist* HeadNode::AddPointLocal(pointlist *newPoint)
{
    if (newPoint->headNode != NULL) // if point belongs to a list then
        remove it from that list without deleting it
        newPoint->headNode->RemovePointLocal(newPoint);
    newPoint->headNode = this;
    newPoint->prev = lastNode;
    lastNode->next = newPoint;
    lastNode = newPoint;
    count++;
    return newPoint;
}
pointlist* HeadNode::AddPointToBeg(pointlist* newPoint)
{
    pointlist* newPtLst = new pointlist(newPoint);
    newPtLst->headNode = this;
    newPtLst->next = firstNode;
    firstNode->prev = newPtLst;
    firstNode = newPtLst;
    count++;
    return newPtLst;
}
// Removes the supplied point, deletes it from the heap, and returns the
// first point in the list
pointlist* HeadNode::RemovePoint(pointlist* oldPoint)
{
    pointlist *newFirst = RemovePointLocal(oldPoint);
    delete oldPoint;
    return newFirst;
}
// Removes the supplied point from the list without deleting it and
// returns the first point in the list
pointlist* HeadNode::RemovePointLocal(pointlist *oldPoint)

```

```

{
    if (oldPoint->prev != NULL)
        oldPoint->prev->next = oldPoint->next;
    else
        this->firstNode = oldPoint->next;
    if (oldPoint->next != NULL)
        oldPoint->next->prev = oldPoint->prev;
    else
        this->lastNode = oldPoint->prev;
    count--;
    return this->firstNode;
}

```

point

This class defines an object with four data fields: three doubles for location specification, and one int for type specification. Operator overloads are defined that control adding, subtracting, and taking the dot product of two points, and methods are defined for the calculation of the distances and angles between points.

```

// point.h
#ifndef POINT_H
#define POINT_H
#include <cmath>
class point{
public:
    double x, y, z;
    int type;
public:
    point(): x(0), y(0), z(0), type(0) {};
    point(double X, double Y): x(X), y(Y), z(0), type(0) {};
    point(double X, double Y, double Z): x(X), y(Y), z(Z), type(0) {};
    point(double X, double Y, double Z, int ptType): x(X), y(Y), z(Z), type(ptType) {};
    point(point *Q): x(Q->x), y(Q->y), z(Q->z), type(Q->type) {};
    ~point() {};
    bool EqualsValue(point* otherPt) const { return ((this->x == otherPt->x)
        && (this->y == otherPt->y) && (this->z == otherPt->z)); }
    void set(point* pt) { x = pt->x; y = pt->y; z = pt->z; }
    void set(double X, double Y) { x = X; y = Y; }
    void set(double X, double Y, double Z) { x = X; y = Y; z = Z; }
    double DistanceFromSquared(const point* otherPoint) const;
    double DistanceFrom(point* otherPoint) const;
    point operator-(const point &Q) const { return point(x-Q.x, y-Q.y, z-Q.z); }
};

```

```

    double operator*(const point &Q) const { return  x*Q.x + y*Q.y + z*Q.z;}
        // dot product
    point operator*(double scalar) const { return point(x*scalar, y*scalar,
        z*scalar); }    bool operator<(const point &otherPoint) const;
    bool operator==(const point &otherPoint) const { if (x == otherPoint.x
        && y == otherPoint.y && z == otherPoint.z) return true; else return
        false; }
    double AngleBetween(const point &pointOne, const point &pointTwo) const;
};
#endif

// point.cpp
#include "stdafx.h"
#include "cmath"
#include "point.h"
#include "pointlist.h"
double point::DistanceFromSquared(const point *otherPoint) const
{
    double xDiff = otherPoint->x - x;
    double yDiff = otherPoint->y - y;
    double zDiff = otherPoint->z - z;
    return xDiff * xDiff + yDiff * yDiff + zDiff * zDiff;
}
double point::DistanceFrom(point *otherPoint) const
{
    return sqrt(DistanceFromSquared(otherPoint)); // declare inline?
}
// Included so point can be a valid key in map class
bool point::operator <(const point &otherPoint) const
{ // z is major digit, y middle, x minor digit
    if (z < otherPoint.z)
        return true;
    if (z > otherPoint.z)
        return false;
    if (y < otherPoint.y)
        return true;
    if (y > otherPoint.y)
        return false;
    if (x < otherPoint.x)
        return true;
    return false;
}
// Calculates the angle in radians between the lines formed from the
// current point to each of the two provided points.
double point::AngleBetween(const point &A, const point &B) const
{ // c.AngleBetween(a, b) -> ac = a - c, bc = b - c, theta = acos(ac*ab/(|
    a|*|b|))
    point vectA = A - *this;
    point vectB = B - *this;
    double magA = point(0, 0, 0).DistanceFrom(&vectA);
    double magB = point(0, 0, 0).DistanceFrom(&vectB);
    double product = vectA * vectB / (magA * magB);

```

```

    if (product > 1) // Ensure product between -1 and 1, rounding errors
        can push it outside the bounds
        product = 1;
    else if (product < -1)
        product = -1;
    double theta = acos(product);
    return theta;
}

```

point_I

This class is functionally identical to the point class, except its location specifiers are int types and it does not have a type specifier.

```

// point_I.h
#ifndef POINT_I_H
#define POINT_I_H
#include <cmath>
#include "point.h"
class point_I{
public:
    int x, y, z;
public:
    point_I(): x(0), y(0), z(0) {};
    point_I(int X, int Y): x(X), y(Y), z(0) {};
    point_I(int X, int Y, int Z): x(X), y(Y), z(Z) {};
    point_I(point_I *Q): x(Q->x), y(Q->y), z(Q->z) {};
    ~point_I() {};
    bool EqualsValue(point_I* otherPt) const { return ((this->x == otherPt->
        x) && (this->y == otherPt->y) && (this->z == otherPt->z)); }
    double DistanceFromSquared(const point_I* otherPoint) const;
    double DistanceFrom(point_I* otherPoint) const;
    point_I operator-(const point_I &Q) const { return point_I(x-Q.x, y-Q.y,
        z-Q.z); }
    int operator*(const point_I &Q) const { return x*Q.x + y*Q.y + z*Q.z;}
        // dot product
    point operator*(double scalar) const { return point (x*scalar, y*scalar
        , z*scalar); } point_I operator*(int scalar) const { return
        point_I(x*scalar, y*scalar, z*scalar); } bool operator<(const
        point_I &otherPoint) const;
    operator point const () { return point(x, y, z); } // implicit
        conversion to point
    double AngleBetween(const point_I &point_IOne, const point_I &point_ITwo
        ) const;
};
#endif

// point_I.cpp
#include "stdafx.h"
#include "cmath"

```

```

#include "point_I.h"
#include "pointlist.h"
double point_I::DistanceFromSquared(const point_I *otherPoint) const
{
    double xDiff = otherPoint->x - x;
    double yDiff = otherPoint->y - y;
    double zDiff = otherPoint->z - z;
    return xDiff * xDiff + yDiff * yDiff + zDiff * zDiff;
}
double point_I::DistanceFrom(point_I *otherPoint) const
{
    return sqrt(DistanceFromSquared(otherPoint)); // declare inline?
}
// Included so point_I can be a valid key in map class
bool point_I::operator <(const point_I &otherPoint) const
{ // z is major digit, y middle, x minor digit
    if (z < otherPoint.z)
        return true;
    if (z > otherPoint.z)
        return false;
    if (y < otherPoint.y)
        return true;
    if (y > otherPoint.y)
        return false;
    if (x < otherPoint.x)
        return true;
    return false;
}
// Calculates the angle in radians between the lines formed from the
// current point_I to each of the two provided point_Is.
double point_I::AngleBetween(const point_I &A, const point_I &B) const
{ // c.AngleBetween(a, b) -> ac = a - c, bc = b - c, theta = acos(ac*ab/(|
    a|*|b|))
    point_I vectA = A - *this;
    point_I vectB = B - *this;
    double magA = point_I(0, 0, 0).DistanceFrom(&vectA);
    double magB = point_I(0, 0, 0).DistanceFrom(&vectB);
    double product = vectA * vectB / (magA * magB);
    if (product > 1) // Ensure product between -1 and 1, rounding errors
        can push it outside the bounds
        product = 1;
    else if (product < -1)
        product = -1;
    double theta = acos(product);
    return theta;
}

```

BinaryHeap

This class is a standard minimum binary heap class which optimizes the extraction of the smallest object from a large data set. As elements are added to the heap, it stores them in a binary tree structure, sorting by size, and the top item of the heap is always maintained as the item of smallest value in the entire heap. This allows for rapid retrieval of the smallest object.

```
// BinaryHeap.h
#ifndef BINARY_HEAP_H
#define BINARY_HEAP_H
#include <vector>
using namespace std;
template<class T>
class BinaryHeap
{
private:
    vector<int> myints;
    vector<T*> heapItems;
    void BubbleDown (int index);
    void BubbleUp (int index);
    void Swap(int index1, int index2) { T* temp = heapItems[index1];
        heapItems[index1] = heapItems[index2]; heapItems[index2] = temp; }
    int ChildOne(int index) { return index * 2 + 1; }
    int ChildTwo(int index) { return index * 2 + 2; }
    int Parent(int index) { return (index - 1) / 2; }
public:
    BinaryHeap(void) {}
    ~BinaryHeap(void) { for (int i = 0; i < (int)heapItems.size(); i++)
        delete heapItems[i]; }
    T* PopHeap();
    T* PeekHeap() { return heapItems[0]; }
    void PushHeap(const T &itemToAdd);
    void Clear() { for (int i = 0; i < (int)heapItems.size(); i++) delete
        heapItems[i]; }
    int GetNumItems() const { return heapItems.size(); }
};
#include "BinaryHeap.cpp"
#endif

// BinaryHeap.cpp
#include "StdAfx.h"
#include "BinaryHeap.h"
#ifndef BINARY_HEAP_CPP
#define BINARY_HEAP_CPP
// The item at index may be too high in the tree
template<class T>
void BinaryHeap<T>::BubbleDown(int index)
{
    int childOne = ChildOne(index);
```



```

int childTwo = ChildTwo(index);
if ( childOne < (int)heapItems.size()
    && *heapItems[index] > *heapItems[childOne] )
{
    if ( childTwo < (int)heapItems.size()
        && *heapItems[childTwo] < *heapItems[childOne])
    { // Swap larger parent with smallest child (for minheap)
      Swap(index, childTwo);
      BubbleDown(childTwo);
    }
    else
    {
      Swap(index, childOne);
      BubbleDown(childOne);
    }
}
else if ( childTwo < (int)heapItems.size()
    && *heapItems[index] > *heapItems[childTwo])
{
    Swap(index, childTwo);
    BubbleDown(childTwo);
}
}
// The item at index may be too low in the tree
template<class T>
void BinaryHeap<T>::BubbleUp(int index)
{
    if (index > 0)
    {
        int parent = Parent(index);
        if (*heapItems[index] < *heapItems[parent])
        { // Swap smaller child with larger parent
          Swap(index, parent);
          BubbleUp(parent);
        }
    }
}
// Extract top item from heap and return a pointer to it (Item will be
   stored on free store, remember to delete it when done with it)
template<class T>
T* BinaryHeap<T>::PopHeap()
{
    T* returnItem = heapItems[0];
    heapItems[0] = heapItems.back();
    heapItems.pop_back();
    BubbleDown(0);
    return returnItem;
}
// Add item to end of heap
template<class T>
inline void BinaryHeap<T>::PushHeap(const T &itemToAdd)
{
    heapItems.push_back(new T(itemToAdd));
}

```

```

    BubbleUp(heapItems.size() - 1);
}
#endif

```

PointGrid

This class performs the bulk of the work in calculating the MST. When a pointlist is added to the PointGrid, it stores the points in a grid, which is basically an array of pointlists which represents the points present in each grid box. Grid boxes can be rapidly accessed based on point location, which allows for rapid retrieval of points in the PointGrid based on point location, and subsequent fast determination of neighboring points.

```

// PointGrid.h
#ifndef POINTGRID_H
#define POINTGRID_H
#include <map>
#include "pointlist.h"
#include "BinaryHeap.h"
#include "point_I.h"
#include <limits>
class pointlist;
class point;
using namespace std;
// Stores a set of points in a grid which allows for fast retrieval of
// points near a specified location
class PointGrid
{
public: struct Segment;
private:
    struct GridDimensions
    {
        double gridSide;
        point fullGridSize;
        point_I numGridBoxes;
        point bottomLeft;
    };
    struct NeighborData
    {
        BinaryHeap<Segment> segmentHeap;
        int largestNearestNeighbor;
        NeighborData(void): largestNearestNeighbor(0) {}
    };
    pointlist** grid;
    GridDimensions gridDim;
    BinaryHeap<Segment> segmentHeap;
    map<pointlist*, NeighborData> neighborDataMap;

```

```

map<pointlist*, bool> deadPoints; // after removing points from grid
    there will still be many references to these points in the
    segmentHeap, so must store the point locations to allow later access
    and eventual deletion
double CalcDistances(pointlist *pt, pointlist *gridPoints);
double CalcDistances(pointlist *pt, const vector<pointlist*> &pointLists
);
int GetGridIndex(int xIndex, int yIndex, int zIndex) const { return
    zIndex * gridDim.numGridBoxes.y * gridDim.numGridBoxes.x + yIndex *
    gridDim.numGridBoxes.x + xIndex; }
int GetGridIndex(const point_I &xyzIndices) const { return GetGridIndex(
    xyzIndices.x, xyzIndices.y, xyzIndices.z); }
void RemovePoint(pointlist *ptlist, int index);
public:
    PointGrid(void) {}
    PointGrid(pointlist* points, double sizeOfGridSide);
    PointGrid(pointlist* points, PointGrid sampleGrid);
    ~PointGrid(void);
    // Stores pointers to two pointlist members and the distance between
        them, which is used to sort Segment variables. Does not handle data
        storage, only pointers are retained.
    struct Segment
    {
        double lengthSqr;
        pointlist *pt1;
        pointlist *pt2;
        Segment(double segLengthSqr, pointlist *firstPt, pointlist *secondPt):
            lengthSqr(segLengthSqr), pt1(firstPt), pt2(secondPt) {}
        ~Segment(void) {}
        bool operator<(const Segment &otherSegment) const { if (lengthSqr <
            otherSegment.lengthSqr) return true; else return false; }
        bool operator>(const Segment &otherSegment) const { if (lengthSqr >
            otherSegment.lengthSqr) return true; else return false; }
    };
    point_I GetGridBox(const point* pt) const { return ( point_I( (int)((
        pt->x - gridDim.bottomLeft.x)/gridDim.gridSide), (int)((pt->y -
        gridDim.bottomLeft.y)/gridDim.gridSide), (int)((pt->z - gridDim.
        bottomLeft.z)/gridDim.gridSide) )); }

    void GetNeighboringGridBoxes(const point_I &gridBox, int
        nearestNeighborNumber, vector<pointlist*> *neighborPointLists) const;
    bool AddNearest(pointlist *pt);
    Segment* PopClosest();
    void ClearNeighbors() { segmentHeap.Clear(); }
    void RemovePoint(pointlist* ptList);
    void RemovePoint(point &pt);
};
#endif

// PointGrid.cpp
#include "StdAfx.h"
#include "PointGrid.h"

```

```

// Make sure all points that define grid boundaries are supplied to this
// constructor
PointGrid::PointGrid(pointlist *points, double sizeOfGridSide)
{
    point minX = points->minX();
    point minY = points->minY();
    point minZ = points->minZ();
    point maxX = points->maxX();
    point maxY = points->maxY();
    point maxZ = points->maxZ();
    point_I numBoxes = point_I( (int)((maxX.x - minX.x)/sizeOfGridSide +
        1.5), (int)((maxY.y - minY.y)/sizeOfGridSide + 1.5), (int)((maxZ.z
        - minZ.z)/sizeOfGridSide + 1.5) );
    gridDim.bottomLeft = point(minX.x - sizeOfGridSide / 2, minY.y -
        sizeOfGridSide / 2, minZ.z - sizeOfGridSide / 2);
    gridDim.fullGridSize = numBoxes * sizeOfGridSide;
    gridDim.gridSide = sizeOfGridSide;
    gridDim.numGridBoxes = numBoxes;
    int totalNumBoxes = numBoxes.x * numBoxes.y * numBoxes.z;
    grid = new pointlist*[totalNumBoxes];
    for (int i = 0; i < totalNumBoxes; i++)
        grid[i] = NULL;
    int numPoints = points->count();
    pointlist* currentPoint = points;
    while (currentPoint != NULL)
    { // Populate grid with points in supplied pointlist
        int gridIndex = GetGridIndex( GetGridBox(currentPoint) );
        if (grid[gridIndex] == NULL)
        { // create new pointlist
            grid[gridIndex] = new pointlist(currentPoint);
        }
        else
        { // add to existing pointlist
            grid[gridIndex]->add(&point(currentPoint));
        }
        currentPoint = currentPoint->next;
    }
}

// Constructs a PointGrid containing the supplied points and with the
// capacity as specified in the sample grid supplied
PointGrid::PointGrid(pointlist *points, PointGrid sampleGrid)
{
    gridDim = sampleGrid.gridDim;
    pointlist* currentPoint = points;
    while (currentPoint != NULL)
    {
        int gridIndex = GetGridIndex( GetGridBox(currentPoint) );
        grid[gridIndex] = currentPoint;
    }
}

// Destructor: clean up memory
PointGrid::~~PointGrid()
{

```

```

int numBoxes = gridDim.numGridBoxes.x * gridDim.numGridBoxes.y * gridDim
    .numGridBoxes.z;
for (int i = 0; i < numBoxes; i++)
    delete grid[i];
delete [] grid;
for (map<pointlist*, bool>::iterator it = deadPoints.begin(); it !=
    deadPoints.end(); ++it)
    delete it->first;
}
// Adds all pointlists contained within the specified nearest neighbor
// shell of grid boxes to the specified map
void PointGrid::GetNeighboringGridBoxes(const point_I &gridBox, int
    nearestNeighborNumber, vector<pointlist*> *neighborPointLists) const
{
    vector<point_I> neighborBoxes;
    if (nearestNeighborNumber == 0)
    { // stores the gridBox itself
        neighborBoxes.push_back(point_I(gridBox));
    }
    else
    {
        point_I minPoint = point_I(gridBox.x - nearestNeighborNumber, gridBox.
            y - nearestNeighborNumber, gridBox.z - nearestNeighborNumber);
        point_I maxPoint = point_I(gridBox.x + nearestNeighborNumber, gridBox.
            y + nearestNeighborNumber, gridBox.z + nearestNeighborNumber);
        if (minPoint.z >= 0 || maxPoint.z < gridDim.numGridBoxes.z)
            for (int y = (int)minPoint.y; y <= maxPoint.y; y++) // add in top
                and bottom of shell
                if (y >= 0 && y < gridDim.numGridBoxes.y)
                    for (int x = (int)minPoint.x; x <= maxPoint.x; x++)
                        {
                            if (x >= 0 && x < gridDim.numGridBoxes.x)
                                {
                                    if (minPoint.z >= 0)
                                        neighborBoxes.push_back(point_I(x, y, minPoint.z));
                                    if (maxPoint.z < gridDim.numGridBoxes.z)
                                        neighborBoxes.push_back(point_I(x, y, maxPoint.z));
                                }
                        }
        int zMaxMinus1 = (int)maxPoint.z - 1;
        int yMaxMinus1 = (int)maxPoint.y - 1;
        for (int z = (int)minPoint.z + 1; z <= zMaxMinus1; z++)
        {
            if (z >= 0 && z < gridDim.numGridBoxes.z)
            {
                for (int x = (int)minPoint.x; x <= maxPoint.x; x++) // add front
                    and back of shell minus those in top and bottom
                    {
                        if (x >= 0 && x < gridDim.numGridBoxes.x)
                            {
                                if (minPoint.y >= 0)
                                    neighborBoxes.push_back(point_I(x, minPoint.y, z));
                                if (maxPoint.y < gridDim.numGridBoxes.y)

```

```

        neighborBoxes.push_back(point_I(x, maxPoint.y, z));
    }
}
for (int y = (int)minPoint.y + 1; y <= yMaxMinus1; y++) // add
    right and left of shell minus those in top/bot and front/back
{
    if (y >= 0 && y < gridDim.numGridBoxes.y)
    {
        if (minPoint.x >= 0)
            neighborBoxes.push_back(point_I(minPoint.x, y, z));
        if (maxPoint.x < gridDim.numGridBoxes.x)
            neighborBoxes.push_back(point_I(maxPoint.x, y, z));
    }
}
} // next z
} // Else
// Now check for corresponding pointlist structures in each of the grid
// boxes held in neighborBoxes and return them
int numBoxes = neighborBoxes.size();
for (int i = 0; i < numBoxes; i++)
{
    int gridIndex = GetGridIndex(neighborBoxes[i]);
    if (grid[gridIndex] != NULL)
    {
        neighborPointLists->push_back(grid[gridIndex]);
    }
}
}
// Analyzes area around the specified point and adds the segments between
// the point and the pointgrid to segmentHeap. The minimum segment
// distance in the heap will be the nearest neighbor to the specified
// point. Saves analysis data to optimize future searching for the same
// pointlist* (so keep the same pointlist pointers when calling this
// routine). If no neighbor points are found, this function returns false
.
bool PointGrid::AddNearest(pointlist *pt)
{
    point_I gridBox = GetGridBox(pt);
    bool foundPoint = false;
    double minDistSqr;
    int neighborNum = 0;
    vector<pointlist*> *neighborPointLists = new vector<pointlist*>;
    map<pointlist*, NeighborData>::iterator neighborDataIT = neighborDataMap
        .find(pt);
    if (neighborDataIT == neighborDataMap.end())
        neighborDataIT = neighborDataMap.insert(pair<pointlist*, NeighborData>
            >(pt, NeighborData())).first;
    if (neighborDataIT->second.segmentHeap.GetNumItems() == 0)
    { // Find the nearest neighbor point in the smallest neighbor shell
        number (not necessarily the nearest point overall)
        double maxGridDim = gridDim.numGridBoxes.x > gridDim.numGridBoxes.y ?
            gridDim.numGridBoxes.x : gridDim.numGridBoxes.y;

```

```

maxGridDim = maxGridDim > gridDim.numGridBoxes.z ? maxGridDim :
    gridDim.numGridBoxes.z; // prevents infinite loop upon failure
while (!foundPoint && neighborNum < maxGridDim)
{ // Find nearest point by searching nearest neighbor shells of
  gridboxes (using shells in cube shape around gridBox)
  GetNeighboringGridBoxes(gridBox, neighborNum++, neighborPointLists);
  if (neighborPointLists->size() != 0)
  {
    foundPoint = true;
    minDistSqr = CalcDistances(pt, *neighborPointLists);
  }
}
neighborDataIT->second.largestNearestNeighbor = neighborNum - 1;
if (neighborNum > maxGridDim)
{
  delete neighborPointLists;
  return false;
}
}
else
{ // already have at least one neighbor point stored in the segmentHeap,
  so just set the minDistSqr
  minDistSqr = neighborDataIT->second.segmentHeap.PeekHeap()->lengthSqr;
  neighborNum = neighborDataIT->second.largestNearestNeighbor + 1;
}
neighborPointLists->clear();
// Found point using nearby grid method, now need to check all grids
within the radius of the min distance found
int maxNeighbor = (int)(sqrt(minDistSqr) / gridDim.gridSide + 1);
for (int i = neighborNum; i <= maxNeighbor; i++)
{ // Search remaining gridBox shells within the distance of the first
  point found
  GetNeighboringGridBoxes(gridBox, i, neighborPointLists);
}
neighborDataIT->second.largestNearestNeighbor = maxNeighbor;
if (neighborPointLists->size() != 0)
  CalcDistances(pt, *neighborPointLists);
delete neighborPointLists;
return true;
}
// Calculates the square of the distances between the supplied pt and all
points in the pointlist and stores the distances in the
binary heap segmentHeap. Returns the shortest distance squared
encountered.
double PointGrid::CalcDistances(pointlist *pt, pointlist *gridPoints)
{
  pointlist *currentPoint = gridPoints;
  double minDistSqr = numeric_limits<double>::max();
  map<pointlist*, NeighborData>::iterator it = neighborDataMap.find(pt);
  if (currentPoint != NULL && it == neighborDataMap.end())
    it = neighborDataMap.insert(pair<pointlist*, NeighborData>(pt,
      NeighborData())).first;
  while (currentPoint != NULL)

```

```

{
    Segment newSegment(pt->DistanceFromSquared(currentPoint), pt,
        currentPoint);
    if (newSegment.lengthSqr < minDistSqr)
        minDistSqr = newSegment.lengthSqr;
    segmentHeap.PushHeap(newSegment);
    it->second.segmentHeap.PushHeap(newSegment);
    currentPoint = currentPoint->next;
}
return minDistSqr;
}
// Calculates the square of the distances between the supplied pt and all
// pointlists in the supplied vector and stores the distances in the
// binary heap segmentHeap. Returns the shortest distance squared
// encountered
double PointGrid::CalcDistances(pointlist *pt, const vector<pointlist*> &
pointLists)
{
    double minDistSqr = numeric_limits<double>::max();
    int numPts = pointLists.size();
    for (int i = 0; i < numPts; i++)
    {
        double tempMinDistSqr = CalcDistances(pt, pointLists[i]);
        if (tempMinDistSqr < minDistSqr)
            minDistSqr = tempMinDistSqr;
    }
    return minDistSqr;
}
// Returns the shortest segment from the heap as found through AddNearest.
// pt1 is in the MST while pt2 is from the pointgrid. Remember to delete
// the segment when finished
PointGrid::Segment* PointGrid::PopClosest()
{
    delete neighborDataMap[segmentHeap.PeekHeap()->pt1].segmentHeap.PopHeap
        (); // keep the individual and main segment heaps in sync
    return segmentHeap.PopHeap();
}
// Removes the point from its pointlist and clears its reference in the
// grid. Assumes point is on heap and deletes it upon destruction of the
// PointGrid.
// Supplied pointlist must be in the PointGrid, otherwise erratic behavior
// may ensue.
void PointGrid::RemovePoint(pointlist *ptList)
{
    int index = GetGridIndex( GetGridBox(ptList) );
    RemovePoint(ptList, index);
}
// Searches for and finds the proper point then removes it from its
// pointlist. Assumes point is on heap and deletes it upon destruction of
// the PointGrid.
void PointGrid::RemovePoint(point &pt)
{
    int index = GetGridIndex( GetGridBox(&pt) );

```



```

pointlist *currentPoint = grid[index];
pointlist *foundPt = NULL;
while (currentPoint != NULL)
{
    if (currentPoint->EqualsValue(&pt))
        foundPt = currentPoint;
    currentPoint = currentPoint->next;
}
RemovePoint(foundPt, index);
}
// Local method that actually performs point removal
void PointGrid::RemovePoint(pointlist *ptList, int index)
{
    grid[index] = grid[index]->removeLocal(ptList); // This will NULL grid[
        index] if necessary
    deadPoints[ptList] = true; // Store point location for later
        deletion
}

```

MST_NameMaker_CL

This is a command line program that takes two inputs: a directory name, and an optional second parameter “output” that if present specifies whether to provide program output to the command line. The program creates a plain text file named “MST_FILENAMES.txt” that contains a list of all *.txt* in the specified directory. This program is called within the user interface portion of the MST calculation program in order to automate calculation of all potential MST files in a directory.

This code is programmed in C# and compiled for the .NET framework version 3.5 using Microsoft Visual Studio 2008.

```

// MST_NameMaker_CL.cs

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.IO;

namespace MST_NameMaker_CL
{
    class Program
    {
        // C:\pathname output
        static void Main(string[] args)
        {

```

```

    if (args.Length > 0)
    {
        bool output = false;
        if (args.Length > 1 && args[1] == "output")
            output = true;
        string dirName = args[0];
        if (output)
            Console.WriteLine("Received:_" + dirName);
        if (!Directory.Exists(dirName))
        {
            if (output)
                Console.WriteLine("Directory_does_not_exist!");
            return;
        }
        DirectoryInfo info = new DirectoryInfo(dirName);
        FileInfo[] files = info.GetFiles();
        StreamWriter writer = null;
        try
        {
            writer = new StreamWriter(dirName + "\\MST.FILENAMES.
            txt");
            for (int i = 0; i < files.Length; i++)
            {
                if (files[i].Name.EndsWith(".txt"))
                {
                    string fileLine = files[i].Name.Substring(0,
                        files[i].Name.Length - 4);
                    writer.WriteLine(fileLine);
                }
            }
            if (output)
                Console.WriteLine("MST.FILENAMES_created_
                successfully.");
        }
        catch (Exception ex)
        {
            if (output)
                Console.WriteLine(ex.ToString());
        }
        finally
        {
            if (writer != null)
                writer.Close();
        }
    }
}
}
}

```

Image Analysis Code

The following code was used to perform a large number of image analysis calculations on the Al-Si system. Primary methods include the radial distribution and critical diameter calculations described earlier. Additionally, methods are provided for the calculation of phase fraction, random line spacing as described in the Al-Si Flake to Fiber Transition section, and for the outlining of objects, which is also used in calculation of the radial distribution function. The class is designed for use in a multithreaded environment and provides handlers for events that are raised when calculations are finished. Analysis settings are passed in using the AnalysisOptions class and results are returned through data classes passed by events raised upon finishing calculation. All image calculation methods are encapsulated in the class ImageCalc.

The class ImageCalc was programmed in C# and compiled for the .NET Framework version 3.5 using Microsoft Visual Studio 2008. A windows forms user interface class was created for user interaction with the ImageCalc class, however, that code is not concerned with the actual image analysis, and is therefore not presented here.

ImageCalc

This class provides a large number of highly optimized image analysis methods.

```
// ImageCalc.cs
using System;
using System.Drawing;
using System.Drawing.Drawing2D;
using System.Drawing.Imaging;
using System.Collections;
using System.Collections.Generic;
using System.Runtime.InteropServices;
using System.Threading;
namespace Image_Analyzer
{
    public unsafe class ImageCalc
    {
        private ObjectList theObjects;
        private bool stopCalculations = false;
        private bool calculationInProgress = false;
        public delegate void LockAnalyzeMenuHandler(object sender, System.
            EventArgs args);
```

```

    public event LockAnalyzeMenuHandler OnLockAnalyzeMenu;
    public delegate void UnLockAnalyzeMenuHandler(object sender,
        System.EventArgs args);
    public event UnLockAnalyzeMenuHandler OnUnLockAnalyzeMenu;
    public delegate void ProgressUpdateHandler(object sender, double
        percentIncrement);
    public event ProgressUpdateHandler OnProgressUpdate;
    public delegate void AfterSpacingCalcHandler(object sender,
        SpacingCalcInfo results);
    public event AfterSpacingCalcHandler AfterSpacingCalc;
    public delegate void AfterPhaseCalcHandler(object sender,
        PhaseFractionResults results);
    public event AfterPhaseCalcHandler AfterPhaseCalc;
    public delegate void ObjectsFoundHandler(object sender,
        FindObjectResults results);
    public event ObjectsFoundHandler OnObjectsFound;
    // The following events used to call back in MainForm
    public delegate void FinishedFindObjectHandler(object sender,
        System.EventArgs args);
    public event FinishedFindObjectHandler OnFinishedFindObject;
    public delegate void FinishedSpacingCalcHandler(object sender,
        System.EventArgs args);
    public event FinishedSpacingCalcHandler OnFinishedSpacingCalc;
    public delegate void FinishedPhaseCalcHandler(object sender,
        System.EventArgs args);
    public event FinishedPhaseCalcHandler OnFinishedPhaseCalc;
    public delegate void FinishedCritDiamCalcHandler(object sender,
        CriticalDiameterResults results);
    public event FinishedCritDiamCalcHandler OnFinishedCritDiamCalc;
    public delegate void FinishedPixelDistCalcHandler(object sender,
        System.EventArgs args);
    public event FinishedPixelDistCalcHandler OnFinishedRadialDistCalc
        ;
    private int radDistWorkerDoneCount = 0;
    private int radDistWorkerThreadCount = 0;
    private readonly object ThisLock = new object();
    private double pixelsPerMicron;
    private string fileName;
    private string resultsPath;
    private List<int> objectCalcOrder;
    private OptionsForm.AnalysisOptions.RadialDistributionSettings
        radDistSettings;
    private Color radDistPhaseColor;
    List<uint[]> sumDistResults;
    List<uint[]> sumDistResultsLocal;
    List<int> intersects;
    List<double> distance;
    int spacingWorkerDoneCount = 0;
    int spacingWorkerThreadCount = 0;
    int totalLines = 0;
    OptionsForm.AnalysisOptions.SpacingSettings spacingSettings;
    private bool showLines;
    private int critDiamPhaseCount = 0;

```

```

private int critDiamNumPixels = 0;
private List<List<double>> critDiamPhaseFractions;
private OptionsForm.AnalysisOptions.CriticalDiameterSettings
    critDiamSettings;
const int MAXPOINTS = 2000;
private List<FastBitmap> fastBitmaps = new List<FastBitmap>();
private List<FastBitmap> fastBitmapsAlt = new List<FastBitmap>();
public ImageCalc(Bitmap bitmap)
{
    fastBitmaps.Add(new FastBitmap(bitmap));
}

/// <summary>
/// Attempts to lock calculation by checking calculationInProgress
/// . If calculation is already in progress, an error messagebox
/// is
/// displayed and this function returns false. Returns true and
/// sets calculationInProgress to true otherwise.
/// </summary>
/// <returns></returns>
private bool LockCalculation()
{
    lock (ThisLock)
    {
        if (calculationInProgress)
        {
            System.Windows.Forms.MessageBox.Show("Cannot start _
                calculation _ already in progress.");
            return false;
        }
        calculationInProgress = true;
        return true;
    }
}

private void UnlockCalculation()
{
    lock (ThisLock)
    {
        calculationInProgress = false;
    }
}

public void CalculateSpacing(OptionsForm.AnalysisOptions.
    SpacingSettings settings, double pixelsPerMicron, string
    fileName, string resultsPath)
{
/// Need to create multiple bitmaps, using one bitmap doesn't work
/// because required locks go slower than single threaded app
    if (!LockCalculation())
        return;
    int numProcessors = System.Environment.ProcessorCount;
    if (settings.NumberOfLines < 100 && fastBitmaps[0].Width *
        fastBitmaps[0].Height < 10000000)
        numProcessors = 1; // no need to incur overhead of
        multiple images and threads on such a small number of
        lines

```

```

this.fileName = fileName;
this.showLines = settings.ShowLines;
this.intersects = new List<int>();
this.distance = new List<double>();
this.spacingWorkerDoneCount = 0;
this.spacingWorkerThreadCount = numProcessors;
this.totalLines = settings.NumberOfLines;
this.pixelsPerMicron = pixelsPerMicron;
this.spacingSettings = settings;
if (fastBitmaps.Count > 1) // need to get rid of images (
    should only have one fastBitmap in list after finished calc
    {
        for (int i = 1; i < fastBitmaps.Count; i++)
        {
            fastBitmaps[i].Bitmap.Dispose();
            fastBitmaps.RemoveAt(i);
        }
    }
if (fastBitmapsAlt.Count > 0)
    {
        for (int i = 0; i < fastBitmapsAlt.Count; i++)
        {
            fastBitmapsAlt[i].Bitmap.Dispose();
            fastBitmapsAlt.RemoveAt(i);
        }
    }
for (int i = 1; i < numProcessors; i++) // Create new
    FastBitmaps for every processor we have to work with
    fastBitmaps.Add(new FastBitmap(fastBitmaps[0].Bitmap));
if (showLines)
    for (int i = 0; i < numProcessors; i++) // Create new
        FastBitmaps for every processor we have to work with
        fastBitmapsAlt.Add(new FastBitmap(fastBitmaps[0].
            Bitmap));
for (int i = 0; i < fastBitmaps.Count; i++)
    fastBitmaps[i].LockBitmap();
if (showLines)
    for (int i = 0; i < fastBitmapsAlt.Count; i++)
        fastBitmapsAlt[i].LockBitmap();
Thread newThread;
for (int i = 0; i < numProcessors; i++)
    {
        intersects.Add(0);
        distance.Add(0);
        int threadNum = i;
        if (i + 1 < numProcessors)
        {
            newThread = new Thread(delegate ()
            {
                CalculateSpacingWorker(new OptionsForm.
                    AnalysisOptions.SpacingSettings(settings.
                        NumberOfLines / numProcessors,
                        settings.ShowLines), threadNum);
            });
        }
    }

```

```

        });
    }
    else
    { // on last iteration, may need to add a couple
      iterations due to integer rounding
      newThread = new Thread(delegate()
      {
          int numCalc = settings.NumberOfLines /
              numProcessors;
          int numLines = settings.NumberOfLines - numCalc *
              (numProcessors - 1);
          CalculateSpacingWorker(new OptionsForm.
              AnalysisOptions.SpacingSettings(numLines,
              settings.ShowLines), threadNum);
      });
    }
    newThread.IsBackground = true;
    newThread.Name = "Worker_" + i.ToString() + "_(" + Spacing + ")";
    newThread.Start();
}

public void CalculateSpacingWorker(OptionsForm.AnalysisOptions.
    SpacingSettings settings, int threadNumber)
{
    int numLines = settings.NumberOfLines;
    double progressInterval;
    lock (ThisLock)
    {
        progressInterval = 1.0 / spacingWorkerThreadCount;
    }
    // need to calculate random lines and make sure they do not
      intersect any "red" areas
    int lines = 0, x1 = 0, y1 = 0, x2 = 0, y2 = 0;
    System.Random r = new Random();
    Line myLine;
    Point pointOne, pointTwo;
    for (lines = 0; lines < numLines; lines++)
    {
        Color myColor;
        int col = fastBitmaps[threadNumber].Width;
        int row = fastBitmaps[threadNumber].Height;
    do // randomizes pixel point that is not red
    {
        x1 = r.Next(col - 1);
        y1 = r.Next(row - 1);
        myColor = fastBitmaps[threadNumber].GetPixel(x1, y1);
    } while (myColor == Color.FromArgb(255, 0, 0));
    do // randomizes second pixel point that is not red
    {
        x2 = r.Next(col - 1);
        y2 = r.Next(row - 1);
        myColor = fastBitmaps[threadNumber].GetPixel(x2, y2);

```

```

} while ((myColor == Color.FromArgb(255, 0, 0)) || (x2 == x1 && y2
    == y1));
pointOne = new Point(x1, y1);
pointTwo = new Point(x2, y2);
myLine = new Line(pointOne, pointTwo);
    int pointNumber = 0;
intersects[threadNumber] += FindIntersects(myLine, Color.FromArgb
    (255, 255, 255), Color.FromArgb(255, 0, 0), ref pointNumber,
        threadNumber);
    if (pointNumber > 0) // means a stop color (red) was
        encountered
    {
        myLine.ClipTo(pointNumber - 1);
        y2 = myLine.GetPoint(pointNumber - 1).Y; // make
            sure x2 and y2 are the proper points
        x2 = myLine.GetPoint(pointNumber - 1).X;
    }
    else
    {
        y2 = myLine.GetPoint(myLine.Points - 1).Y; // make
            sure x2 and y2 are the proper points
        x2 = myLine.GetPoint(myLine.Points - 1).X;
    }
distance[threadNumber] += System.Math.Sqrt(Convert.ToDouble((y2-y1)*(y2-y1)
    +(x2-x1)*(x2-x1)));
    if (lines % (numLines/100.0) == 0)
    {
        if (OnProgressUpdate != null)
        {
            OnProgressUpdate(this, progressInterval);
        }
    }
    CalculateSpacingFinished();
}
public void CalculateSpacingFinished()
{
    bool lastThreadDone = false;
    lock (ThisLock)
    {
        spacingWorkerDoneCount++;
        if (spacingWorkerDoneCount == spacingWorkerThreadCount)
            lastThreadDone = true;
    }
    if (lastThreadDone == false)
        return;
    int numRows = fastBitmaps[0].Height;
    int numCol = fastBitmaps[0].Width;
    if (spacingSettings.ShowLines & spacingWorkerThreadCount > 1)
    {
        Color color;
        for (int row = 0; row < numRows; row++)
            for (int col = 0; col < numCol; col++)

```



```

        for (int i = 1; i < fastBitmaps.Count; i++)
        {
            color = fastBitmapsAlt[i].GetPixel(col, row);
            if (color == Color.FromArgb(0, 128, 0)) //
                Careful, Color.Green does not work here
                even if that's how it's been set
            {
                fastBitmapsAlt[0].SetPixel(col, row, color);
                break;
            }
        }
    }
    for (int i = 0; i < fastBitmaps.Count; i++)
    {
        fastBitmaps[i].UnlockBitmap();
        if (showLines)
            fastBitmapsAlt[i].UnlockBitmap();
    }
    int totalIntersects = 0;
    double totalDistance = 0;
    double spacing;
    for (int i = 0; i < spacingWorkerThreadCount; i++)
    {
        totalDistance += distance[i];
        totalIntersects += intersects[i];
    }
    if (totalIntersects > 0)
    {
        spacing = totalDistance / Convert.ToDouble(totalIntersects)
            / pixelsPerMicron;
    }
    else
    {
        spacing = 0;
    }
    for (int i = 1; i < fastBitmaps.Count; i++)
    {
        fastBitmaps[i].Bitmap.Dispose();
        fastBitmaps.RemoveAt(i);
    }
    for (int i = 1; i < fastBitmapsAlt.Count; i++)
    {
        fastBitmapsAlt[i].Bitmap.Dispose();
        fastBitmapsAlt.RemoveAt(i);
    }
    if (AfterSpacingCalc != null)
    {
        SpacingCalcInfo results = new SpacingCalcInfo(spacing,
            totalIntersects, totalDistance, totalLines);
        AfterSpacingCalc(this, results);
    }
    if (OnFinishedSpacingCalc != null)

```

```

    {
        OnFinishedSpacingCalc(this , new System.EventArgs());
    }
    UnlockCalculation();
}
/// <summary>
/// Finds and counts the number of intersects on a given line ,
/// counts individual spacings if trackDistribution is set and
/// stores them
/// in the supplied spacingList, which is ignored otherwise.
/// </summary>
/// <param name="line">Line to analyze</param>
/// <param name="color">Color to count intersections of</param>
/// <param name="stopColor">Color to halt calculation</param>
/// <param name="pointNumber">Point number at which stopColor was
/// encountered, equal to 0 if it was not encountered</param>
/// <returns></returns>
int FindIntersects(Line line , Color color , Color stopColor , ref
int pointNumber , int threadNumber)
{
    int intersects = 0;
    Color oldColor , newColor;
if (color == Color.FromArgb(0,0,0)) // 0,0,0 is black
    oldColor = Color.FromArgb(255,255,255); // 255, 255, 255 is white
else
    oldColor = Color.FromArgb(0,0,0);
    for (int i = 0; i<line.Points; i++)
    {
        newColor = fastBitmaps[threadNumber].GetPixel(line.
            GetPoint(i).X, line.GetPoint(i).Y);
        if (i == 0)
            oldColor = newColor; // prevent first pixel from
            causing intersect
if (newColor != stopColor) // if stop color hit (red) then abort
            line checking process
        {
            if (newColor == color && oldColor != color)
            {
                intersects++;
                if (showLines)
                {
                    fastBitmapsAlt[threadNumber].SetPixel(line.
                        GetPoint(i).X, line.GetPoint(i).Y, Color.
                        Red);
                }
            }
else
            {
                if (showLines)
                {
                    fastBitmapsAlt[threadNumber].SetPixel(line.
                        GetPoint(i).X, line.GetPoint(i).Y, Color.
                        Green);
                }
            }
        }
    }
}

```

```

        }
    }
    oldColor = newColor;
}
else // stop color hit, abort process
{
    pointNumber = i;
    i = line.Points;
}
}
return intersects;
}
/// <summary>
/// Sets pointNumber to point of first intersect, returns false if no
/// intersects are found
/// </summary>
bool FindFirstIntersect(Line line, Color color, ref int pointNumber,
    int threadNumber)
{
    Color oldColor, newColor;
    int i = 0;
    if (color == Color.FromArgb(0,0,0))
        newColor = Color.FromArgb(255,255,255);
    else
        newColor = Color.FromArgb(0,0,0);
    do
    {
        oldColor = newColor;
        newColor = fastBitmaps[threadNumber].GetPixel(line.
            GetPoint(i).X, line.GetPoint(i).Y);
        i++;
    } while (i < line.Points && ((newColor != color) || (oldColor != color
        )));
    if (i == line.Points)
    {
        return false; // no intersects found
    }
    else
    {
        pointNumber = i - 1;
        return true;
    }
}

public void CalculatePhaseFraction(OptionsForm.AnalysisOptions.
    PhaseSettings phaseInfo)
{
    if (!LockCalculation())
        return;
    Thread phaseCalcThread = new Thread(delegate()
    {
        CalculatePhaseFractionWorker(phaseInfo);
    });
    phaseCalcThread.IsBackground = true;
}

```

```

        phaseCalcThread.Name = "Phase_Calc_Worker_0";
        phaseCalcThread.Start();
    }
    public void CalculatePhaseFractionWorker(OptionsForm.
        AnalysisOptions.PhaseSettings phaseInfo)
    {
        int numPhases = phaseInfo.Count;
        int[] phaseCount = new int[numPhases];
        double[] phaseFraction = new double[numPhases];
        Color[] phaseColors = new Color[numPhases];
        Color color;
        int rowPtr;
        phaseInfo.PhaseColors.CopyTo(phaseColors); // speeds up
            analysis slightly
        int numRows = fastBitmaps[0].Height;
        int numCol = fastBitmaps[0].Width;
        int numPixels = numRows * numCol;
        for (int i = 0; i < numPhases; i++)
        {
            phaseCount[i] = 0;
        }
        fastBitmaps[0].LockBitmap();
        for (int row = 0; row < numRows; row++)
        {
            rowPtr = fastBitmaps[0].GetRowPtr(row);
            for (int col = 0; col < numCol; col++)
            {
                color = fastBitmaps[0].GetPixelFromRowPtr(rowPtr, col)
                ;
                for (int j = 0; j < numPhases; j++)
                {
                    if (color == phaseColors[j])
                    {
                        phaseCount[j]++;
                        j = numPhases;
                    }
                }
            }
        }
        fastBitmaps[0].UnlockBitmap();
        for (int i = 0; i < numPhases; i++)
            phaseFraction[i] = Convert.ToDouble(phaseCount[i])/numPixels;
        if (AfterPhaseCalc != null)
        {
            AfterPhaseCalc(this, new PhaseFractionResults(
                phaseFraction));
        }

        if (OnFinishedPhaseCalc != null)
        {
            OnFinishedPhaseCalc(this, new System.EventArgs());
        }
        UnlockCalculation();
    }

```

```

    }
    /// <summary>
    /// Fills circleDiameters and phaseFractions with a list of diameters
    /// (in pixels) and phase
    /// fraction data
    /// </summary>
    /// <param name="phaseColor"></param>
    /// <param name="circleDiameters"></param>
    /// <param name="phaseFractions"></param>
    public void CalcCritDiamFractions(OptionsForm.AnalysisOptions.
        CriticalDiameterSettings settings, Color phaseColor)
    {
        if (!LockCalculation())
            return;
        critDiamSettings = settings;
        Thread t = new Thread(delegate()
        {
            CalcCritDiamMain(phaseColor);
        });
        t.IsBackground = true;
        t.Name = "CriticalDiameterDriverThread";
        t.Start();
    }
    private void CalcCritDiamMain(Color phaseColor)
    {
        int numProcessors = System.Environment.ProcessorCount;
        if (fastBitmaps.Count > 1) // need to get rid of images (
            should only have one fastBitmap in list after finished calc
        {
            for (int i = 1; i < fastBitmaps.Count; i++)
            {
                fastBitmaps[i].Bitmap.Dispose();
                fastBitmaps.RemoveAt(i);
            }
        }
        for (int i = 1; i < numProcessors; i++) // Create new
            FastBitmaps for every processor we have to work with
            fastBitmaps.Add(new FastBitmap(fastBitmaps[0].Bitmap));
        for (int i = 0; i < fastBitmaps.Count; i++)
            fastBitmaps[i].LockBitmap();

        int[] diameterList = {10, 20, 30, 40, 50, 60, 70, 80, 90, 100,
            120, 140, 160, 180, 200, 220,
            240, 260, 280, 300, 320, 340, 360, 380, 400, 450, 500,
            550, 600,
            650, 700, 750, 800, 850, 900, 950};
        ArrayList diameters = new ArrayList(diameterList);
        ArrayList phaseFractions = new ArrayList();
        ArrayList circleDiameters = new ArrayList();
        Color rgbPhaseColor = Color.FromArgb(phaseColor.R, phaseColor.
            G, phaseColor.B);
        foreach (object diameter in diameters)
        {

```

```

        if ((int)diameter < fastBitmaps[0].Width && (int)diameter
            < fastBitmaps[0].Height)
        {
            double[] fractions = CalcPhaseFractionDriver(
                rgbPhaseColor, (int)diameter);
            phaseFractions.AddRange(fractions);
            for (int i = 0; i < fractions.Length; i++)
            {
                circleDiameters.Add((int)diameter);
            }
        }
        if (OnProgressUpdate != null)
        {
            OnProgressUpdate(this, 1.0 / diameters.Count * 100);
        }
    }
    for (int i = 0; i < fastBitmaps.Count; i++)
        fastBitmaps[i].UnlockBitmap();
    if (OnFinishedCritDiamCalc != null)
    {
        OnFinishedCritDiamCalc(this, new CriticalDiameterResults(
            circleDiameters, phaseFractions));
    }
    UnlockCalculation();
}
///<summary>
///Returns an array of phase fractions calculated from circles
///with the supplied diameter.
///</summary>
///<param name="phaseColor"></param>
///<param name="diameter"></param>
///<returns></returns>
private double[] CalcPhaseFractionDriver(Color phaseColor, int
    diameter)
{
    if (critDiamSettings.CalculationMethod == OptionsForm.
        AnalysisOptions.CriticalDiameterSettings.CalcMethod.
        RandomCircles)
        return CalcByRandomCircles(phaseColor, diameter);
    else
        return CalcByGridCircles(phaseColor, diameter);
}
private double[] CalcByGridCircles(Color phaseColor, int diameter)
{
    int maxLtoR = (fastBitmaps[0].Width - 1) / diameter;
    int maxTtoB = (fastBitmaps[0].Height - 1) / diameter;
    int xDistBetweenCircles = 0;
    int yDistBetweenCircles = 0;
    if (maxLtoR > 5)
    {
        maxLtoR = 5;
        ///xDistBetweenCircles = this.mnCol/5-diameter;
    }
}

```

```

xDistBetweenCircles = fastBitmaps[0].Width / maxLtoR -
    diameter;
if (maxTtoB > 5)
{
    maxTtoB = 5;
    //yDistBetweenCircles = this.mnRow/5-diameter;
}
yDistBetweenCircles = fastBitmaps[0].Height / maxTtoB -
    diameter;
int numCircles = maxLtoR * maxTtoB;
double[] phaseFractions = new double[numCircles];
//Circle circle = new Circle(diameter, new Point(diameter/2,
    diameter/2));
Circle circle = new Circle(diameter, new Point(diameter / 2 +
    xDistBetweenCircles / 2, diameter / 2 + yDistBetweenCircles
    / 2));
for (int x = 0; x < maxLtoR; x++)
{
    for (int y = 0; y < maxTtoB; y++)
    {
        //if (maxTtoB == 1 && maxLtoR == 1)
        //circle.DrawCircle(this);
        phaseFractions[x * maxTtoB + y] =
            CalcPhaseFractionCircle(phaseColor, circle);
        circle.Offset(new Point(0, diameter +
            yDistBetweenCircles));
    }
    circle.Offset(new Point(diameter + xDistBetweenCircles, -(
        diameter + yDistBetweenCircles) * maxTtoB));
}
return phaseFractions;
}

/// <summary>
///
/// </summary>
/// <param name="phaseColor"></param>
/// <param name="diameter"></param>
/// <returns></returns>
private double[] CalcByRandomCircles(Color phaseColor, int
    diameter)
{
    // Now want to generate randomly placed circles of the
    specified diameter and compile phase fraction measurements
    to a double array
    double totalCircleArea = fastBitmaps[0].Height * fastBitmaps
        [0].Width;
    double individualCircleArea = diameter * diameter * 0.7853982;
        // Area = pi * r^2, or Area = d^2 * pi / 4
    int numCircles = (int)(totalCircleArea / individualCircleArea
        + 0.5);
    int numProcessors = fastBitmaps.Count;
    critDiamPhaseFractions = new List<List<double>>();
    bool splitCircles = false;

```

```

if (numCircles < numProcessors)
{
    // Have more processors than circles, so will run just one
    worker thread and tell it to split circles up
    numProcessors = 1;
    splitCircles = true;
}
Random seed = new Random();
Thread[] threads = new Thread[numProcessors];
for (int i = 0; i < numProcessors; i++)
{
    int d = diameter;
    int threadNum = i;
    int threadCircles;
    Random r = new Random(seed.Next());
    //Thread.Sleep(1);    // ensures the next random number
    //won't be created with the same seed number
    if (i + 1 < numProcessors)
        threadCircles = numCircles / numProcessors;
    else
        threadCircles = numCircles - numCircles /
            numProcessors * (numProcessors - 1);
    critDiamPhaseFractions.Add(new List<double>(threadCircles)
    );
    threads[i] = new Thread(delegate()
    {
        CalcByRandomCirclesWorker( Color.FromArgb(
            phaseColor.R, phaseColor.G, phaseColor.B), d,
            threadNum, threadCircles,
            splitCircles, r);
    });
    threads[i].IsBackground = true;
    threads[i].Name = "CD_Rand_Worker_" + threadNum.ToString();
    ;
    threads[i].Start();
}
for (int i = 0; i < numProcessors; i++)
{
    threads[i].Join();
}
List<double> combPhaseFractions = new List<double>(numCircles)
;
for (int i = 0; i < numProcessors; i++)
{
    combPhaseFractions.AddRange(critDiamPhaseFractions[i]);
}
return combPhaseFractions.ToArray();
}
/// <summary>
/// Generates the specified number of circles and calculates the
    phase fraction using the circle phase fraction worker function.
    Phase
/// fraction results are stored to critDiamPhaseFractions[
    threadNum]

```



```

/// </summary>
/// <param name="phaseColor"></param>
/// <param name="diameter"></param>
/// <param name="threadNum"></param>
/// <param name="numCircles"></param>
private void CalcByRandomCirclesWorker(Color phaseColor, int
    diameter, int threadNum, int numCircles, bool splitCircles,
    Random r)
{
    int radius = diameter / 2 + 1;
    int maxX = fastBitmaps[threadNum].Width - radius;
    int minX = radius;
    int maxY = fastBitmaps[threadNum].Height - radius;
    int minY = radius;
    int xRange = maxX - minX;
    int yRange = maxY - minY;
    int x = r.Next(minX, maxX + 1);
    int y = r.Next(minY, maxY + 1);
    Circle circle = new Circle(diameter, x, y);
    for (int i = 0; i < numCircles; i++)
    {
        x = r.Next(minX, maxX + 1);
        y = r.Next(minY, maxY + 1);
        circle.Offset(new Point(x - circle.Center.X, y - circle.
            Center.Y));
        if (splitCircles)
        {
            critDiamPhaseFractions[threadNum].Add(
                CalcPhaseFractionCircle(phaseColor, circle));
        }
        else
        {
            CalcPhaseFractionCircleWorker(phaseColor, circle,
                threadNum, 0, circle.Diameter);
        }
    }
}

private double CalcPhaseFractionCircleSingleThreaded(Color
    phaseColor, Circle theCircle)
{
    int numPixels = 0;
    int phaseCount = 0;
    for (int y = 0; y <= theCircle.Diameter; y++) // iterate top to
        bottom
    {
        for (int x = theCircle.LeftPoints[y].X; x <= theCircle.RightPoints
            [y].X; x++)
        {
            if (fastBitmaps[0].GetPixel(x, theCircle.LeftPoints[y
                ].Y) == phaseColor)
            {
                phaseCount++;
            }
        }
    }
}

```

```

numPixels++;
}
}
double phaseFraction = Convert.ToDouble(phaseCount)/numPixels;
return phaseFraction;
}
///<summary>
///Calculates phase fraction within the given circle using as
///many threads as there are fastbitmaps available.
///</summary>
///<param name="phaseColor"></param>
///<param name="theCircle"></param>
private double CalcPhaseFractionCircle(Color phaseColor, Circle
theCircle)
{
    int numThreads = fastBitmaps.Count;
    List<Circle> circles = new List<Circle>();
    for (int i = 0; i < numThreads; i++)
    {
        circles.Add(new Circle(theCircle.Diameter, theCircle.
Center));
    }
    Thread[] threads = new Thread[numThreads];
    critDiamNumPixels = 0;
    critDiamPhaseCount = 0;
    for (int i = 0; i < numThreads; i++)
    {
        int threadNum = i;
        if (i + i < numThreads)
        {
            threads[threadNum] = new Thread(delegate()
            {
                CalcPhaseFractionCircleWorker(Color.FromArgb(
phaseColor.R, phaseColor.G, phaseColor.B),
circles[threadNum], threadNum,
threadNum*circles[threadNum].Diameter/
numThreads, (threadNum+1)*circles[
threadNum].Diameter/numThreads -1);
            });
        }
        else // may need more loops for the last thread
        {
            threads[threadNum] = new Thread(delegate()
            {
                CalcPhaseFractionCircleWorker(Color.FromArgb(
phaseColor.R, phaseColor.G, phaseColor.B),
circles[threadNum], threadNum,
threadNum * circles[threadNum].Diameter /
numThreads, circles[threadNum].Diameter
));
            });
        }
    }
}

```

```

    for (int i = 0; i < numThreads; i++)
    {
        threads[i].IsBackground = true;
        threads[i].Name = "Crit_Diam_Worker_" + i.ToString();
        threads[i].Start();
    }
    for (int i = 0; i < numThreads; i++)
    {
        threads[i].Join();
    }
    return ((double)critDiamPhaseCount) / critDiamNumPixels;
}
/// <summary>
/// Calculates phase fraction for the supplied circle and the
/// given y values using fastbitmap[threadNum]. If a full circle
/// is being
/// calculated (based on endY - startY == theCircle.diameter) then
/// results are stored to critDiamPhaseFractions[threadNum]. If a
/// part
/// circle is being calculated then results are stored to
/// critDiamNumPixels and critDiamPhaseCount with thread-safe code.
/// </summary>
/// <param name="phaseColor"></param>
/// <param name="theCircle"></param>
/// <param name="threadNum"></param>
/// <param name="startY"></param>
/// <param name="endY"></param>
private void CalcPhaseFractionCircleWorker(Color phaseColor,
    Circle theCircle, int threadNum, int startY, int endY)
{
    int numPixels = 0;
    int phaseCount = 0;
    for (int y = startY; y <= endY; y++)
    {
        for (int x = theCircle.LeftPoints[y].X; x <= theCircle.
            RightPoints[y].X; x++)
        {
            if (fastBitmaps[threadNum].GetPixel(x, theCircle.
                LeftPoints[y].Y) == phaseColor)
            {
                phaseCount++;
            }
            numPixels++;
        }
    }
    if (endY - startY == theCircle.Diameter) // Detects if a
        full circle is being calculated
    {
        critDiamPhaseFractions[threadNum].Add(((double)phaseCount)
            / numPixels);
    }
    else
    {

```

```

        lock (ThisLock)
        {
            critDiamNumPixels += numPixels;
            critDiamPhaseCount += phaseCount;
        }
    }
}
/// <summary>
/// Calculates pixel distribution of the supplied image and
/// outputs the data to a text file
/// </summary>
/// <param name="phaseColor">Phase color for which to analyze the
/// distance between pixels</param>
/// <param name="pixelsPerMicron">Image calibration</param>
/// <param name="fileName">File name of the image being analyzed,
/// not including the path, including the extension</param>
public void CalcRadialDistribution(Color phaseColor, double
pixelsPerMicron, string fileName, string resultsPath,
OptionsForm.AnalysisOptions.RadialDistributionSettings
radDistSettings)
{
    if (!LockCalculation())
        return;
    this.fileName = fileName;
    this.resultsPath = resultsPath;
    this.pixelsPerMicron = pixelsPerMicron;
    this.radDistSettings = radDistSettings;
    stopCalculations = false;
    if (radDistSettings.LocalAnalysis)
    {
        if (radDistSettings.RadDistCalcStyle ==
OptionsForm.AnalysisOptions.RadialDistributionSettings.
RadialDistributionCalculationStyle.FromCircle)
        {
            CalcRadialDistLocalFromCircle(phaseColor);
        }
        else
        {
            CalcRadialDistLocalFromObject(phaseColor);
        }
        return;
    }
    fastBitmaps[0].LockBitmap();
    // Stores locations of pixels of color specified by phaseColor
    List<ushort> phasePointsX = new List<ushort>();
    List<ushort> phasePointsY = new List<ushort>();
    for (int row = 0; row < fastBitmaps[0].Height; row++)
        for (int col = 0; col < fastBitmaps[0].Width; col++)
        {
            if (fastBitmaps[0].GetPixel(col, row) == phaseColor)
            {
                phasePointsX.Add((ushort)col);
                phasePointsY.Add((ushort)row);
            }
        }
    }
}

```

```

    }
}
fastBitmaps[0].UnlockBitmap();

ushort[] pointX = phasePointsX.ToArray();
ushort[] pointY = phasePointsY.ToArray();
if (fastBitmaps[0].Height * fastBitmaps[0].Width > 10000000)
{
    System.Windows.Forms.DialogResult result = System.Windows.
        Forms.MessageBox.Show("Image_size_larger_than_10_
        million_pixels."
        + "This_calculation_may_take_a_long_time_to_complete
        or_slow_down_your_system.Continue_anyway?", "
        Warning!",
        System.Windows.Forms.MessageBoxButtons.YesNo);
    if (result == System.Windows.Forms.DialogResult.No)
        return;
}
DateTime timeStart = DateTime.Now;
int numProcessors = System.Environment.ProcessorCount;
int loopsMain = pointX.Length - 1;
int loopsSecondary = 0;
if (numProcessors > 1)
{
    numProcessors = 2;
    loopsMain = Convert.ToInt32((pointX.Length) * 0.293 + .01)
        ; // 0.01 because 4.5 is converted to 4
    loopsSecondary = Convert.ToInt32((pointX.Length) * 0.707 +
        .01); // 0.01 because 4.5 is converted to 4
    if (loopsMain + loopsSecondary < pointX.Length)
        loopsMain += pointX.Length - loopsSecondary -
            loopsMain;
}
else
{
    loopsMain = pointX.Length - 1;
}
radDistWorkerThreadCount = numProcessors;
radDistWorkerDoneCount = 0;
if (OnLockAnalyzeMenu != null)
{
    OnLockAnalyzeMenu(this, new EventArgs());
}
sumDistResults = new List<uint[]>();
Thread[] t = new Thread[numProcessors];
int numDistances = fastBitmaps[0].Height * fastBitmaps[0].
    Width * 10;
for (int i = 0; i < numProcessors; i++)
{
    sumDistResults.Add(new uint[numDistances]);
    CalcPixelDistThreadParameters calcParams = new
        CalcPixelDistThreadParameters(i, loopsMain, pointX,
        pointY);

```

```

        t[i] = new Thread(new ParameterizedThreadStart(
            CalcPixelDistributionWorker));
        t[i].IsBackground = true;
        t[i].Start(calcParams);
    }
    DateTime timeEnd = DateTime.Now;
    TimeSpan timeDiff = timeEnd - timeStart;
}
private void CalcPixelDistributionWorker(object calcParams)
{ // loopsMain is how many loops the first thread should do, this
  is currently just optimized for dual core
    CalcPixelDistThreadParameters parameters = (
        CalcPixelDistThreadParameters)calcParams;
    int yDiff;
    int xDiff;
    int lastLoop;
    if (parameters.threadNumber == 0)
        lastLoop = parameters.loopsMain;
    else
        lastLoop = parameters.pointX.Length - 1;

    int interval = parameters.pointX.Length / 100;
    if (interval < 1)
        interval = 100 / parameters.pointX.Length;
    for (int i = parameters.loopsMain * parameters.threadNumber; i
        < lastLoop; i++)
    {
        for (int j = i + 1; j < parameters.pointX.Length; j++)
        {
            yDiff = parameters.pointY[j] - parameters.pointY[i];
            xDiff = parameters.pointX[j] - parameters.pointX[i];

            sumDistResults[parameters.threadNumber][xDiff * xDiff
                + yDiff * yDiff]++;
        }
        if (i % interval == 0)
        {
            if (OnProgressUpdate != null)
            {
                OnProgressUpdate(this, 1);
            }
            if (stopCalculations == true)
            {
                break;
            }
        }
    }
}
bool allThreadsFinished = false;
lock (ThisLock)
{
    radDistWorkerDoneCount++;
    if (radDistWorkerDoneCount == radDistWorkerThreadCount)
        allThreadsFinished = true;
}

```

```

    }
    if (allThreadsFinished)
    {
        CalcPixelDistThreadsDone();
    }
}

private void CalcPixelDistThreadsDone()
{
    // All worker threads are now done calculating, results are
    // contained within sumDist[][]
    int numDistances = fastBitmaps[0].Height * fastBitmaps[0].
        Width * 10;
    uint[] sumDist = new uint[numDistances];
    sumDist.Initialize();
    for (int i = 0; i < numDistances; i++)
        for (int j = 0; j < radDistWorkerThreadCount; j++)
        {
            sumDist[i] += sumDistResults[j][i];
        }
    System.IO.StreamWriter outFile = null;
    try
    {
        string path = resultsPath + "\\Pixel_Distribution_Results"
            ;
        System.IO.DirectoryInfo resultsDirectory = new System.IO.
            DirectoryInfo(path);
        if (!resultsDirectory.Exists)
            resultsDirectory.Create();
        int periodIndex = fileName.LastIndexOf(".");
        if (periodIndex == -1)
            periodIndex = fileName.Length;
        string resultsFileName = path + "\\\" + fileName.Substring
            (0, periodIndex) + ".txt";
        outFile = new System.IO.StreamWriter(resultsFileName);
        outFile.WriteLine("D, _mic\tCounts");
        double distanceNormalized;
        Random rand = new Random();
        for (int i = 0; i < sumDist.Length; i++)
        {
            if (sumDist[i] > 0)
            {
                distanceNormalized = System.Math.Sqrt(i) /
                    pixelsPerMicron;
                outFile.WriteLine(distanceNormalized.ToString() +
                    "\t" + sumDist[i].ToString()); // uncomment
                    this and ignore next to revert to normal
            }
        }
    }
    catch (Exception ex)
    {
        System.Windows.Forms.MessageBox.Show("Error in _ImageCalc, _
            unable to _output _file. _ _ _Exception _details:\r\n" + ex.
                ToString());
    }
}

```

```

    }
    finally
    {
        if (outFile != null)
            outFile.Close();
    }
    if (OnUnLockAnalyzeMenu != null)    // this event also spawns
        the ChildForm's AfterCalculatePixelDistribution method
    {
        OnUnLockAnalyzeMenu(this, new EventArgs());
    }
    if (OnFinishedRadialDistCalc != null)
    {
        OnFinishedRadialDistCalc(this, new EventArgs());
    }
    UnlockCalculation();
}
private class CalcPixelDistThreadParameters
{
    public int threadNumber;
    public int loopsMain;
    public ushort[] pointX;
    public ushort[] pointY;
    public CalcPixelDistThreadParameters(int threadNumber, int
        loopsMain, ushort[] pointX, ushort[] pointY)
    {
        this.threadNumber = threadNumber;
        this.loopsMain = loopsMain;
        this.pointX = pointX;
        this.pointY = pointY;
    }
}
private void CalcRadialDistLocalFromCircle(Color phaseColor)
{
    if (radDistSettings.MaxRadius * 2 * pixelsPerMicron >
        fastBitmaps[0].Width ||
        radDistSettings.MaxRadius * 2 * pixelsPerMicron >
            fastBitmaps[0].Height)
    {
        System.Windows.Forms.MessageBox.Show("Circle_dimensions_
            larger_than_image_dimensions...Cannot_proceed.");
        if (OnUnLockAnalyzeMenu != null)    // this event also
            spawns the ChildForm's AfterCalculatePixelDistribution
            method
        {
            OnUnLockAnalyzeMenu(this, new EventArgs());
        }
        if (OnFinishedRadialDistCalc != null)
        {
            OnFinishedRadialDistCalc(this, new EventArgs());
        }
        calculationInProgress = false;
        return;
    }
}

```



```

}
int maxCircleD = (int)(radDistSettings.MaxRadius *
    pixelsPerMicron * 2);
int maxCircleR = (int)(radDistSettings.MaxRadius *
    pixelsPerMicron);
int localCircleR = (int)(radDistSettings.LocalRadius *
    pixelsPerMicron * 2);
Point circleLocation = new Point(maxCircleR, maxCircleR);
Circle localCircle = new Circle(localCircleR, circleLocation);
Circle maxCircle = new Circle(maxCircleD, circleLocation);
int yPoint;
int numCircleRows = fastBitmaps[0].Height / maxCircleD;
int numCircleCols = fastBitmaps[0].Width / maxCircleD;
int numCircles = numCircleCols * numCircleRows;
int xOffset = 0;
int yOffset = 0;
// Stores locations of pixels of color specified by phaseColor
List<List<ushort>> phasePointsXlocal = new List<List<ushort
    >>();
List<List<ushort>> phasePointsYlocal = new List<List<ushort
    >>();
List<List<ushort>> phasePointsXmax = new List<List<ushort>>();
List<List<ushort>> phasePointsYmax = new List<List<ushort>>();
for (int i = 0; i < numCircles; i++)
{
    phasePointsXlocal.Add(new List<ushort>(maxCircleD)); //
        currently just scaling initial size of list with size
        of circles
    phasePointsYlocal.Add(new List<ushort>(maxCircleD));
    phasePointsXmax.Add(new List<ushort>(maxCircleD));
    phasePointsYmax.Add(new List<ushort>(maxCircleD));
}
bool cancelAnalysis = false;
if (radDistSettings.ShowCircles)
{
    fastBitmapsAlt.Clear();
    fastBitmapsAlt.Add(new FastBitmap(fastBitmaps[0].Bitmap));
    fastBitmapsAlt[0].LockBitmap();
}
fastBitmaps[0].LockBitmap();
for (int circleNum = 0; circleNum < numCircles; circleNum++)
{
    for (int i = 0; i < localCircle.Diameter; i++) // This
        iterates through all points within one local (small)
        circle
    {
        yPoint = localCircle.LeftPoints[i].Y;
        for (int j = localCircle.LeftPoints[i].X; j <
            localCircle.RightPoints[i].X; j++)
            if (fastBitmaps[0].GetPixel(j, yPoint) ==
                phaseColor)
            {
                phasePointsXlocal[circleNum].Add((ushort)j);
            }
        }
    }
}

```

```

        phasePointsYlocal[circleNum].Add((ushort)
            yPoint);
    }
}
if (phasePointsXlocal[circleNum].Count == 0) // no pixels
found in circle, so cannot use it
{
    numCircles--;
    circleNum--;
    if (numCircles <= 0)
    {
        cancelAnalysis = true;
        break;
    }
    continue;
}
for (int i = 0; i < maxCircle.Diameter; i++) // This
iterates through all points within one local (small)
circle
{
    yPoint = maxCircle.LeftPoints[i].Y;
    for (int j = maxCircle.LeftPoints[i].X; j < maxCircle.
        RightPoints[i].X; j++)
        if (fastBitmaps[0].GetPixel(j, yPoint) ==
            phaseColor)
        {
            phasePointsXmax[circleNum].Add((ushort)j);
            phasePointsYmax[circleNum].Add((ushort)yPoint)
                ;
        }
}
if (radDistSettings.ShowCircles)
{
    localCircle.DrawCircle(fastBitmapsAlt[0]);
    maxCircle.DrawCircle(fastBitmapsAlt[0]);
}
xOffset = maxCircleD;    // Scan circles left to right, top
to bottom
yOffset = 0;
if (localCircle.Center.X + maxCircleR * 3 > fastBitmaps
    [0].Width)
{
    xOffset = maxCircleR - localCircle.Center.X;    // Set
x value of circle back to initial value
    yOffset = maxCircleD;
}
localCircle.Offset(new Point(xOffset, yOffset));
maxCircle.Offset(new Point(xOffset, yOffset));
}
fastBitmaps[0].UnlockBitmap();
if (radDistSettings.ShowCircles)
{
    fastBitmapsAlt[0].UnlockBitmap();

```

```

}
if (cancelAnalysis == true)
{
    System.Windows.Forms.MessageBox.Show("No pixels of the
        specified phase were found! Cannot proceed.");
    return;
}
List<ushort[]> pointXlocal = new List<ushort[]>(numCircles);
List<ushort[]> pointYlocal = new List<ushort[]>(numCircles);
List<ushort[]> pointXmax = new List<ushort[]>(numCircles);
List<ushort[]> pointYmax = new List<ushort[]>(numCircles);
for (int i = 0; i < numCircles; i++)
{
    pointXlocal.Add(phasePointsXlocal[i].ToArray());
    pointYlocal.Add(phasePointsYlocal[i].ToArray());
    pointXmax.Add(phasePointsXmax[i].ToArray());
    pointYmax.Add(phasePointsYmax[i].ToArray());
}
// now have list of ushort arrays, each array corresponds
to different local circle/max circle image pairs
int numProcessors = System.Environment.ProcessorCount;
// max dist is ~ Rlocal + Rmax, so max sum squares is this
value squared
int numDistances = (maxCircleR + localCircleR + 1) * (
    maxCircleR + localCircleR + 1); // add one just to be safe
from rounding effects
radDistWorkerThreadCount = numProcessors;
radDistWorkerDoneCount = 0;
sumDistResults = new List<uint[]>();
for (int i = 0; i < numCircles; i++)
{
    sumDistResults.Add(new uint[numDistances]); // initializes
the proper number of array elements in the results
list
}
Thread newThread;
for (int i = 0; i < numProcessors; i++)
{
    int threadNum = i;
    int startNum = (int)((double)numCircles / numProcessors *
        i); // determines the first and last pointX arrays
to calculate
    int endNum = (int)((double)numCircles / numProcessors * (i
        + 1)); // for each thread
    newThread = new Thread(delegate()
    {
        CalcRadialDistWorkerLocal(threadNum, pointXlocal,
            pointYlocal, pointXmax, pointYmax, startNum, endNum
        );
    });
    newThread.IsBackground = true;
    newThread.Name = "Worker_" + i.ToString() + "_" +
        RadDistLocalCircle";
    newThread.Start();
}

```

```

    }
}
/// <summary>
/// Called when Radial Distribution Calculation needs objects to
/// be found... need to call back the RadDist routine
/// </summary>
/// <param name="sender"></param>
/// <param name="args"></param>
void ImageCalc_OnObjectsFound(object sender, ImageCalc.
    FindObjectResults results)
{
    OnObjectsFound -= new ObjectsFoundHandler(
        ImageCalc_OnObjectsFound);
    if (theObjects != null) // avoid infinite loop if theObjects
        was not initialized for some reason
        CalcRadialDistLocalFromObject(radDistPhaseColor);
}
private void CalcRadialDistLocalFromObject(Color phaseColor)
{
    // want to calculate radial distribution on every object whose
    bounds are far enough from the edge to prevent pixel cut
    off when determining the max circle pixels
    // First make sure FindObjects has been run
    if (theObjects == null)
    {
        OnObjectsFound += new ObjectsFoundHandler(
            ImageCalc_OnObjectsFound);
        radDistPhaseColor = phaseColor;
        UnlockCalculation(); // in the future may want to call
            different private FindObjects to avoid this split
            second unlocking of calc
        FindObjects(phaseColor);
        return;
    }
    List<ObjectInfo> validObjects = new List<ObjectInfo>();
    objectCalcOrder = new List<int>();
    BoolBitmap bBitmap = new BoolBitmap(fastBitmaps[0]);
    List<ushort[]> pointXlocal = new List<ushort[]>();
    List<ushort[]> pointYlocal = new List<ushort[]>();
    List<ushort[]> pointXmax = new List<ushort[]>();
    List<ushort[]> pointYmax = new List<ushort[]>();
    List<ushort> phasePointsXmax = new List<ushort>();
    List<ushort> phasePointsYmax = new List<ushort>();
    ushort[] xLocals;
    ushort[] yLocals;
    int radius = (int)(radDistSettings.MaxRadius * pixelsPerMicron
        + 0.5);
    int largestDiameter = 0;
    int largestBoundDiag = 0;
    if (radDistSettings.ShowCircles)
    {
        fastBitmapsAlt.Clear();
        fastBitmapsAlt.Add(new FastBitmap(fastBitmaps[0].Bitmap));
    }
}

```

```

        fastBitmapsAlt[0].LockBitmap();
    }
    fastBitmaps[0].LockBitmap();
    foreach (ObjectInfo obj in theObjects.GetObjects[phaseColor])
    {
        if (obj.ObjectBounds.Left - radius > 0 &&
            obj.ObjectBounds.Right + radius < fastBitmaps[0].Width
            &&
            obj.ObjectBounds.Top - radius > 0 &&
            obj.ObjectBounds.Bottom + radius < fastBitmaps[0].
                Height)
        {
            // add the object points to be analyzed if it is far
            enough from the edge to prevent pixel cut off
            int boundDiag = (int)(System.Math.Sqrt(obj.
                ObjectBounds.Width * obj.ObjectBounds.Width +
                obj.ObjectBounds.Height * obj.ObjectBounds.Height)
                / 2 + 0.5);
            int maxCircleR = (int)(radDistSettings.MaxRadius *
                pixelsPerMicron + boundDiag + 0.5);
            obj.ObjectBounds); // + boundDiag + 0.5);
            Point center = new Point(obj.ObjectBounds.Left + obj.
                ObjectBounds.Width / 2,
                obj.ObjectBounds.Top + obj.ObjectBounds.Height /
                2);
            if (center.X - maxCircleR <= 0 ||
                center.X + maxCircleR >= fastBitmaps[0].Width ||
                center.Y - maxCircleR <= 0 ||
                center.Y + maxCircleR >= fastBitmaps[0].Height)
            {
                continue;
            }
            Circle maxCircle = new Circle(maxCircleR * 2, center);
            if (maxCircle.Diameter > largestDiameter)
                largestDiameter = maxCircle.Diameter;
            if (boundDiag > largestBoundDiag)
                largestBoundDiag = boundDiag;
            int yPoint;
            phasePointsXmax.Clear();
            phasePointsYmax.Clear();
            for (int i = 0; i < obj.AllPoints.Length; i++)
                bBitmap.Set(obj.AllPoints[i].X, obj.AllPoints[i].Y
                    , true);
            for (int i = 0; i < maxCircle.Diameter; i++) // This
                iterates through all points within the large circle
            {
                yPoint = maxCircle.LeftPoints[i].Y;
                for (int j = maxCircle.LeftPoints[i].X; j <
                    maxCircle.RightPoints[i].X; j++)
                    if (fastBitmaps[0].GetPixel(j, yPoint) ==
                        phaseColor && bBitmap.Get(j, yPoint) ==
                            false)
                    {
                        // only add points if they are not from
                        within the object area

```

```

        phasePointsXmax.Add((ushort)j); // Build a
            list of points for the max circle
        phasePointsYmax.Add((ushort)yPoint);
    }
}
pointXmax.Add(phasePointsXmax.ToArray()); // Store
    the list of points as an array of points
pointYmax.Add(phasePointsYmax.ToArray());
for (int i = 0; i < obj.AllPoints.Length; i++)
    bBitmap.Set(obj.AllPoints[i].X, obj.AllPoints[i].Y
        , false);

    if (radDistSettings.ShowCircles)
        maxCircle.DrawCircle(fastBitmapsAlt[0]);
    obj.ToXYArray(out xLocals, out yLocals);
    pointXlocal.Add(xLocals);
    pointYlocal.Add(yLocals);
    validObjects.Add(obj);
    objectCalcOrder.Add(obj.ObjectNumber);
}
}
fastBitmaps[0].UnlockBitmap();
if (radDistSettings.ShowCircles)
{
    fastBitmapsAlt[0].UnlockBitmap();
}
if (pointXlocal.Count <= 0 || pointXmax.Count <= 0)
{
    System.Windows.Forms.MessageBox.Show("No objects found
        within bounds specified by Max Radius.");
    if (OnUnLockAnalyzeMenu != null) // this event also
        spawns the ChildForm's AfterCalculatePixelDistribution
        method
    {
        OnUnLockAnalyzeMenu(this, new EventArgs());
    }
    if (OnFinishedRadialDistCalc != null)
    {
        OnFinishedRadialDistCalc(this, new EventArgs());
    }
    UnlockCalculation();
    return;
}
int numProcessors = System.Environment.ProcessorCount;
if (pointXlocal.Count == 1)
    numProcessors = 1;
// max dist is ~ Rlocal + Rmax, so max sum squares is this
value squared
int numDistances = (largestDiameter / 2 + largestBoundDiag +
    1) * (largestDiameter / 2 + largestBoundDiag + 1) * 2;
int numDistancesLocal = largestBoundDiag * largestBoundDiag *
    8;
int numCircles = pointXlocal.Count;

```

```

radDistWorkerThreadCount = numProcessors;
radDistWorkerDoneCount = 0;
sumDistResults = new List<uint[]>();
sumDistResultsLocal = new List<uint[]>();
for (int i = 0; i < numCircles; i++)
{
    sumDistResults.Add(new uint[numDistances]); // initializes
        the proper number of array elements in the results
        list
    sumDistResultsLocal.Add(new uint[numDistancesLocal]);
}
Thread newThread;
for (int i = 0; i < numProcessors; i++)
{
    int threadNum = i;
    int startNum = (int)((double)numCircles / numProcessors *
        i); // determines the first and last pointX arrays
        to calculate
    int endNum = (int)((double)numCircles / numProcessors * (i
        + 1)); // for each thread
    newThread = new Thread(delegate()
    {
        CalcRadialDistWorkerLocal(threadNum, pointXlocal,
            pointYlocal, pointXmax, pointYmax, startNum, endNum
        );
    });
    newThread.IsBackground = true;
    newThread.Name = "Worker_" + i.ToString() + "_" +
        RadDistLocalObject);
    newThread.Start();
}
}
/// <summary>
/// Calculates radial distribution from supplied lists of x and y
/// point values, stored as ushort array lists. Frequency of
/// results are
/// stored in sumDistResults[arraynum][SumOfSquares]
/// </summary>
/// <param name="threadNumber">Thread number</param>
/// <param name="pointXlocal">List of arrays containing local
/// point X values</param>
/// <param name="pointYlocal">List of arrays containing local
/// point Y values</param>
/// <param name="pointXmax">List of arrays containing point X
/// values outside the local area</param>
/// <param name="pointYmax">List of arrays containing point Y
/// values outside the local area</param>
/// <param name="startNum">Specifies start value of array to store
/// the results in</param>
/// <param name="endNum">Specifies end value of array to store the
/// results in</param>
private void CalcRadialDistWorkerLocal(int threadNumber, List<
    ushort[]> pointXlocal, List<ushort[]> pointYlocal,

```

```

List<ushort[]> pointXmax, List<ushort[]> pointYmax, int
    startNum, int endNum)
{
    int yDiff;
    int xDiff;
    int loopNum = 0;
    int totalPoints = 0;
    for (int i = 0; i < pointXlocal.Count; i++)
    {
        totalPoints += pointXlocal[i].Length;
    }
    int numThreads;
    lock (ThisLock)
    {
        numThreads = radDistWorkerThreadCount;
    }
    double increment = 1.0;
    int interval = totalPoints / 100;
    if (interval < 1)
    {
        interval = 1;
        increment = 100 / totalPoints;
    }
    for (int arrayNum = startNum; arrayNum < endNum; arrayNum++)
    {
        int numLocalPoints = pointXlocal[arrayNum].Length;
        int numMaxPoints = pointXmax[arrayNum].Length;
        for (int i = 0; i < numLocalPoints; i++)
        {
            for (int j = i + 1; j < numMaxPoints; j++)
            {
                // long range pixel distances
                yDiff = pointYmax[arrayNum][j] - pointYlocal[
                    arrayNum][i]; // could try presetting the
                reference to point[arrayNum]
                xDiff = pointXmax[arrayNum][j] - pointXlocal[
                    arrayNum][i]; // if the compiler does not
                optimize enough
                sumDistResults[arrayNum][xDiff * xDiff + yDiff *
                    yDiff]++;
            }
            for (int j = i + 1; j < numLocalPoints; j++)
            {
                // local pixel distances
                yDiff = pointYlocal[arrayNum][j] - pointYlocal[
                    arrayNum][i]; // could try presetting the
                reference to point[arrayNum]
                xDiff = pointXlocal[arrayNum][j] - pointXlocal[
                    arrayNum][i]; // if the compiler does not
                optimize enough
                sumDistResultsLocal[arrayNum][xDiff * xDiff +
                    yDiff * yDiff]++;
            }
            loopNum++;
            if (loopNum % interval == 0)

```



```

        {
            if (OnProgressUpdate != null)
            {
                OnProgressUpdate(this, increment);
            }
            if (stopCalculations == true)
            {
                break;
            }
        }
    }
}
bool allThreadsFinished = false;
lock (ThisLock)
{
    radDistWorkerDoneCount++;
    if (radDistWorkerDoneCount >= radDistWorkerThreadCount)
        allThreadsFinished = true;
    if (radDistWorkerDoneCount > radDistWorkerThreadCount)
        System.Windows.Forms.MessageBox.Show("Error detected. _
        _Miscount_of_pixel_distribution_threads._
        Attempting_to_continue...");
}
if (allThreadsFinished)
{
    CalcRadialDistThreadsDoneLocal();
}
}
private void CalcRadialDistThreadsDoneLocal()
{
    // All threads have finished calculation, results are contained
    within sumDist[circleNum][sum square distances]
    bool objectStyle = true;
    if (radDistSettings.RadDistCalcStyle ==
        OptionsForm.AnalysisOptions.RadialDistributionSettings.
        RadialDistributionCalculationStyle.FromCircle)
    {
        objectStyle = false;
    }
    System.IO.StreamWriter outFile = null;
    System.IO.StreamWriter outFileLocal = null;
    try
    {
        string path;
        if (objectStyle)
            path = resultsPath + "\\Radial_Distribution_Results_
            Obj-Based";
        else
            path = resultsPath + "\\Radial_Distribution_Results_
            Circle-Based";
        System.IO.DirectoryInfo resultsDirectory = new System.IO.
        DirectoryInfo(path);
        if (!resultsDirectory.Exists)
            resultsDirectory.Create();
    }
    catch { }
}

```

```

int periodIndex = fileName.LastIndexOf(".");
if (periodIndex == -1)
    periodIndex = fileName.Length;
string resultsFileName = path + "\\\" + fileName.Substring
(0, periodIndex) + "(" + radDistSettings.LocalRadius.
ToString("0.00") +
    "-" + radDistSettings.MaxRadius.ToString("0.00") + ")_
    Long.txt";
string resultsFileNameLocal = path + "\\\" + fileName.
Substring(0, periodIndex) + "(" + radDistSettings.
LocalRadius.ToString("0.00") +
    "-" + radDistSettings.MaxRadius.ToString("0.00") + ")_
    Local.txt";
outFile = new System.IO.StreamWriter(resultsFileName);
outFile.WriteLine(resultsFileName);
if (objectStyle)
    outFile.WriteLine("Radial_Distribution_results_
        calculated_from_individual_objects_out_to_max_
        circle_radius_of_
        + radDistSettings.MaxRadius.ToString("0.000") + "_
        microns.");
else
    outFile.WriteLine("Radial_Distribution_results_
        calculated_from_a_local_circle_ +
        radDistSettings.LocalRadius.ToString("0.000") + "_
        microns_in_diameter_out_to_max_circle_radius_of
        _" +
        radDistSettings.MaxRadius.ToString("0.000") + "_
        microns.");
outFile.WriteLine("Image_Calibration:_\" + pixelsPerMicron.
ToString("0.000") + "_pixels_per_micron");
outFile.WriteLine(sumDistResults.Count.ToString() + "_
    objects_analyzed");
outFile.WriteLine();
System.Text.StringBuilder str = new System.Text.
    StringBuilder(7 * sumDistResults.Count);
str.Append("D,_pix\tD,_mic");
for (int i = 0; i < sumDistResults.Count; i++)
{
    str.Append("\tObject_" + objectCalcOrder[i].ToString()
        );
}
outFile.WriteLine(str.ToString());
// now call a method to combine the different distances
// and counts into one file with a single distance and
// multiple count columns
int numCircles = sumDistResults.Count;
bool foundDistance = false;
double distance;
int edgeEffectLimit;
if (objectStyle)
    edgeEffectLimit = (int)(radDistSettings.MaxRadius *
        radDistSettings.MaxRadius * pixelsPerMicron *

```

```

        pixelsPerMicron);
else
    edgeEffectLimit = (int)((radDistSettings.MaxRadius -
        radDistSettings.LocalRadius) * pixelsPerMicron *
        (radDistSettings.MaxRadius - radDistSettings.
            LocalRadius) * pixelsPerMicron);
for (int i = 1; i < edgeEffectLimit; i++)
{
    // iterate through all the sumSquares which are not
    // cut off by the edge effect
    foundDistance = false;
    for (int circleNum = 0; circleNum < numCircles;
        circleNum++)
    {
        if (sumDistResults[circleNum][i] > 0)
        {
            foundDistance = true;
            circleNum = numCircles;
        }
    }
    if (foundDistance)
    {
        str = new System.Text.StringBuilder(7 *
            sumDistResults.Count);
        distance = System.Math.Sqrt(i);
        str.Append(distance.ToString() + "\t");
        str.Append((distance / pixelsPerMicron).ToString()
            ); // Adds the distance measurement to the left
            column
        for (int circleNum = 0; circleNum < numCircles;
            circleNum++)
        {
            str.Append("\t" + sumDistResults[circleNum][i]
                .ToString());
        }
        outFile.WriteLine(str.ToString());
    }
}
if (objectStyle)    // output local data to new file
{
    outFileLocal = new System.IO.StreamWriter(
        resultsFileNameLocal);
    outFileLocal.WriteLine(resultsFileNameLocal);
    if (objectStyle)
        outFileLocal.WriteLine("Local_Radial_Distribution_
            results_calculated_from_individual_objects_out_
            to_max_circle_radius_of_
            + radDistSettings.MaxRadius.ToString("0.000")
            + "_microns.");
    else
        outFileLocal.WriteLine("Radial_Distribution_
            results_calculated_from_a_local_circle_ +
            radDistSettings.LocalRadius.ToString("0.000")
            + "_microns_in_diameter_out_to_max_circle_

```

```

        radius_of_" +
        radDistSettings.MaxRadius.ToString("0.000") +
        "_microns.");
outFileLocal.WriteLine("Image_Calibration:" +
    pixelsPerMicron.ToString("0.000") + "_pixels_per_" +
    "micron");
outFileLocal.WriteLine(sumDistResults.Count.ToString() +
    + "_objects_analyzed");
outFileLocal.WriteLine();
str = new System.Text.StringBuilder(7 * sumDistResults
    .Count);
str.Append("D, \tpix\tD, \tmic");
for (int i = 0; i < sumDistResults.Count; i++)
{
    str.Append("\tObject_" + objectCalcOrder[i].
        ToString());
}
outFileLocal.WriteLine(str.ToString());
// now call a method to combine the different
// distances and counts into one file with a single
// distance and multiple count columns
numCircles = sumDistResultsLocal.Count;
foundDistance = false;
for (int i = 1; i < sumDistResultsLocal[0].Length; i
    ++)
{
    // iterate through all the sumSquares which are
    // not cut off by the edge effect
    foundDistance = false;
    for (int circleNum = 0; circleNum < numCircles;
        circleNum++)
    {
        if (sumDistResultsLocal[circleNum][i] > 0)
        {
            foundDistance = true;
            circleNum = numCircles;
        }
    }
    if(foundDistance)
    {
        str = new System.Text.StringBuilder(7 *
            sumDistResultsLocal.Count);
        distance = System.Math.Sqrt(i);
        str.Append(distance.ToString() + "\t");
        str.Append((distance / pixelsPerMicron).
            ToString()); // Adds the distance
            // measurement to the left column
        for (int circleNum = 0; circleNum < numCircles
            ; circleNum++)
        {
            str.Append("\t" + sumDistResultsLocal[
                circleNum][i].ToString());
        }
        outFileLocal.WriteLine(str.ToString());
    }
}

```

```

        }
    }
}
catch (Exception ex)
{
    System.Windows.Forms.MessageBox.Show("Error in ImageCalc, unable to output file. \r\n" + ex.ToString());
}
finally
{
    if (outFile != null)
        outFile.Close();
}
sumDistResults = new List<uint[]>();
sumDistResultsLocal = new List<uint[]>();
if (OnUnLockAnalyzeMenu != null) // this event also spawns the ChildForm's AfterCalculatePixelDistribution method
{
    OnUnLockAnalyzeMenu(this, new EventArgs());
}
if (OnFinishedRadialDistCalc != null)
{
    OnFinishedRadialDistCalc(this, new EventArgs());
}
UnlockCalculation();
}
///<summary>
///Finds and stores the objects of the specified phase color on the image to an Objectlist. Runs on a dedicated worker thread.
///</summary>
public void FindObjects(Color phaseColor)
{
    if (!LockCalculation())
        return;
    Thread newThread = new Thread(delegate()
    {
        FindObjectsWorkerThread(phaseColor);
    });
    newThread.IsBackground = true;
    newThread.Name = "FindObjectsWorker";
    newThread.Start();
}
private void FindObjectsWorkerThread(Color phaseColor)
{
    theObjects = new ObjectList();
    theObjects.FindObjects(phaseColor, fastBitmaps[0]);
    if (fastBitmapsAlt.Count == 0)
        fastBitmapsAlt.Add(new FastBitmap(fastBitmaps[0].Bitmap));
    else
        fastBitmapsAlt[0] = new FastBitmap(fastBitmaps[0].Bitmap);
}

```

```

fastBitmapsAlt[0].DrawObjects(theObjects.GetObjects[Color.
    FromArgb(255, 255, 255)].ToArray(), true, Color.FromArgb(0,
    255, 0));
UnlockCalculation();
if (OnObjectsFound != null)
    OnObjectsFound(this, new FindObjectResults(theObjects.
        GetObjects[Color.FromArgb(255, 255, 255)].Count));
if (OnFinishedFindObjects != null)
    OnFinishedFindObjects(this, new System.EventArgs());
}
/// <summary>
/// Provides functionality for fast pixel access. Must use
    LockBitmap prior to pixel access and UnlockBitmap to end pixel
    access session.
/// </summary>
private class FastBitmap
{
    /// <summary>
    /// This object contains pointer addressable information about
        the bitmap
    /// </summary>
    private System.Drawing.Imaging.BitmapData bitmapData = null;
    /// <summary>
    /// The pointer to the upper left corner of the bitmap
    /// </summary>
    private byte* pBase = null;
    private Bitmap bitmap = null;
    private int numCol;
    private int numRows;
    private int numPixels;
    private bool bitmapLocked = false;
    public FastBitmap(Bitmap bitmap)
    {
        Bitmap b2 = new Bitmap(bitmap.Size.Width, bitmap.Size.
            Height, PixelFormat.Format24bppRgb);
        Graphics g = Graphics.FromImage(b2);
        g.DrawImage(bitmap, 0, 0, bitmap.Width, bitmap.Height);
        this.bitmap = b2;
        g.Dispose();

        numCol = bitmap.Width;
        numRows = bitmap.Height;
        numPixels = numCol * numRows;
    }
    /// <summary>
    /// Lock the bitmap so we can use the pointers to access it.
        Locking multiple times will have no additional effect.
    /// </summary>
    public void LockBitmap()
    {
        if (!bitmapLocked)
        {

```

```

        bitmapData = bitmap.LockBits(new Rectangle(0, 0,
            bitmap.Width, bitmap.Height), ImageLockMode.
            ReadOnly, bitmap.PixelFormat);
        bitmapLocked = true;
        pBase = (Byte*)bitmapData.Scan0.ToPointer();
    }
}
/// <summary>
/// Unlock the bitmap to end the pointer access session.
/// Unlocking multiple times will have no additional effect.
/// </summary>
public void UnlockBitmap()
{
    if (bitmapLocked)
    {
        bitmap.UnlockBits(bitmapData);
        bitmapData = null;
        pBase = null;
        bitmapLocked = false;
    }
}
/// <summary>
/// Get a pixel using unsafe code. Call only after locking
/// fastbitmap with LockBitmap.
/// </summary>
/// <param name="x">Column</param>
/// <param name="y">Row</param>
/// <returns>Color</returns>
public Color GetPixel(int x, int y)
{
    PixelData* p = PixelAt(x, y);
    return Color.FromArgb((int)p->Red, (int)p->Green, (int)p->
        Blue);
}
public Color GetPixelFromRowPtr(int rowPtr, int col)
{
    PixelData* p = PixelAtFromRowPtr(rowPtr, col);
    return Color.FromArgb((int)p->Red, (int)p->Green, (int)p->
        Blue);
}
}
/// <summary>
/// Set a pixel color using unsafe code. Call only after
/// locking fastbitmap with LockBitmap.
/// </summary>
/// <param name="x">Column</param>
/// <param name="y">Row</param>
/// <param name="c">Color</param>
public void SetPixel(int x, int y, Color c)
{
    PixelData* p = PixelAt(x, y);
    p->Red = c.R;
    p->Green = c.G;
    p->Blue = c.B;
}

```

```

}
/// <summary>
/// Get a pointer to a pixel
/// </summary>
/// <param name="x">Column</param>
/// <param name="y">Row</param>
/// <returns></returns>
private unsafe PixelData* PixelAt(int x, int y)
{
    return (PixelData*)(pBase + y * bitmapData.Stride + x *
        sizeof(PixelData));
}

private unsafe PixelData* PixelAtFromRowPtr(int rowPtr, int
    col)
{
    //return (PixelData*)(rowPtr + col * sizeof(PixelData));
    return (PixelData*)(rowPtr + col * sizeof(PixelData));
}
/// <summary>
/// Returns an int value to be supplied as a row pointer value
/// to PixelAt, not as the actual row value
/// </summary>
/// <param name="row"></param>
/// <returns></returns>
public int GetRowPtr(int row)
{
    return (int)pBase + row * bitmapData.Stride;
}
/// <summary>
/// Draws supplied objects to bitmap
/// </summary>
/// <param name="objects">Array of ObjectInfo instances to
draw</param>
/// <param name="outLineOnly">True to draw outlines, false to
draw entire objects</param>
/// <param name="color">Color to outline/fill objects with</
param>
/// <param name="displayNumber">Indicates whether to draw the
object numbers or not</param>
public void DrawObjects(ObjectInfo[] objects, bool outLineOnly
, Color color)
{
    bool initialState = bitmapLocked;
    if (bitmapLocked)
        UnlockBitmap();
    Pen pen = new Pen(color);
    SolidBrush brush = new SolidBrush(color);
    Graphics g = Graphics.FromImage(bitmap);
    List<Point> singlePoints = new List<Point>();
    if (outLineOnly)
    {
        for (int i = 0; i < objects.Length; i++)

```



```

    {
        if (objects[i].BorderPoints.Length > 2)
            g.DrawPolygon(pen, objects[i].BorderPoints);
        else if (objects[i].BorderPoints.Length == 2)
            g.DrawLine(pen, objects[i].BorderPoints[0],
                objects[i].BorderPoints[1]);
        else
            singlePoints.Add(objects[i].BorderPoints[0]);
    }
}
else
{
    for (int i = 0; i < objects.Length; i++)
    {
        if (objects[i].BorderPoints.Length > 2)
        {
            g.FillPolygon(brush, objects[i].BorderPoints);
            g.DrawPolygon(pen, objects[i].BorderPoints);
            // FillPolygon leaves some borders
            unfilled
        }
        else if (objects[i].BorderPoints.Length == 2)
            g.DrawLine(pen, objects[i].BorderPoints[0],
                objects[i].BorderPoints[1]);
        else
            singlePoints.Add(objects[i].BorderPoints[0]);
    }
}
g.Dispose();
LockBitmap();
foreach (Point p in singlePoints)
{
    SetPixel(p.X, p.Y, color);
}
if (!initialState)
    UnlockBitmap();
}
public void DrawNumbers(ObjectInfo[] objects, Color color,
    Font labelFont)
{
    bool initialState = bitmapLocked;
    if (bitmapLocked)
        UnlockBitmap();
    SolidBrush brush = new SolidBrush(color);
    Point textPoint;
    Graphics g = Graphics.FromImage(bitmap);
    for (int i = 0; i < objects.Length; i++)
    {
        textPoint = new Point(objects[i].ObjectBounds.Left +
            objects[i].ObjectBounds.Width / 2,
            objects[i].ObjectBounds.Top + objects[i].
                ObjectBounds.Height / 2);
    }
}

```

```

        g.DrawString((objects[i].ObjectNumber).ToString(),
            labelFont, brush, textPoint);
    }
    g.Dispose();
    if (initialState)
        LockBitmap();
}
/// <summary>
/// This struct is used to hold the RGB values when we find a
/// pixel using the pointer
/// </summary>
private struct PixelData
{
    public byte Blue;
    public byte Green;
    public byte Red;
}
/// <summary>
/// Get the Image we are working with
/// </summary>
public Bitmap Bitmap
{
    get
    {
        return bitmap;
    }
}
public int Width
{
    get
    {
        return numCol;
    }
}
public int Height
{
    get
    {
        return numRows;
    }
}
public int NumPixels
{
    get
    {
        return numPixels;
    }
}
}
/// <summary>
/// Stores a Dictionary of ObjectInfo Lists keyed to different
/// object colors. Encapsulates methods for finding and outlining
/// objects

```

```

/// from a FastBitmap.
/// </summary>
private class ObjectList
{
    Dictionary<Color , List<ObjectInfo>> objects;
    int numObjects = 0;
    public ObjectList()
    {
        objects = new Dictionary<Color , List<ObjectInfo>>();
    }
    public void FindObjects(FastBitmap fBitmap)
    {
    }
    public void FindObjects(Color objectColor , FastBitmap fBitmap)
    {
        if (!objects.ContainsKey(objectColor))
            objects[objectColor] = new List<ObjectInfo>();

        Point start = Point.Empty;
        Color color = Color.FromArgb(255, 255, 255); // white
        BoolBitmap pixelsChecked = new BoolBitmap(fBitmap);
        pixelsChecked.SetFalse();
        List<Point> allPoints = new List<Point>();
        fBitmap.LockBitmap();
        while (true) // find all objects
        {
            bool objectFound = false;
            while (true)
            {
                if (!pixelsChecked.Get(start.X, start.Y) &&
                    fBitmap.GetPixel(start.X, start.Y) == color)
                {
                    objectFound = true;
                    break;
                }
                start.X++;
                if (start.X >= fBitmap.Width)
                {
                    start.X = 0;
                    start.Y++;
                    if (start.Y >= fBitmap.Height)
                        break;
                }
            }
            if (!objectFound)
                break;
            Queue<Point> queue = new Queue<Point>(16);
            queue.Enqueue(start);
            int top = int.MaxValue;
            int bottom = int.MinValue;
            int left = int.MaxValue;
            int right = int.MinValue;
            allPoints.Clear();

```

```

while (queue.Count > 0) // fill one object
{
    Point pt = queue.Dequeue();
    int rowPtr = fBitmap.GetRowPtr(pt.Y);
    int localLeft = pt.X - 1;
    int localRight = pt.X;
    while (localLeft >= 0 && !pixelsChecked.Get(
        localLeft, pt.Y) && fBitmap.GetPixelFromRowPtr(
            rowPtr, localLeft) == color)
    {
        pixelsChecked.Set(localLeft, pt.Y, true);
        allPoints.Add(new Point(localLeft, pt.Y));
        --localLeft;
    }
    while (localRight < fBitmap.Width && !
        pixelsChecked.Get(localRight, pt.Y) &&
        fBitmap.GetPixelFromRowPtr(rowPtr,
            localRight) == color)
    {
        pixelsChecked.Set(localRight, pt.Y, true);
        allPoints.Add(new Point(localRight, pt.Y));
        ++localRight;
    }
    ++localLeft;
    --localRight;
    if (pt.Y > 0)
    {
        int sleft = localLeft;
        int sright = localRight;
        int rowPtrUp = fBitmap.GetRowPtr(pt.Y - 1);

        for (int sx = localLeft; sx <= localRight; ++
            sx)
        {
            if (!pixelsChecked.Get(sx, pt.Y - 1) &&
                fBitmap.GetPixelFromRowPtr(rowPtrUp, sx
                ) == color)
            {
                ++sright;
            }
            else
            {
                if (sright - sleft > 0)
                {
                    queue.Enqueue(new Point(sleft, pt.
                        Y - 1));
                }
                ++sright;
                sleft = sright;
            }
        }
        if (sright - sleft > 0)
        {

```

```

        queue.Enqueue(new Point(sleft, pt.Y - 1));
    }
}
if (pt.Y < fBitmap.Height - 1)
{
    int sleft = localLeft;
    int sright = localLeft;
    int rowPtrDown = fBitmap.GetRowPtr(pt.Y + 1);
    for (int sx = localLeft; sx <= localRight; ++sx)
    {
        if (!pixelsChecked.Get(sx, pt.Y + 1) &&
            fBitmap.GetPixelFromRowPtr(rowPtrDown,
            sx) == color)
        {
            ++sright;
        }
        else
        {
            if (sright - sleft > 0)
            {
                queue.Enqueue(new Point(sleft, pt.Y + 1));
            }
            ++sright;
            sleft = sright;
        }
    }
    if (sright - sleft > 0)
    {
        queue.Enqueue(new Point(sleft, pt.Y + 1));
    }
}
if (localLeft < left)
{
    left = localLeft;
}
if (localRight > right)
{
    right = localRight;
}
if (pt.Y < top)
{
    top = pt.Y;
}
if (pt.Y > bottom)
{
    bottom = pt.Y;
}
}
// Now outline object
Point[] borderPoints = OutlineObject(pixelsChecked,
start);

```

```

        objects[objectColor].Add(new ObjectInfo(allPoints.
            ToArray(), borderPoints, Rectangle.FromLTRB(left,
            top, right, bottom),
            numObjects + 1));
        numObjects++;
    }
    fBitmap.UnlockBitmap();
}
private Point[] OutlineObject(BoolBitmap bBitmap, Point start)
{
    List<Point> pts = new List<Point>();
    pts.Add(start);
    Rectangle bounds = new Rectangle(0, 0, bBitmap.Width,
        bBitmap.Height);
    Point last = new Point(start.X, start.Y - 1);
    Point curr = new Point(start.X, start.Y);
    Point next = curr;
    Point left = Point.Empty;
    Point right = Point.Empty;
    Point straight = Point.Empty;
    Point back = Point.Empty;
    while (true)
    {
        // trace object outline
        if (curr.Y < last.Y) // Up
        {
            left = new Point(curr.X - 1, curr.Y);
            right = new Point(curr.X + 1, curr.Y);
        }
        else if (curr.Y > last.Y) // Down
        {
            left = new Point(curr.X + 1, curr.Y);
            right = new Point(curr.X - 1, curr.Y);
        }
        else if (curr.X > last.X) // Right
        {
            left = new Point(curr.X, curr.Y - 1);
            right = new Point(curr.X, curr.Y + 1);
        }
        else // Left
        {
            left = new Point(curr.X, curr.Y + 1);
            right = new Point(curr.X, curr.Y - 1);
        }
        straight = new Point(curr.X + curr.X - last.X, curr.Y
            + curr.Y - last.Y);
        if (bounds.Contains(left) && bBitmap.Get(left.X, left.
            Y))
        {
            // turn left
            next = left;
        }
        else if (bounds.Contains(straight) && bBitmap.Get(
            straight.X, straight.Y))
        {
            // go straight

```

```

        next = straight;
    }
    else if (bounds.Contains(right) && bBitmap.Get(right.X
        , right.Y))
    {
        // turn right
        next = right;
    }
    else
    {
        // go back or end
        back = new Point(curr.X + last.X - curr.X, curr.Y
            + last.Y - curr.Y);
        if (bounds.Contains(back) && bBitmap.Get(back.X,
            back.Y))
        {
            // go back
            next = back;
        }
    }
    if (Math.Sign(next.X - curr.X) != Math.Sign(curr.X -
        last.X) ||
        Math.Sign(next.Y - curr.Y) != Math.Sign(curr.Y -
            last.Y))
    {
        pts.Add(curr);
    }
    last = curr;
    curr = next;
    if (next.X == start.X && next.Y == start.Y)
    {
        if (last.X == start.X + 1 && last.Y == start.Y &&
            bounds.Contains(start.X, start.Y + 1) &&
            bBitmap.Get(start.X, start.Y + 1))
        {
            // if you just came from the point to the
            // right of start then might have to add pixels
            // below yet
            // now need to keep going and eventually will
            // get back to start
        }
        else
        {
            break;
        }
    }
}
if (pts.Count > 1 && pts[0] == pts[1]) // prevents double
    counting of first point
    pts.RemoveAt(1);
return pts.ToArray();
}
public Dictionary<Color, List<ObjectInfo>> GetObjects
{
    get
    {
        return objects;
    }
}

```

```

    }
}
}
private class ObjectInfo
{
    Point[] allPoints;
    Point[] borderPoints;
    Color objectColor;
    Rectangle objectBounds;
    int objectNumber;
    public ObjectInfo(Point[] allPoints, Point[] borderPoints,
        Rectangle objectBounds, int objectNumber)
    {
        this.allPoints = allPoints;
        this.borderPoints = borderPoints;
        this.objectBounds = objectBounds;
        this.objectNumber = objectNumber;
    }
    public void ToXYArray(out ushort[] xValues, out ushort[]
        yValues)
    {
        xValues = new ushort[allPoints.Length];
        yValues = new ushort[allPoints.Length];
        for (int i = 0; i < allPoints.Length; i++)
        {
            xValues[i] = (ushort)allPoints[i].X;
            yValues[i] = (ushort)allPoints[i].Y;
        }
    }
    public int ObjectNumber
    {
        get
        {
            return objectNumber;
        }
    }
    /// <summary>
    /// Gets the smallest rectangle which entirely encapsulates
    /// the object
    /// </summary>
    public Rectangle ObjectBounds
    {
        get
        {
            return objectBounds;
        }
    }
    /// <summary>
    /// Gets the number of points within the ObjectInfo item
    /// </summary>
    public int NumPoints
    {
        get

```



```

        {
            return allPoints.Length;
        }
    }
    /// <summary>
    /// Gets the PointS array that stores all points contained
    /// within the ObjectInfo item.
    /// </summary>
    public Point[] AllPoints
    {
        get
        {
            return allPoints;
        }
    }
    /// <summary>
    /// Gets the PointS array that stores the points that make up
    /// the border of the ObjectInfo item. The border points are
    /// simply the
    /// outermost points of the item.
    /// </summary>
    public Point[] BorderPoints
    {
        get
        {
            return borderPoints;
        }
    }
    /// <summary>
    /// Gets or sets the color the object represents, should be
    /// related to the phase represented by the object.
    /// </summary>
    public Color ObjectColor
    {
        get
        {
            if (objectColor != null)
                return objectColor;
            else
                return Color.FromArgb(0, 0, 0);
        }
        set
        {
            objectColor = value;
        }
    }
}
/// <summary>
/// Encapsulates some Point functionality except storing the
/// values in shorts to save space
/// </summary>
public struct PointS
{

```

```

short xValue;
short yValue;
public PointS(short x, short y)
{
    xValue = x;
    yValue = y;
}
public static implicit operator Point(PointS p)
{
    return new Point(p.xValue, p.yValue);
}
public short X
{
    get
    {
        return xValue;
    }
    set
    {
        xValue = value;
    }
}
public short Y
{
    get
    {
        return yValue;
    }
    set
    {
        yValue = value;
    }
}
}
private class BoolBitmap
{
    bool[] theBools;
    int width;
    int height;
    int numPixels;
    /// <summary>
    /// Constructs a BoolBitmap of width and height equivalent to
    /// the supplied FastBitmap and initializes all values to false
    /// </summary>
    /// <param name="pairedBitmap"></param>
    public BoolBitmap(FastBitmap pairedBitmap)
    {
        this.width = pairedBitmap.Width;
        this.height = pairedBitmap.Height;
        this.numPixels = width * height;
        theBools = new bool[numPixels];
        for (int i = 0; i < numPixels; i++)
            theBools[i] = false;
    }
}

```

```

    }
    public void SetFalse()
    {
        for (int i = 0; i < numPixels; i++)
            theBools[i] = false;
    }
    public void SetTrue()
    {
        for (int i = 0; i < numPixels; i++)
            theBools[i] = true;
    }
    public void Set(int col, int row, bool value)
    {
        theBools[row * width + col] = value;
    }
    public bool Get(int col, int row)
    {
        return theBools[row * width + col];
    }
    public int Width
    {
        get
        {
            return width;
        }
    }
    public int Height
    {
        get
        {
            return height;
        }
    }
}
/// <summary>
/// Contains array of points that constitute a straight lines
/// between two points
/// </summary>
public class Line
{
    int mnPoints;
    Point[] mPoints;
    /// <summary>
    /// Line constructor, takes two points and determines a line between
    /// them
    /// </summary>
    public Line(Point pointOne, Point pointTwo)
    {
        int xDiff = pointOne.X - pointTwo.X;
        int yDiff = pointOne.Y - pointTwo.Y;
        Point difference = new Point(xDiff, yDiff);
        bool xAxisMax;
        // 29 right x 28 up

```

```

if (difference.X > 0)
    xDiff = difference.X;
else
    xDiff = -difference.X;

if (difference.Y > 0)
    yDiff = difference.Y;
else
    yDiff = -difference.Y;
int absDiff = System.Math.Max(xDiff, yDiff); // specifies number
    of pixels needed
mnPoints = absDiff;
    mPoints = new Point[mnPoints];
int maxDiff = System.Math.Max(difference.X, difference.Y);
// the number of pixels is equal to the greatest difference
// need to find out if the direction of maxDiff is positive or
    negative (which way to plot points)
// if difference x is biggest difference and x is positive then
    increment is negative
int i, increment;
double slope;
if (xDiff == absDiff)
{ // x direction determines number of pixels
    xAxisMax = true;
    if (xDiff == difference.X)
    { // if x difference is positive then increment is negative
        increment = -1;
    }
    else
    {
        increment = 1;
    }
}
else
{ // y direction determines number of pixels
    xAxisMax = false;
    if (yDiff == difference.Y)
    { // if y difference is positive then increment is negative
        increment = -1;
    }
    else
    {
        increment = 1;
    }
}
if (pointOne.X != pointTwo.X)
    slope = Convert.ToDouble(pointOne.Y-pointTwo.Y)/Convert.ToDouble
        (pointOne.X-pointTwo.X);
else
    slope = 1000000000;
for (i = 0; i<absDiff; i++)
{
    if (xAxisMax)

```

```

{
    if (slope < 0)
    {
        mPoints[i] = pointOne;
        mPoints[i].Offset(i*increment, (int)(Convert.ToDouble(i)*
            slope+0.5)*increment); // + new Point(i*increment,
            Convert.ToInt16(Convert.ToDouble(i)*slope+0.5)*increment)
        ;
    }
    if (slope >= 0)
    {
        mPoints[i] = pointOne;
        mPoints[i].Offset(i*increment, (int)(Convert.ToDouble(i)*
            slope+0.5)*increment);
    }
}
else
{
    if (slope < 0)
    {
        mPoints[i] = pointOne;
        mPoints[i].Offset((int)(Convert.ToDouble(i)*1/slope+0.5)*
            increment, i*increment);
    }
    if (slope >= 0)
    {
        mPoints[i] = pointOne;
        mPoints[i].Offset((int)(Convert.ToDouble(i)*1/slope+0.5)*
            increment, i*increment);
    }
}
}
}
}
/// <summary>
/// Gets number of points in a line
/// </summary>
public int Points
{
    get
    {
        return mnPoints;
    }
}
public Point GetPoint(int n)
{
    return mPoints[n];
}

public void ClipTo(int n)
{
    if (n < mnPoints)
        mnPoints = n;
}

```

```

}
/// <summary>
/// Represents a circle based upon a supplied diameter and
/// location. Circle points are stored in two arrays for the left
/// and right sides
/// of the circle and only one point per Y value is created per
/// side, thus the circle is meant to be iterated through left to
/// right then
/// top to bottom. If an odd diameter is supplied the next
/// highest even integer diameter is used to create the circle.
/// </summary>
private class Circle
{
    private Point[] leftPoints;
    private Point[] rightPoints;
    private int diameter = 0;
    private Point location = new Point(0, 0);
    public Circle(int diam, int x, int y)
    {
        if (diam % 2 != 0)
            diam++;
        leftPoints = new Point[diam + 2];
        rightPoints = new Point[diam + 2];
        int count = 0;
        int maxY = y + diam / 2;
        int rad = diam / 2;
        double radSqr = rad * rad;
        int xVal;
        for (int i = y - diam / 2; i <= maxY; i++)
        {
            //  $(x-x_0)^2 + (y-y_0)^2 = (d/2)^2$ 
            //  $x = ((d/2)^2 - (y-y_0)^2)^{1/2} + x_0$ 
            xVal = (int)(System.Math.Sqrt(radSqr - (i - y) * (i -
                y)) + 0.5);
            leftPoints[count].X = x - xVal;
            leftPoints[count].Y = i;
            rightPoints[count].X = x + xVal;
            rightPoints[count].Y = i;
            count++;
        }
        diameter = diam;
        this.location = new Point(x, y);
    }
    public Circle(int diam, Point location) : this(diam, location.
        X, location.Y)
    {
    }
    public void DrawCircle(FastBitmap fastBitmap)
    {
        for (int i = 0; i < this.diameter; i++)
        {
            fastBitmap.SetPixel(leftPoints[i].X, leftPoints[i].Y, Color.
                Green);
        }
    }
}

```

```

        fastBitmap.SetPixel(rightPoints[i].X, rightPoints[i].Y
            , Color.Green);
    }
    }
    /// <summary>
    /// Offsets the circle by a specified point
    /// </summary>
    /// <param name="offset"></param>
    public void Offset(Point offset)
{
    for (int i = 0; i <= diameter; i++)
    {
        leftPoints[i].Offset(offset.X, offset.Y);
        rightPoints[i].Offset(offset.X, offset.Y);
    }
    location.Offset(offset.X, offset.Y);
}
    public int Diameter
{
    get
    {
        return diameter;
    }
}
    public Point Center
{
    get
    {
        return location;
    }
}
    public Point[] LeftPoints
{
    get
    {
        return leftPoints;
    }
}
    public Point[] RightPoints
{
    get
    {
        return rightPoints;
    }
}
}
public class SpacingCalcInfo
{
    private double spacing;
    private int intercepts;
    private double distance;
    private int numLines;

```

```

public SpacingCalcInfo(double theSpacing, int theIntercepts, double
    theDistance, int theNumLines)
{
    spacing = theSpacing;
    intercepts = theIntercepts;
    distance = theDistance;
    numLines = theNumLines;
}
public double Spacing
{
    get
    {
        return spacing;
    }
}
public int Intercepts
{
    get
    {
        return intercepts;
    }
}
public double Distance
{
    get
    {
        return distance;
    }
}
public int NumberOfLines
{
    get
    {
        return numLines;
    }
}
}

public class PhaseFractionResults
{
    private double[] phaseFraction;

    public PhaseFractionResults(double[] phaseFraction)
    {
        this.phaseFraction = phaseFraction;
    }
    public double[] PhaseFractions
    {
        get
        {
            return phaseFraction;
        }
    }
}

```



```

public class CriticalDiameterResults
{
    private ArrayList circleDiameters;
    private ArrayList phaseFractions;
    public CriticalDiameterResults(ArrayList circleDiameters ,
        ArrayList phaseFractions)
    {
        this.circleDiameters = circleDiameters;
        this.phaseFractions = phaseFractions;
    }
    public ArrayList CircleDiameters
    {
        get
        {
            return circleDiameters;
        }
    }
    public ArrayList PhaseFractions
    {
        get
        {
            return phaseFractions;
        }
    }
}

public class FindObjectResults
{
    private int numberOfObjects;
    public FindObjectResults(int numberOfObjects)
    {
        this.numberOfObjects = numberOfObjects;
    }
    public int NumberOfObjects
    {
        get
        {
            return numberOfObjects;
        }
    }
}

/// <summary>
/// Burns objects and object details into unaltered bitmap based
/// on the supplied settings and returns the resulting bitmap.
/// </summary>
/// <param name="settings"></param>
/// <returns></returns>
public Bitmap GetDetailedBitmap(Image_Analyzer.ObjectDetailsForm.
    ObjectSettings settings , double zoom)
{
    if (fastBitmapsAlt.Count == 0)
        fastBitmapsAlt.Add(new FastBitmap(fastBitmaps[0].Bitmap));
    else
        fastBitmapsAlt[0] = new FastBitmap(fastBitmaps[0].Bitmap);
}

```

```

        bool outlineOnly = (settings.ObjectStyle == ObjectDetailsForm.
            ObjectStyle.Outline);
        bool displayNumbers = (settings.ObjectLabel ==
            ObjectDetailsForm.ObjectLabel.ObjectNumber);
        ObjectInfo[] objArray = theObjects.GetObjects[Color.FromArgb
            (255, 255, 255)].ToArray();
        fastBitmapsAlt[0].DrawObjects(objArray, outlineOnly, Color.
            FromArgb(0, 255, 0));
        Font zoomModFont = settings.LabelFont;
        if (displayNumbers)
            fastBitmapsAlt[0].DrawNumbers(objArray, Color.FromArgb
                (255, 0, 0), zoomModFont);
        return fastBitmapsAlt[0].Bitmap;
    }
    /// <summary>
    /// Get the Image we are working with
    /// </summary>
    public Bitmap Bitmap
    {
        get
        {
            return fastBitmaps[0].Bitmap;
        }
    }
    /// <summary>
    /// Returns the bitmap with lines on it from spacing calculation.
    /// Returns unaltered bitmap if no line bitmap exists.
    /// </summary>
    public Bitmap LineBitmap
    {
        get
        {
            if (fastBitmapsAlt != null && fastBitmapsAlt.Count > 0)
                return fastBitmapsAlt[0].Bitmap;
            else
                return fastBitmaps[0].Bitmap;
        }
    }
    /// <summary>
    /// Creates a new bitmap from the base bitmap and draws the
    /// objects to it
    /// </summary>
    public Bitmap ObjectBitmap
    {
        get
        {
            if (fastBitmapsAlt.Count == 0)
                fastBitmapsAlt.Add(new FastBitmap(fastBitmaps[0].
                    Bitmap));
            else
                fastBitmapsAlt[0] = new FastBitmap(fastBitmaps[0].
                    Bitmap);
        }
    }

```

```

        fastBitmapsAlt[0].DrawObjects(theObjects.GetObjects[Color.
            FromArgb(255, 255, 255)].ToArray(), true, Color.
            FromArgb(0, 255, 0));
        return fastBitmapsAlt[0].Bitmap;
    }
}
///<summary>
///Gets or sets a variable that determines whether the ImageCalc
///is to stop calculations at next available opportunity
///</summary>
public bool StopCalculations
{
    get
    {
        return stopCalculations;
    }
    set
    {
        stopCalculations = value;
    }
}
}
}

```

AnalysisOptions

This class stores settings for the different calculations available in ImageCalc.

```

// AnalysisOptions
public class AnalysisOptions
{
    private GeneralSettings generalSettings;
    private CalibrationSettings calibrationSettings;
    private PhaseSettings phaseSettings;
    private SpacingSettings spacingSettings;
    private CriticalDiameterSettings critDiamSettings;
    private RadialDistributionSettings radialDistSettings;
    public AnalysisOptions()
    {
    }
}
public class GeneralSettings
{
    private string resultsDirectory;
    public GeneralSettings(string resultsDirectory)
    {
        this.resultsDirectory = resultsDirectory;
    }
    ///<summary>
    ///Gets or sets the directory where analysis results will be
    ///stored to
    ///</summary>
}

```

```

    public string ResultsDirectory
    {
        get
        {
            return resultsDirectory;
        }
        set
        {
            resultsDirectory = value;
        }
    }
}
public class CalibrationSettings
{
    private string calName;
    private double calNumber;
    public CalibrationSettings(string calibrationName, double
        calibrationNumber)
    {
        calName = calibrationName;
        calNumber = calibrationNumber;
    }
    public string CalibrationName
    {
        get
        {
            return calName;
        }
        set
        {
            calName = value;
        }
    }
    public double CalibrationNumber
    {
        get
        {
            return calNumber;
        }
        set
        {
            calNumber = value;
        }
    }
}
/// <summary>
/// Contains information about phase settings, such as the names and
/// assigned colors of each phase
/// </summary>
public class PhaseSettings
{
    private ArrayList phaseNames = new ArrayList(15);
    private ArrayList phaseColors = new ArrayList(15);

```

```

private ArrayList phaseColorNames = new ArrayList(15);
public PhaseSettings()
{
}
public PhaseSettings(ArrayList phases, ArrayList colorNames,
    ArrayList colors)
{
    phaseNames = phases;
    phaseColorNames = colorNames;
    phaseColors = colors;
}
///<summary>
///Gets or sets an ArrayList of strings containing the names of
///each phase that has been defined
///</summary>
public ArrayList PhaseNames
{
    get
    {
        return phaseNames;
    }
    set
    {
        phaseNames = value;
    }
}
///<summary>
///Gets or sets an ArrayList of strings containing the names of
///each color defined for the phases
///</summary>
public ArrayList PhaseColorNames
{
    get
    {
        return phaseColorNames;
    }
    set
    {
        phaseColorNames = value;
    }
}
///<summary>
///Gets or sets an ArrayList containing Argb Colors for each
///phase that has been defined
///</summary>
public ArrayList PhaseColors
{
    get
    {
        return phaseColors;
    }
    set
    {

```

```

        phaseColors = value;
    }
}
///<summary>
///Returns number of phases assigned
///</summary>
public int Count
{
    get
    {
        return phaseNames.Count;
    }
}
}
public class SpacingSettings
{
    private int numLines;
    private bool showLines;
    public SpacingSettings(int numberOfLines, bool showTheLines)
    {
        numLines = numberOfLines;
        showLines = showTheLines;
    }
    ///<summary>
    ///Gets or sets number of lines to calculate, equals -1 if manual
    ///line determination is set with no line limit in place
    ///</summary>
    public int NumberOfLines
    {
        get
        {
            return numLines;
        }
        set
        {
            numLines = value;
        }
    }
    public bool ShowLines
    {
        get
        {
            return showLines;
        }
        set
        {
            showLines = value;
        }
    }
}
public class CriticalDiameterSettings
{
    CalcMethod calcMethod;

```

```

public CriticalDiameterSettings(CalcMethod calcMethod)
{
    this.calcMethod = calcMethod;
}
public enum CalcMethod
{
    GridCircles , RandomCircles
}
public CalcMethod CalculationMethod
{
    get
    {
        return calcMethod;
    }
    set
    {
        calcMethod = value;
    }
}
}
public class RadialDistributionSettings
{
    private double localRadiusMicrons;
    private double maxRadiusMicrons;
    private bool showCircles;
    private bool localAnalysis;
    private RadialDistributionCalculationStyle radDistCalcStyle;
    public enum RadialDistributionCalculationStyle
    {
        FromCircle ,
        FromObject
    }
    public RadialDistributionSettings(
        RadialDistributionCalculationStyle radDistCalcStyle , double
        localRadiusMicrons ,
        double maxRadiusMicrons , bool showCircles , bool localAnalysis)
    {
        this.radDistCalcStyle = radDistCalcStyle;
        this.localRadiusMicrons = localRadiusMicrons;
        this.maxRadiusMicrons = maxRadiusMicrons;
        this.showCircles = showCircles;
        this.localAnalysis = localAnalysis;
    }
    /// <summary>
    /// Gets or sets the style of radial distribution calculation ,
    /// indicating how the local area of calculation is determined.
    /// </summary>
    public RadialDistributionCalculationStyle RadDistCalcStyle
    {
        get
        {
            return radDistCalcStyle;
        }
    }
}

```

```

        set
        {
            radDistCalcStyle = value;
        }
    }
    /// <summary>
    /// Gets or sets the radius of the local area, in microns, where
    /// the radial distribution will be calculated from
    /// </summary>
    public double LocalRadius
    {
        get
        {
            return localRadiusMicrons;
        }
        set
        {
            localRadiusMicrons = value;
        }
    }
    /// <summary>
    /// Gets or sets the maximum radius, in microns, at which pixel
    /// distances will be calculated
    /// </summary>
    public double MaxRadius
    {
        get
        {
            return maxRadiusMicrons;
        }
        set
        {
            maxRadiusMicrons = value;
        }
    }
    /// <summary>
    /// Gets or sets whether to show circles used in radial
    /// distribution calculations
    /// </summary>
    public bool ShowCircles
    {
        get
        {
            return showCircles;
        }
        set
        {
            showCircles = value;
        }
    }
    public bool LocalAnalysis
    {
        get

```



```

        {
            return localAnalysis;
        }
        set
        {
            localAnalysis = value;
        }
    }
}
public GeneralSettings GetGeneralSettings
{
    get
    {
        return generalSettings;
    }
}
public CalibrationSettings GetCalibrationSettings
{
    get
    {
        return calibrationSettings;
    }
}
public PhaseSettings GetPhaseSettings
{
    get
    {
        return phaseSettings;
    }
}
public SpacingSettings GetSpacingSettings
{
    get
    {
        return spacingSettings;
    }
}
public CriticalDiameterSettings GetCriticalDiameterSettings
{
    get
    {
        return critDiamSettings;
    }
}
public RadialDistributionSettings GetRadialDistributionSettings
{
    get
    {
        return radialDistSettings;
    }
}
}

```