CHARACTERIZATION OF VISCOELASTIC MATERIALS THROUGH AN ACTIVE MIXER BY DIRECT-INK WRITING

By
ERIC DRAKE
Bachelor of Science in Electrical Engineering
Oklahoma State University
Stillwater, Oklahoma
2014

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
MASTER OF SCIENCE
May, 2017

CHARACTERIZATION OF VISCOELASTIC MATERIALS THROUGH AN ACTIVE MIXER BY DIRECT-INK WRITING

Thesis Approved:

Advisor - James Smay, Ph.D.

Outside Committee Member - Charles Bunting, Ph.D.

Committee Member - Pankaj Sarin, Ph.D.

Committee Member - Ranji Vaidyanathan, Ph.D.

ACKNOWLEDGEMENTS

To my wife, Melissa, and family for their unwavering love and support in the advancement of my studies – Without them, I would not be where I am today. Also, to the support of my advisor, Dr. Smay. His guidance and direction were pivotal not only to the success of this thesis, but to my growth as a researcher. To Lukasz and Yang, I'm thankful for all the assistance you've provided to me, I've learned much from you. To the faculty and students - thank you for fostering an excellent research group and facility to which I am both lucky and grateful to be a part of.

Name: ERIC DRAKE

Date of Degree: MAY 2017

Title of Study: CHARACTERIZATION OF VISCOELASTIC MATERIALS THROUGH AN ACTIVE MIXER BY DIRECT-INK WRITING

Major Field: MATERIALS SCIENCE AND ENGINEERING

Abstract: The goal of this thesis is two-fold: First, to determine mixing effectiveness of an active mixer attachment to a three-dimensional (3D) printer by characterizing actively-mixed, three-dimensionally printed silicone elastomers. Second, to understand mechanical properties of a printed lattice structure with varying geometry and composition.

Ober *et al* defines mixing effectiveness as a measureable quantity characterized by two key variables: (i) a dimensionless impeller parameter ($\widetilde{\Omega}$) that depends on mixer geometry as well as Peclet number (Pe) and (ii) a coefficient of variation (COV) that describes the mixer effectiveness based upon image intensity. The first objective utilizes tungsten tracer particles distributed throughout a batch of Dow Corning SE1700 (two parts silicone) - ink "A". Ink "B" is made from pure SE1700. Using the in-site active mixer, both ink "A" and "B" coalesce to form a hybrid ink just before extrusion.

Two samples of varying mixer speeds and composition ratios are printed and analyzed by microcomputed tomography (MicroCT). A continuous stirred tank reactor (CSTR) model is applied to better understand mixing behavior. Results are then compared with computer models to verify the hypothesis. Data suggests good mixing for the sample with higher impeller speed. A Radial Distrubtion Function (RDF) macro is used to provide further qualitative analysis of mixing efficiency.

The second objective of this thesis utilized three-dimensionally printed samples of varying geometry and composition to ascertain mechanical properties. Samples were printed using SE1700 provided by Lawrence Livermore National Laboratory with a face-centered tetragonal (FCT) structure. Hardness testing is conducted using a Shore OO durometer guided by a computer-controlled, three-axis translation stage to provide precise movements. Data is collected across an 'x-y' plane of the specimen.

To explain the data, a simply supported beam model is applied to a single unit cell which yields basic structural behavior per cell. Characterizing the sample as a whole requires a more rigorous approach and non-trivial complexities due to varying geometries and compositions exist. The data demonstrates a uniform change in hardness as a function of position. Additionally, the data indicates periodicities in the lattice structure.

TABLE OF CONTENTS

Chapter Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Figure Page

CHAPTER I: INTRODUCTION

1.1 Additive Manufacturing

Additive Manufacturing (AM) is a powerful manufacturing technique capable of constructing models layer-by-layer from computer aided design (CAD) software using a wide variety of materials such as ceramics [3, 4], polymers [5,6], cellular composites [7], thermoplastics [8], and metals [9]. The key feature of AM is that material is selectively added to build the part rather than subtracted from a block of feedstock. AM can be accomplished with a variety of approaches including: voxel-by-voxel techniques (e.g., stereolithography, laser-engineered net shaping (LENS), ink jet printing), filament deposition techniques (e.g., fused deposition, robocasting, direct-ink writing (DIW)), or complete layer-by-layer techniques (e.g., laminated object manufacturing (LOM), Carbon3D). Of these techniques, LENS, ink jet printing and DIW offer the best opportunity for multiple materials within the same print. The work described in this thesis utilized the DIW technique to build structures for two separate characterization studies.

DIW is an AM technique by which three-dimensional structures are assembled layer-by-layer by depositing feedstock material (ink) through a capillary nozzle of fixed internal diameter (d_{tip}) onto a substrate or previously deposited layers. The layers are of finite thickness (c.a., $\Delta z = d_{tip} \cdot \pi/4$), with the filament usually considered to have a round cross section with $d_{filament} \approx d_{tip}$. Motion controllers, motors, amplifiers, and actuators enable DIW to accurately trace a toolpath to deposit ink for precise construction of any given part. The toolpath is calculated via CAD software and is interpreted by the motion controller. Electrical signals are then sent to axis-specific amplifiers, motors, and actuators located on an x-y-z translation stage. The translation stage is thus capable of

movement in three dimensions, *x*, *y*, and *z*. Because the part is built in a vertical fashion (i.e. from $z=0$ to $z=(\#\text{ of layers})\cdot(\Delta z)$), the ink deposition nozzle is affixed to the *z*-axis of the gantry robot (see figure 1.2).

Toolpaths are a combination of two primary geometric objects: three-dimensional lines and two-dimensional arcs. Lines, represented by G1 in G-code syntax, are generated by scaling the velocity of one or more specific axes (i.e. *x*, *y*, or *z*) to trace a line from start to end point at a (typically) constant velocity along the line segment. Such control of multiple axes is known as contoured motion. For arcs, represented by G2 or G3 in G-code syntax, appropriate axes are moved to trace an arc such that the tangential velocity is constant. G-code is the language generated by CAD software and interpreted by the motion controller. A typical G-code file consists of an ordered set of G1, G2, and G3 motion commands coupled with print settings such as write speed and ink dispensing rate. The motion controller translates G-code files in a line-by-line fashion similar to other programming languages. Most controllers also allow variable definition, logic and loop structures (i.e., flow control) as well as subroutines within the G-code.

In DIW, syringe pumps are used to dispense ink as the robot traces the toolpath. Some DIW systems use constant pressure driven pumps if the ink rheology and pressure versus flow rate relationship is well-understood. A more reliable method is to use constant volume driven pumps, where the plunger motion of the syringe is precisely controlled by a linear actuator. The constant volume method is capable of accommodating changes in ink rheology and varying pressure drop at the deposition nozzle exit. In the work described in this thesis, constant volume syringe pumps are used.

From DIW, a wide array of promising applications have been demonstrated such as bone and tissue scaffolds [11, 12], composites [7, 13], piezoelectric sensors [13, 14], and soft robots [15]. A detailed

process flow diagram and model is provided in figures 1.1 and 1.2 to illustrate the basic components of a DIW system.

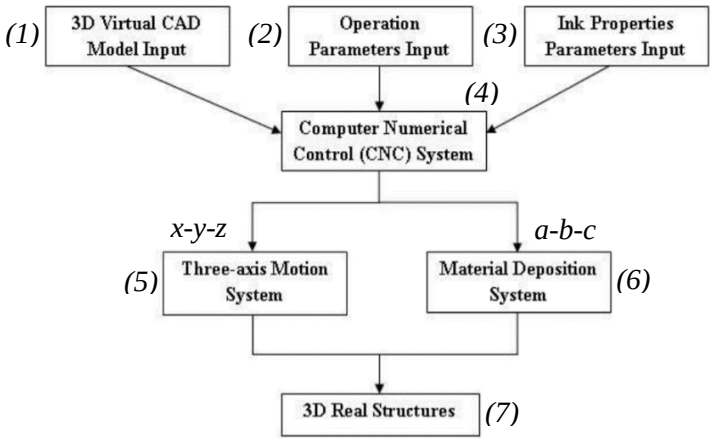


Figure 1.1: Flow Diagram of DIW Process [16]

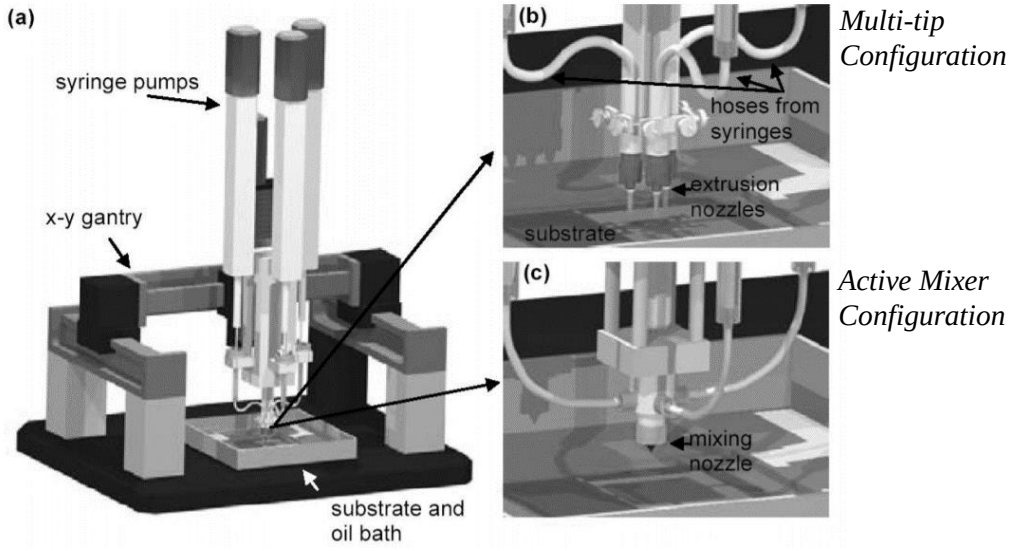


Figure 1.2: Schematic of a DIW System (b) robot overview, (c) multi-tip extrusion array, and (d) active mixer configuration [17]

Figure 1.1 illustrates the process flow of a DIW system. Three input operations, (1), (2), and (3), drive the Computer Numerical Control (CNC) System (4). The CNC system is comprised of the motion controllers and hardware. The first process, CAD model input (1), details model structural properties such as layer height, number of layers, and overall geometry and composition specification. Models are constructed by two methods: (i) free-hand design utilizing line drawing tools such as polyline and (ii) by import from a pre-existing design file (STL – Stereolithography File Format). Operational parameters (2) include write speed, syringe and tip diameters, and compositional values. From (1) and (2), all necessary calculations regarding distance, velocity, and acceleration for axis movement are computed. Processes (1), (2), and (3) are translated into G-code commands with which the CNC system (4) interprets into motion in all respective axes. This seven step process establishes all essential parameters for building 3D structures. Custom CAD software, RoboCAD, was tailored for this machine [18]. RoboCAD is discussed in-depth in section 3.2.3.1.

1.2 Ink

This thesis utilizes a reaction-cure, heat-activated silicone as the ink material. Silicone is a non-diene, elastomer used in many different applications such as electronics, construction, and mold-making [27, 28]. Elastomers are amorphous polymers in which the glass transition temperature, (T_g), is well below room temperature, yielding highly viscoelastic properties. For extrusion-based, 3D printing approaches, the ink must to be extruded and stabilized to maintain shape after the shear stress is removed. The silicone used in this thesis exhibits the shear-thinning properties necessary with a high viscosity, which is suitable for this approach.

1.3 Active Mixing

Blending two or more inks *in-situ*, gives the desirable ability to vary composition along the toolpath and, thereby, tailor various material properties (i.e., chemical, mechanical, electrical, etc.). Microfluidic mixers exist in two forms; passive and active [19]. Passive mixing occurs when two or

more fluids blend to form a homogenous product by no external forces [19] while active mixing utilizes outside forces to induce mixing such as acoustical [20], integrated microvalves/pumps [21], small impellers [22], or periodic variation of flowrates [23]. Passive microfluidic mixing nearly always suffers from low Reynolds number (Re) despite the use of low-viscosities (i.e., water, oil, etc.) fluids and depends on substantial contributions from diffusion and/or long flow paths with many twists and turns. For DIW, the viscosities are designed to be high for shape retention after deposition, further exacerbating the mixing problem. Hence, active mixing is required for DIW of compositionally graded toolpaths.

An active mixer may be viewed as a constantly stirred tank reactor or CSTR, where the residence time is ideally V/Q (V= reactor volume, Q=volumetric flow rate). For rapid composition change, the residence time of the mixer should be as small as feasible; however, for good mixing, the impeller should rotate some number of times per residence time in the CSTR.

In this work, two main questions were answered: First, given a very small active mixer with short residence time, how effectively can very viscous silicones be mixed to enable homogeneous composition of the extruded material? Second, if two silicones that, when cured, display disparate elastic modulus are blended in varying ratios along a toolpath, will volumetric mixing rules apply for the elastic properties at any point along the toolpath? As a corollary to the second question, can structural variation of the toolpath be combined with composition variation to assemble parts with highly variable elastic properties? These questions are formalized as the objectives of this thesis.

1.3.1 Mixing Effectiveness – Objective #1

The first study observed mixing effectiveness of an active mixer by X-Ray microtomography (microCT) through Direct-Ink Writing (DIW). From *Ober et al* [24], a coefficient of variation and dimensionless impeller parameter define good and poor mixing. This study aimed to verify these

mixing characteristic definitions. Samples were constructed in the regime of "good" mixing and analyzed by microCT to verify mixedness.

1.3.2 Mechanical Properties of Graded Lattice Structures – Objective #2

This portion of the thesis sought to explore mechanical behavior and properties of three-dimensionally printed, compositionally and geometrically graded lattice structures. Experiments were conducted using a durometer to measure hardness (resistance to indentation) across the samples. A simple supported beam model was applied to aid in understanding experimental results.

1.4 Significance

The importance of characterizing "mixedness" is two-fold: by actively mixing two or more materials at the microscale, one may customize and control mechanical, chemical, and electrical properties of printed structures. Secondly, limiting discussion of mixing effectiveness to two variables (coefficient of variation and a dimensionless impeller parameter) succinctly defines and describes active mixers. Establishing mechanical properties of graded structures requires application of a simple supported beam model for a single unit cell. The significance of this study is the ability to explore complex structures using simplified methods. Characterizing the entire sample requires an extremely rigorous mathematical approach thus highlighting the necessity of a simplified analysis. Further, tailored sample properties acquired during printing may be verified by specific property experiments, such as hardness tests by durometer.

1.5 Thesis Organization

This thesis is organized into six chapters. Chapter 1 is an introduction to additive manufacturing as well as a brief overview of the methodologies employed. Additionally, two hypotheses are formulated as objectives to be accomplished throughout the thesis. Chapter 2 entails background information necessary to achieve the objectives set in chapter 1 including mixing regimes, fluid dynamics, and viscoelastic materials. The third chapter details the employed methodologies for experimental work. Chapter 4 presents the results obtained from experimental work discussed in chapter 3. Chapter 5 provides a discussion on the results and conclusions drawn from the initial hypotheses. Supporting information is provided in the Appendices, Table of Figures, and List of Figures.

CHAPTER II: LITERATURE REVIEW

2.1 Introduction

Composition change on demand for accurate rendering of real-world objects such as human prosthesis requires ink blending in micro-fluidic systems as the next evolutionary step in 3D printing [30]. Current printers deposit material in a sequential fashion by a single ink or multiple inks at a time. While compositional variations are achievable through multi-material (i.e., multi-tip) printing, the process is arduous; ensuring all tips are aligned as well as the constant starting and stopping of ink flow between nozzles [30]. Passive and active micro-fluidic mixers are one such solution to attaining sharp compositional gradients along a toolpath. Utilizing one nozzle, mixers, passive or active, eliminate the need to properly align multiple tips and the continuous interruption of ink flow. For this discussion, passive mixing refers to mixers with no moving parts other than the fluids and active mixing refers to mixers with an externally driven impeller or paddle.

Currently, passive mixers for fluids in micro-channels rely on methods such as chaotic advection [30]. Chaotic advection occurs when fluids are stretched and folded arising from channel geometry complexities or unsteady flow streams [31, 32]. The Reynold's number (*Re*), the ratio of momentum to viscous forces, and Peclet (*Pe*), the ratio of convective to diffusive rates, number are two driving factors in passive and active micro-scale mixing. Reynold's number is a function of material density (ρ), mean velocity ($\bar{u}$), pipe diameter (d), and inversely proportional to viscosity (μ) (i.e., $Re = \frac{\rho u d}{\mu}$). Low *Re* correlates to a laminar flow regime, typically $Re < 2000$.

Conversely, high *Re* is indicative of turbulent flow. The Peclet number is calculated by: $Pe = \frac{Q}{dD}$, where *Q* is the volumetric flow rate and *D* is the diffusion coefficient. For efficient mixing, the ratio of channel dimensions (i.e., pipe length/pipe diameter) must be greater than *Pe* [24].

Difficulties arise when mixing yield-stress fluids at low *Re* (e.g., viscoelastic materials). The diffusion process is resisted in viscous substances because of large molecular activation energies. These substances, especially in micro-scale systems, exhibit strong laminar flow regimes. Generating turbulence to induce mixing is therefore highly desired. For viscoelastic materials (i.e., yield stress fluid), the Herschel-Buckley (HB) rheological model is applicable, $\tau = \tau_y + K\dot{\gamma}^b$, where τ_y is the yield-stress of initial motion, *K* is the consistency index, *b* is the power-law exponent, and $\dot{\gamma}$ is the characteristic shear rate. The HB model states that, for viscoelastic materials, a certain yield stress, (τ_y), is required to initiate flow. As the yield stress is increased, the viscosity of the fluid decreases (i.e., shear-thinning) in a non-linear fashion and conversely, a decrease in yield stress increases viscosity (i.e., shear-thickening). Herschel-Buckley reduces to Newtonian if *b*=1 and τ_y=0 or Bingham if *b*=1. Another factor in viscoelastic fluids is the Bingham number ($Bi = \frac{\tau_y}{\eta(\dot{\gamma})\dot{\gamma}}$) which emphasizes the importance of yield stress. As *Bi* approaches unity, the material flows as a solid plug and is not conducive to mixing. Increasing shear rate, however, drives fluidization of the material and thereby mixing efficiency.

2.1.1 Passive Mixing

Passive mixers induce chaotic advection usually by smoothed- or grooved-walled, Y- and T-type junctions [19, 24] (see figure 2.1). To ensure efficient mixing, the residence time (t_{res}) must exceed the mixing time (t_{mix}), both of which are dependent on channel geometry and volumetric flow rate. Without chaotic advection, passive mixers with small channel dimensions suffer from low Reynold's number, that is, fluids move in laminar motion and mixing must occur by diffusion [33, 34]. For highly viscous materials, mixing by diffusion operates on large timescales and channel

dimensions nonsensical for rapid composition switching. Therefore, passive mixing of viscoelastic inks (i.e., low Reynold's number) requires either complex mixer geometry or lengthy extrusion channels to induce mixing. Shear rates for static mixers are a function of volumetric flow rate and channel dimensions ($\dot{\gamma} = \frac{Q}{d}$). For better fluidization of materials, one must either increase channel dimensions, which are limited for the case of micro-fluids, or increase flow rate. Increased flow rates consequently create greater pressure drops in small channels - an undesirable side effect.

Several implementations of passive mixers have been characterized such as the staggered herringbone mixer (SHM), barrier embedded micro-mixer (BEM), and the three-dimensional serpentine channel (3D-SC) [35]. One particular application utilized the SHM to analyze mixing parameters and aid in mixer designs for several flow conditions [35]. The SHM is a rectangular channel lined with herringbone grooves to induce chaotic mixing. The advantage of an SHM is in its ease of fabrication and efficient mixing capabilities. However, with highly viscous materials, an alternative to the SHM is still desired. Active mixing is the second and preferred method of blending viscous inks.

2.1.2 Active Mixing

Active mixers use moving parts or external forces to overcome the chaotic advection limits of passive mixing [32]. The active mixer may induce shear rates far exceeding what is achievable by the small flow rates seen in micro-channels despite the intricacies of the static mixer design. The main advantage of an active mixer is the ability to establish high shear rates within small extrusion channels and simple mixer geometry by an active device (e.g., grooved impeller) thus reducing t_{mix} and allowing rapid compositional change times. Another advantage of active mixers is the ability to control when mixing occurs as well as the degree of mixedness at any given point. For an active mixer with rotating impeller, shear rate is only dependent on channel dimensions and impeller rotation rate (i.e., $\dot{\gamma} = \frac{\Delta\Omega}{\delta}$). With δ (see figure 2.1) constant, shear rate is thus controlled only by

rotational velocities, Ω. Faster rotation rates correlate to a more fluidized material, increased turbulent flow, and consequently, better mixing. Similar to passive mixers, the residence time must exceed the mixing time for efficient blending. Sharp compositional gradients depend on chamber dimensions, that is, a low ratio of $\frac{l}{d}$ is desired to reduce unwanted volumes of ink [24].

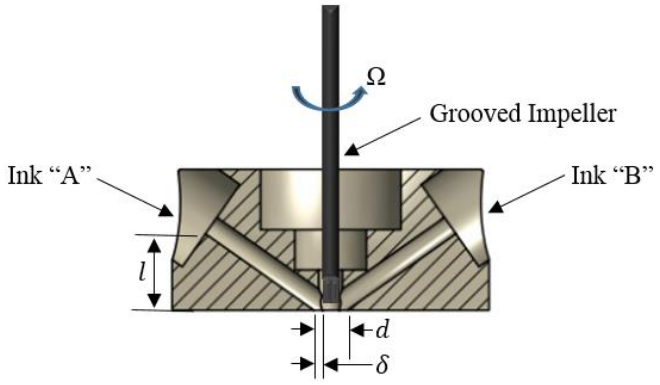


Figure 2.1: Mixing chamber cross-sectional view with impeller inserted [25]

Various active mixer schemes have been implemented into microfluidic systems [24, 34, 35, 36]. One particular mixer design used electrohydrodynamic (EHD) convection for biochemical analysis systems [36]. Two electrode plates were applied above and below a micro-fluidic channel with an electric field induced between the plates. The electric field generates a shear force at an interface layer between both fluids. Surface charges move with the fluid creating turbulent motion and mixing. Another mixer design from *Ober et al* sought to characterize and describe scaling relationships for comparing mixer efficiencies. Both Newtonian and yield-stress fluids were analyzed utilizing an active mixer print-head for direct-ink writing. The active mixer used a rotating impeller and various rotation rates to induce mixing just before deposition onto a substrate. Samples were subjected to imaging analysis to determine good and bad mixing regimes as a function of impeller rotational velocities. It was determined that good and bad mixing are strongly coupled with impeller rate and *Pe*.

This thesis sought to build upon the work of *Ober* by implementing an active mixing print-head of impeller design to further characterize mixing effectiveness as well as to study mechanical properties of graded geometric and compositional lattice structures.

2.2 Mixing Effectiveness

Ober et al states that mixing effectiveness is characterized by two key parameters: (i) a dimensionless impeller variable, ($\widetilde{\Omega}$), based upon the Peclet (*Pe*) number and mixer geometry (eq. 1.1), and (ii) a coefficient of variation (COV), derived from image intensity of a sample. Equation 1.1 states that ranges of good and poor mixing are determined by material properties and the mixer chamber dimensions.

$$\widetilde{\Omega} \geq \frac{1}{\alpha} Pe \left(\ln Pe - \ln(\alpha \frac{l}{d}) \right) \quad [1.1]^{[24]}$$

The dimensionless impeller value is defined as $\widetilde{\Omega} \equiv \frac{ld\Omega}{D}$ where *D* is the material diffusion coefficient, *d* is the diameter of the mixing chamber (figure 2.1), l is the chamber length, and Ω is the impeller's angular rotation rate [24]. By keeping $\frac{l}{d}$ and *Pe* constant, "mixedness" is then determined simply by $\widetilde{\Omega}$ as a function of angular rotation. Further descriptions of mixing effectiveness utilize a coefficient of variation to determine the statistical spread based upon sample imaging analysis, where COV is defined as $COV \equiv \frac{I_{std}}{I_{mean}}$. As opposed to describing mixedness by many terms, one may refer to the dimensionless impeller value or COV for insightful information and characterization of active mixers.

A continuously stirred tank reactor (CSTR) model was applied to better understand mixing behavior. Two inputs, ink "A" and ink "B", are deposited at varying rates into a mixing tank (figure 2.2). After a characteristic residence time, the tank enters a steady state of well-mixed contents. During the transient approach to steady state, both inks exit the tank partially mixed as new ink is

continuously deposited. Let ink "A" exhibit a certain volumetric flow rate, ($\dot{V}_1$), and composition, [C_1], and ink "B" a second flow rate, ($\dot{V}_2$), and composition, [C_2]. Therefore, the exiting product, ($\dot{V}_3$), is the sum of both volumetric flow rates (i.e. $\dot{V}_3 = \dot{V}_1 + \dot{V}_2$). The residence time, τ, is the time it takes to change from one compositional ratio to the next. For example, [C_1] and [C_2] are initially set to 25% and 75% respectively. The program issues a change of composition command to vary [C_1] to 50% and [C_2] to 50%. This change of composition occurs over the residence time. Because transition time is non-instantaneous, unwanted small volumes of ink are extruded during the change. Minimizing residence time is related to mixing chamber geometry, $\frac{l}{d}$ [24].

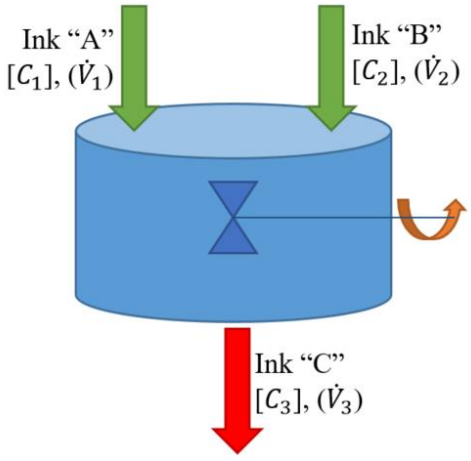


Figure 2.2: Continuous Stirred Tank Reactor (CSTR) diagram

This study implemented a material and active mixer of constant properties and dimensions to establish mixedness. With varying impeller rotational speeds, several samples were constructed in the regime of "good" mixing for imaging analysis.

2.3 Mechanical Properties of Graded Lattice Structures

The second objective of this thesis was to characterize mechanical properties of a compositionally and geometrically graded lattice structure comprised of two different silicone inks. When both inks display disparate elastic properties, it is important to determine if volumetric

mixing rules apply when varying ink ratios along a toolpath. Moreover, if, in addition to compositional gradients along a toolpath, the lattice is graded geometrically (i.e., raster spacing increases or decreases over a given length), how will elastic properties vary over increasing and decreasing geometries. Attaining compositional gradients was detailed in the previous sections, thus the subsequent sections will focus on how a geometrically graded lattice structure is built as well as what characterization techniques were used to answer the questions posed.

2.3.1 Graded Lattice Structures

For this thesis, the term "lattice" describes a three-dimensionally printed and repeated structure with rods of uniform spacing (i.e., road width) along an *x*-*y* plane (see figures 2.3 and 2.4). This pattern is repeated vertically until the desired height is achieved. Typically, every other layer is rotated or offset by some degree to generate various lattice patterns such as face-centered tetragonal (FCT), body-centered cubic (BCC), or hexagonally-closed packed (HCP).

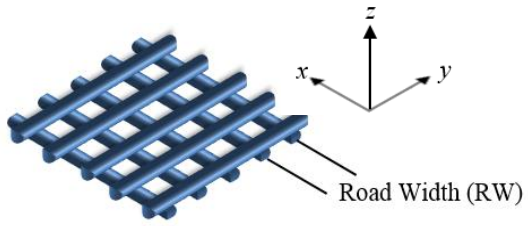


Figure 2.3: Two layer lattice structure with uniform road width and second layer rotated by 90°

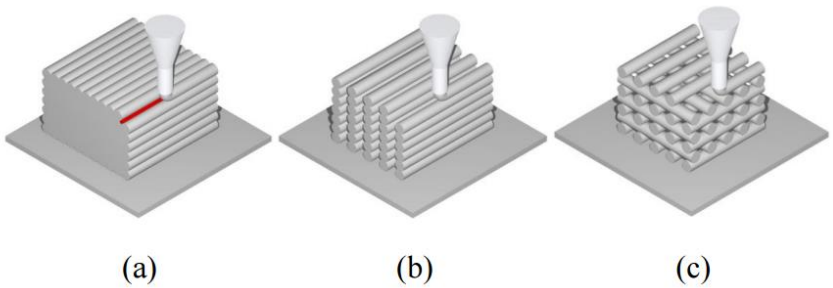


Figure 2.4: 3D lattice structures with (a) RW equal to rod diameter with no rotated layers, (b) RW greater than rod diameter with no rotated layers, and (c) RW greater than rod diameter with every other layer rotated by 90° [16]

Geometrically grading a lattice creates non-uniform road widths across a single layer with a starting pitch, ending pitch, and total box width. If starting pitch is less than the ending pitch, the lattice gradient increases from left to right and conversely, right to left for a greater ending pitch value. There are two methods by which lattice structures are graded: geometric and linear series. However, only the geometric series is discussed. Lattice design is described in the first chapter.

The goal here is to create a raster fill of a desired total width (W) where pitch (i.e., RW) varies linearly with position. There are three initial variables set by the user: starting pitch (p_{st}), ending pitch (p_{en}), and box width (W). A geometric series is applied to derive position and pitch for each raster line with *i* as the counting variable. Tables 2.1 and 2.2 detail the set of variables and equations used to obtain the final result:

Table 2.1: List of geometric series variables applied in Table 2.2

Position	**Width of Pattern *(Known)***	**Start Pitch *(Known)***	**End Pitch *(Known)***	**Pitch at Position '*x*'**	**Pitch Slope such that $p_i=p_{st}+mx_i$**
x	W	p_{st}	p_{en}	p	m

Table 2.2: Geometric series equations for starting pitch 0 to pitch N

$\underline{i}$	$\underline{x_i}$	$\underline{p_i}$
0	0	p_{st}
1	p_{st}	$P_{st}+x_1m \rightarrow p_{st}+p_{st}m \rightarrow p_{st}(1+m)$
2	$p_{st}+p_{st}(1+m) \rightarrow p_{st}[1+(1+m)]$	$p_{st}+mp_{st}+mp_{st}(1+m)$ $\rightarrow p_{st}(1+m)+mp_{st}(1+m)$ $\rightarrow p_{st}(1+m)(1+m) = p_{st}(1+m)^2$
3	$p_{st}+p_{st}(1+m)+ p_{st}(1+m)^2$ $\rightarrow p_{st}[1 +(1+m)+ (1+m)^2]$	$p_{st}+mp_{st}[1+(1+m)+(1+m)^2]$ $\rightarrow p_{st}(1+m[1+(1+m)+(1+m)^2])$ $p_{st}(1+m+m(1+m)+m(1+m)^2)$ $\rightarrow p_{st}(1+m)(1+m+m[1+m])$ $\rightarrow p_{st}(1+m)(1+m)(1+m) = p_{st}(1+m)^3$
4	$p_{st}[1+(1+m)+(1+m)^2+(1+m)^3]$	$p_{st}(1+m)^i$
N-1	$\sum_{i=0}^{N-2} p_{st}(1+m)^i$	$p_{st}(1+m)^{N-1}$
N	Geometric series: $\sum_{i=0}^{N-1} p_{st}(1+m)^i = p_{st}\frac{1-(1+m)^N}{1-(1+m)}$	$p_{st}(1+m)^N$

The geometric series used to produce pitch variation may not exactly match pitch to width, that is, the total length of the raster may exceed or fall short of box width. Therefore, an objective function (i.e., $\Omega = \alpha\left[\frac{(W-x_N)}{W}\right]^2 + \beta\left[\frac{(p_{en}-p_{N-1})}{p_{en}}\right]^2$) with weight factors ($\alpha$, β) is required to further match box width and pitch. Figure 2.5 illustrates the result of a graded lattice structure produced by the geometric series algorithm.

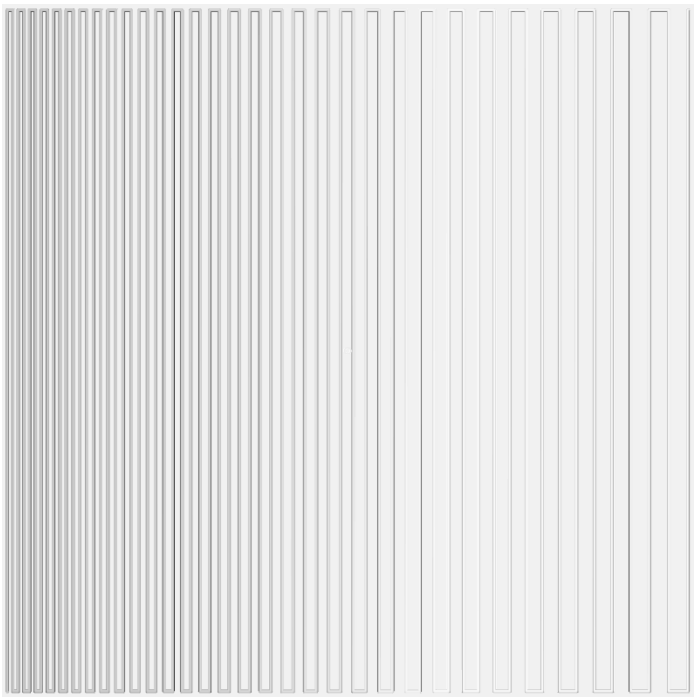


Figure 2.5: Graded lattice structure using the geometric series algorithm [18]

2.3.2 Intrinsic Properties of Silicone

Silicone is a synthetic polymer (i.e., elastomer) exhibiting viscoelastic properties. Viscoelasticity describes a material that exhibits both viscous and elastic behavior under deformation. The material returns to its original shape if stress is removed quickly, however permanently deforms over time. Similar to an elastic modulus (Young's modulus), viscoelastic materials are characterized by a dynamic modulus (i.e., $G^* = G_1 + iG_2$) containing both elastic (storage modulus – G1) and viscous (loss modulus – G2) component. Purely elastic materials exhibit in-phase stress and strain (i.e., strain in response to stress occurs instantaneously) while in purely viscous materials, strain lags stress by a 90° phase shift. The viscoelastic response falls between the purely elastic and purely viscous responses. The storage modulus represents the in-phase elastic component of deformation. Elastic materials 'store' and release energy upon stress and unloading respectively [37]. The loss modulus is important for dynamic mechanical testing where energy is dissipated through each oscillatory cycle. However, this study is only concerned

with static testing, therefore, the loss modulus is ignored. The elastic (storage) component is the main focus of this study.

Relaxation time is an important factor in characterizing viscoelastic materials. The relaxation time is defined as the measured stress over time while holding strain constant. For viscoelastic materials, an exponential reduction of stress occurs over the relaxation time and constant strain. Knowing relaxation time is important for testing a materials hardness. For example, if hardness tests (e.g., durometer indentation) were conducted before stresses achieved steady-state, measurements would vary wildly yielding erroneous results. Therefore, accurate results are best obtained after minimizing stress beyond relaxation time.

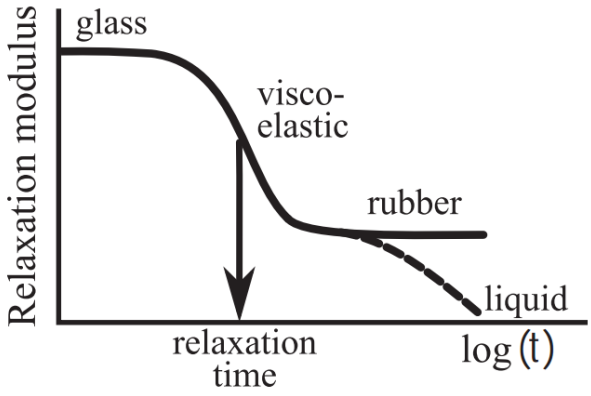


Figure 2.6: Relaxation modulus response over time [38]

2.3.3 Durometer

A durometer measures hardness by depth of indentation of a material. Durometers consist of a spring and indenter of a specific shape based upon Shore scale. Durometer Shore scale uses an alphabetic lettering system to group varying levels of hardness together and a subset of numbers (0-100) to further categorize specific materials.

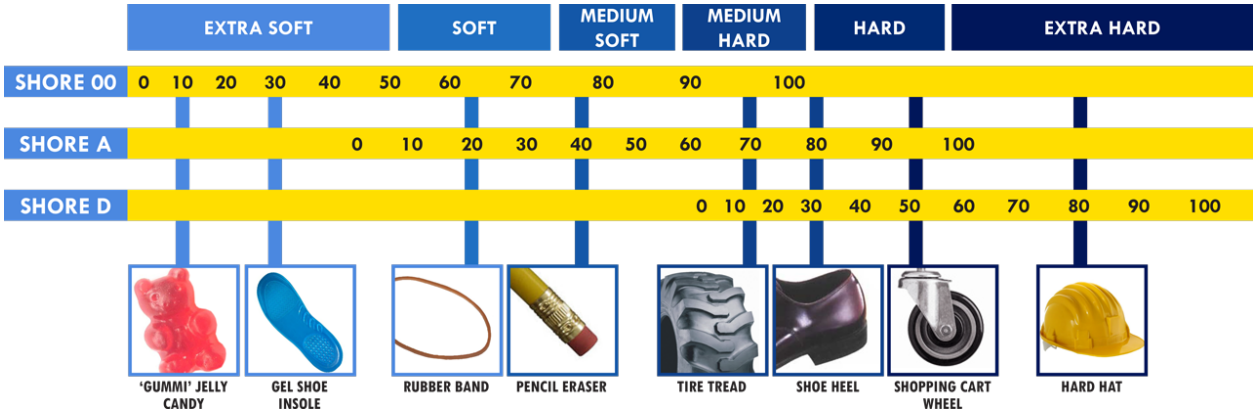


Figure 2.7: Durometer Shore scale [39]

Shore D 50, for example, describes hardness for a shopping cart wheel. The shopping cart wheel may also be characterized using other Shore letters as long as the corresponding object is within the 0-100 range. For softer materials, a more rounded indenter is used. Likewise, for harder materials, a sharper, more conical indenter is required. The indenter is attached to a spring which, when depressed in a material, retracts a specific distance relative to the material hardness. For this study, a durometer is used to measure hardness of a viscoelastic, geometrically and compositionally graded sample. Hardness tests provide insight into the samples elastic response. The goal is to characterize the sample through mechanical testing to understand how geometry and composition gradients affect elasticity.

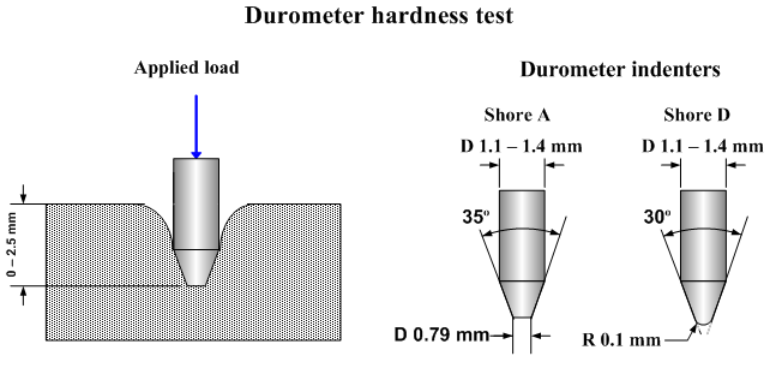


Figure 2.8: Indenter detail with Shore A and Shore D indenter styles [40]

CHAPTER III: METHODOLOGY

3.1 Introduction

This chapter focuses on three specific areas of experimental effort: *3.2 Printing Setup*, *3.3 Sample Fabrication*, and *3.4 Characterization Techniques*. First, *Printing Setup* is separated into three subsections: *3.2.1 SE1700 Preparation*, *3.2.2 Mixer Design*, and *3.2.3 Curing*. Next, sample fabrication briefly illustrates how parts are constructed through the DIW process using an active-mixing deposition nozzle. Further elaboration on sample design is provided in section 1.1.2. Finally, several characterization techniques (i.e., microcomputed tomography (MicroCT), flowrate experiments, and durometer) are detailed. The goal of this chapter is to provide the reader with adequate information to reproduce the results to be presented in Chapter 4 of this thesis.

3.2 Printing Setup

3.2.1 SE1700 Preparation

SE1700 (Dow Corning® SE1700) is a two-part, heat-cured polydimethylsiloxane elastomer consisting of a gel resin and catalyst. The catalyst is introduced in a 10:1 (gel:catalyst) mixture by weight [29]. Neat SE1700 ink (i.e., no additives) is compounded by blending the appropriate weights of gel and catalyst into a mixing cup and homogenized using a Thinky planetary mixer (Thinky AR-250; Thinky, Tokyo, Japan) for one minute and de-foamed an additional minute. For mixing efficiency experiments, 5-µm tungsten tracer particles (Tekna Advanced Material Inc. W-25; Quebec, Canada) at ~10 wt % are blended into a second batch of neat SE1700 using the Thinky

mixer to ensure uniform dispersal of tungsten particles. Finally, each ink is loaded into separate 10cc syringes and readied for printing.

3.2.2 Mixer Design

3.2.2.1 Mechanical

The active mixer used in these experiments is a custom assembly consisting of a drive motor, a mixing chamber, and check valves [25]. By impeller rotation, two streams of ink are actively mixed immediately before extrusion. Mixing is accomplished by a grooved impeller (driven by a DC motor) at the point of convergence of the two (or more) ink streams in the mixing chamber. A detailed schematic of the mixer assembly is provided in figure 3.1; illustrating the basic components. The diameter of the ink inlet channels is 1mm and the mixing impeller is fabricated from stainless steel rod stock of 1.19mm diameter. The estimated volume of the mixing chamber is 30-50 nL. The mixer body is made of PEEK plastic and inlet ports have a silicone rubber gasket to prevent leaks. The mixer assembly is capable of receiving four separate ink streams simultaneously; however, this work only utilized two ports. Additionally, specialized check valves are located inside each inlet port directly upstream of the mixing chamber. The check valves consist of a spring-loaded needle that interferes with the outlet orifice of the ink inlet tubing coming from the syringe(s) to the check valve. The check valves prevent unwanted backflow of ink (e.g., stream "A" bypassing the deposition nozzle and entering stream "B"). A triple O-ring seal inside the mixing chamber restricts ink flow from ascending the impeller shaft during extrusion. Figure 3.3 shows a more detailed view of the active mixer assembly in cross section. Both ink streams converge in the mixing chamber before being deposited as a single filament onto a flat ceramic substrate (ADS-96R, Coors Tek, Grand Junction, CO). The grooved impeller head, residing in the mixing chamber, rotates at a specified rotational speed to ensure uniform mixing. During printing, the mixer assembly is attached to the z-axis of the x-y-z gantry robot as pictured in Figure 3.4.

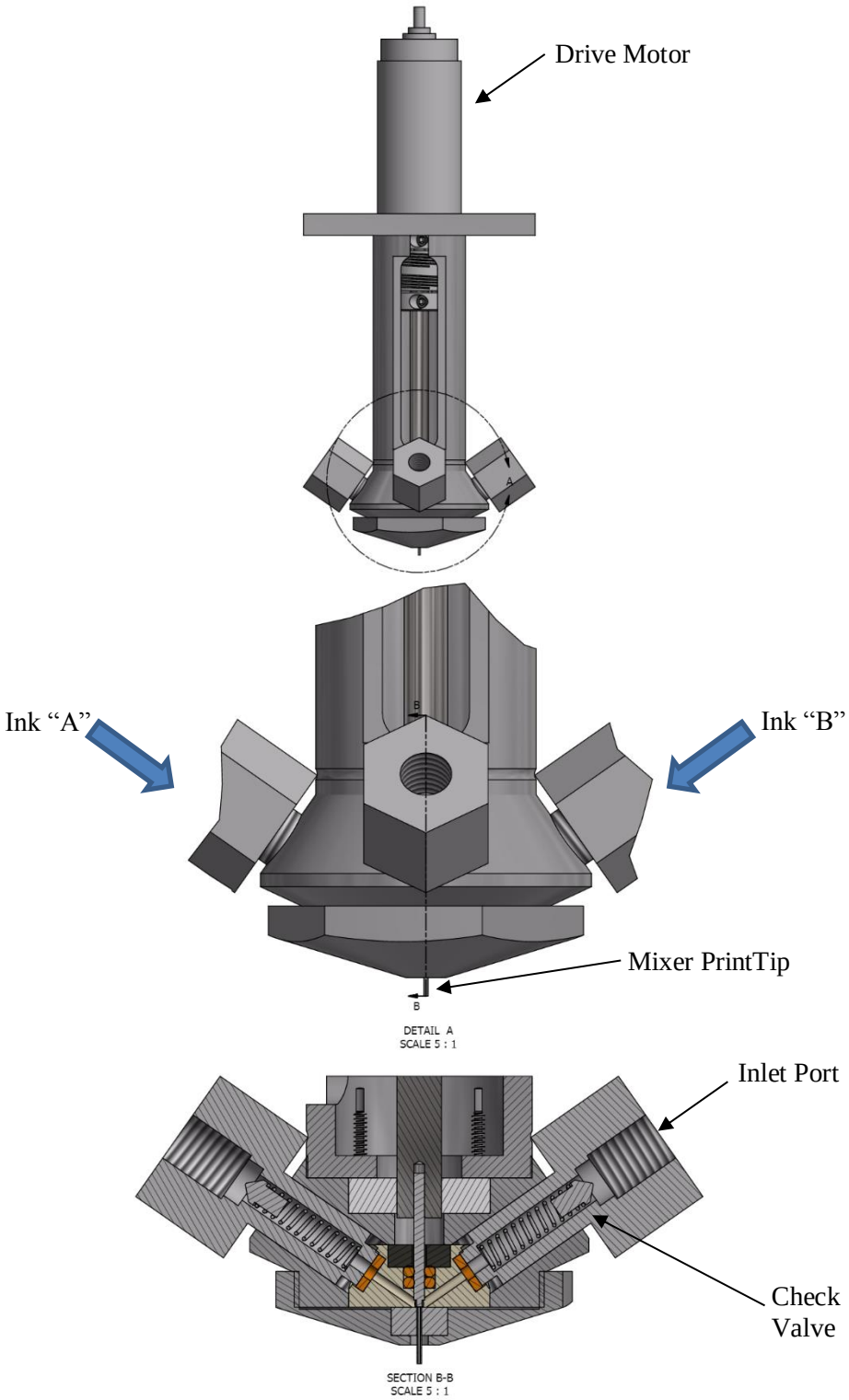


Figure 3.1: Schematic of mixer assembly

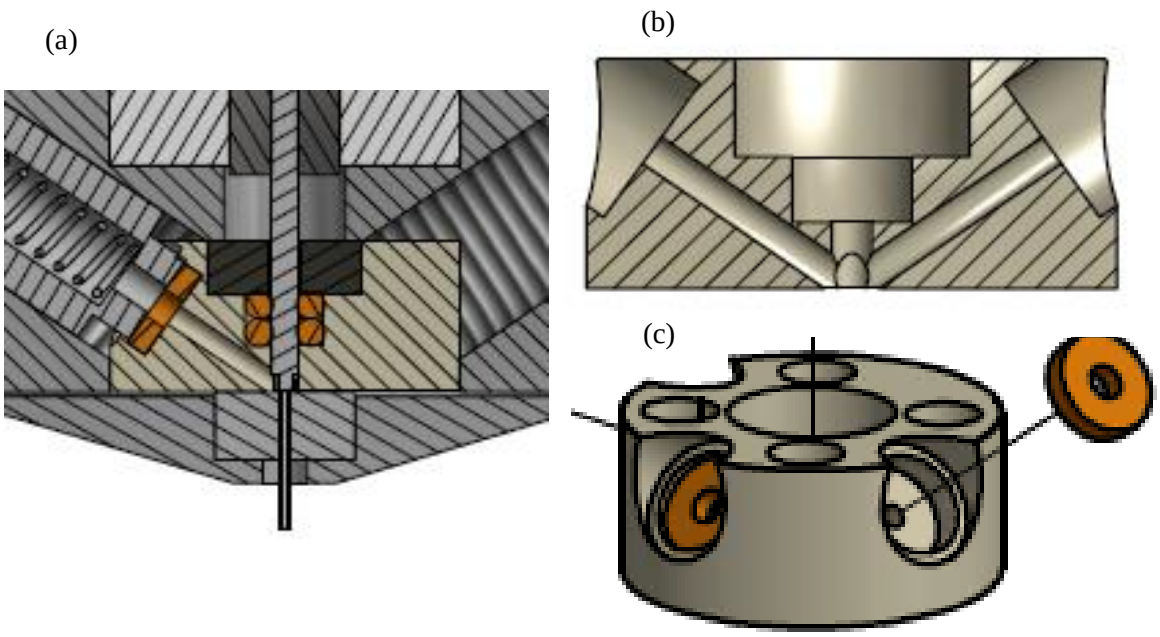


Figure 3.2: (a) Enlarged view of cross section B-B from figure 3.1. (b) Isolated mixer body in cross section, (c) Isolated mixer body showing inlet ports.

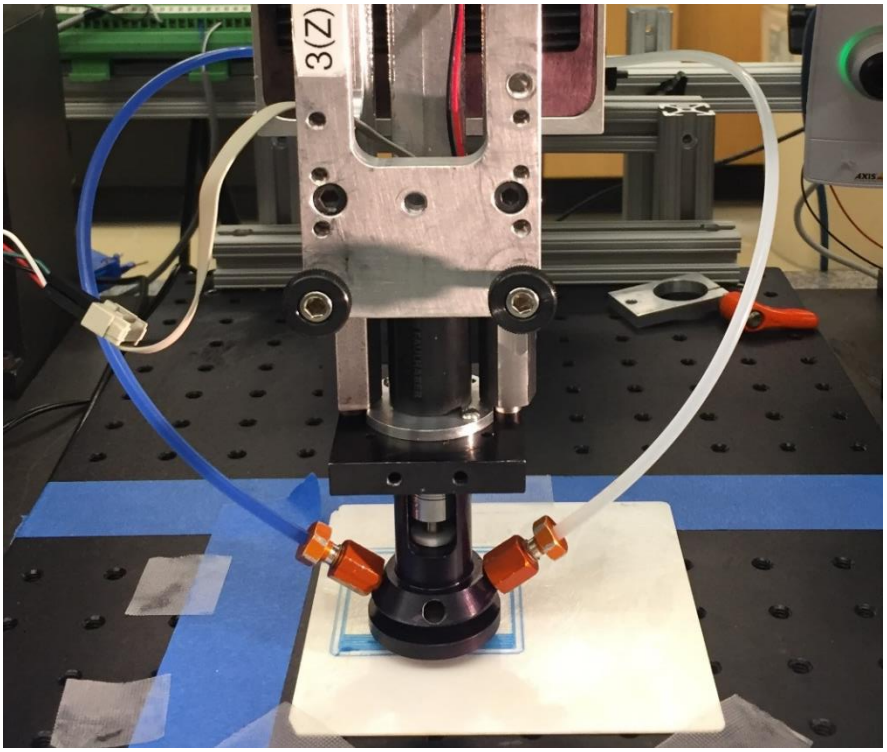


Figure 3.3: Digital image of mixer assembly printing and mixing two streams of ink

3.2.2.2 Electrical

Mixer speed is controlled through a computer interface. The intermediate electronic connections required to control mixer speed are shown schematically in Figure 3.4. The 12V DC motor (Faulhaber model 2342S012C-R) has a 512 line quadrature encoder (x4) and thus produces 2048 edge transitions per revolution of the motor. The rotational speed of the motor is regulated by a fast pulse width modulated (f-PWM) DC motor drive operating at a f-PWM frequency of 31 kHz. An Arduino Mega 2560 (using Amtel Mega 2560 microprocessor) controls the duty cycle of a PWM signal to a motor controller board (Pololu MC33926) that delivers current to the motor. The Arduino is powered and programmed through a universal serial bus (USB) connection. Motor control software is written in the Arduino integrated design environment (IDE), where properties such as velocity control, timer interrupts, and f-PWM setup are included. The C++ code for controlling the mixer speed through the Arduino is viewable in Appendix A. A proportional-integral-derivative (PID) control loop algorithm is implemented to alter the f-PWM duty cycle to achieve the motor speed specified by the user. An interrupt timer is assigned the task of checking motor feedback every 10μs (100 Hz). With each interrupt, the setpoint speed and actual speed values are compared. After each compare statement, the f-PWM duty cycle is adjusted to alter motor speed accordingly (i.e., accelerate or decelerate to match the setpoint value).

Both encoder channels, A and B, are connected to the Arduino serial communication pins (20 SDA and 21 SDC) respectively. A software counter is written to keep track of rotations provided by the motor encoder. The separate motor control board and motor are powered via a 12V, 30A power unit (SUPERNIGHT X-360-12 Switching Power Supply). All major software parameters (i.e., PID coefficients, RPM's, direction, etc.) are tunable through the serial monitor built into the Arduino IDE.

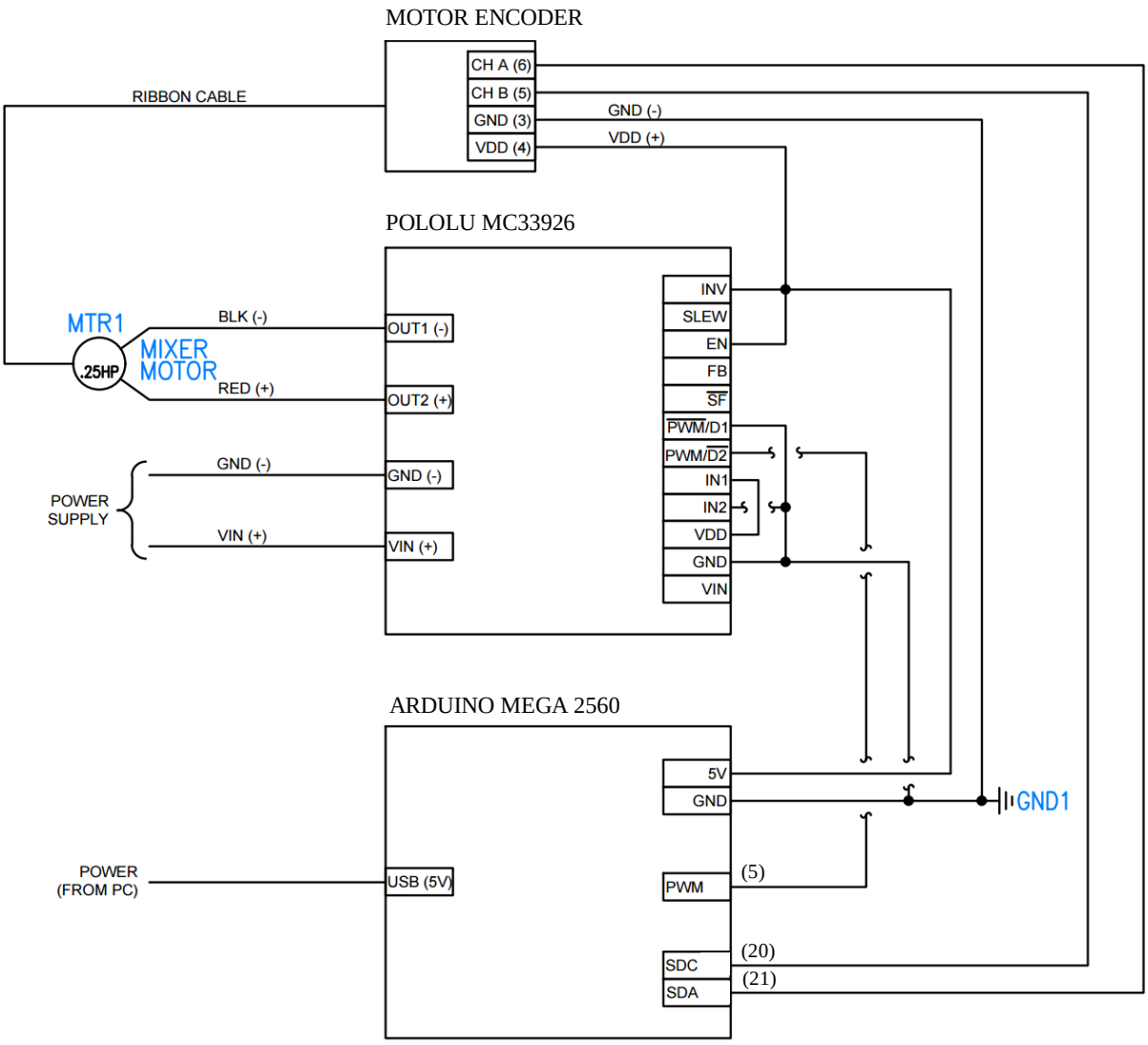


Figure 3.4: Wiring diagram of mixing motor electronics

The PID controller uses a negative feedback signal by continuously comparing a setpoint (e.g., target RPM) to a measured value (e.g, measured RPM). The difference between both variables yields an error term. It is the goal of the PID controller to minimize the error magnitude over time. Thus, if the measured RPM value is below the desired setpoint, a negative error is fed back to the controller to increase f-PWM duty cycle and match the setpoint. Likewise, if the measured RPM

output is faster than the setpoint, the motor will decrease f-PWM duty cycle to match the desired speed. The control loop is tunable by three coefficients: K_p, K_i, and K_d. The tuning coefficients aid in system stabilization by reducing overshoot and other oscillatory effects. Figure 3.5 illustrates the basic components of a PID system. From RoboCAD, a desired RPM value is set thus starting the motor. Motor direction is also controllable via software but is generally set to a clockwise (CW) rotation.

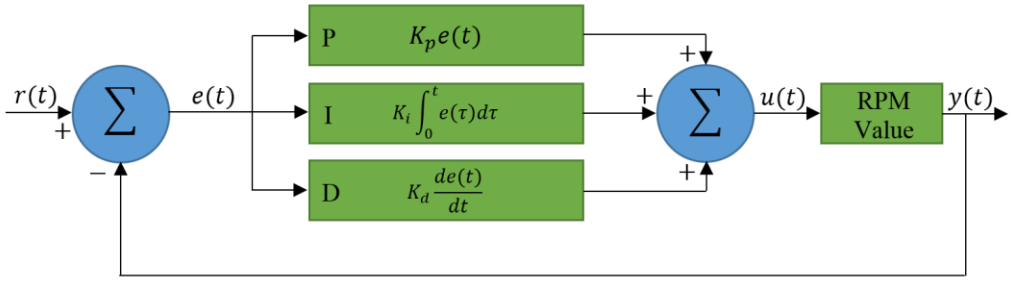


Figure 3.5: PID feedback control schematic

3.2.3 Sample Fabrication

Test samples are printed with discrete composition ratios of two-parts neat SE1700 (part A) and SE1700 laced with ~10 wt% tungsten tracer particles (part B). Details of the printing process are explained in the following section (i.e., section 3.2.3.1). Tungsten is chosen because of its dense atomic structure and good X-ray absorbance characteristics, thus yielding good visual results when subjected to X-ray anlaysis. A and B streams are mixed and extruded layer-by-layer onto a ceramic substrate until completion. Figure 3.8 and table 3.1 illustrates design layout and composition parameters of both samples. Samples 1 and 2 are then subjected to a heat-curing regime (90°C for 1 hour) yielding a final solid product. Mixing efficiency is imaged by X-ray micro-computed tomography and is discussed at length in section 3.3.5.

3.2.3.1 RoboCAD

RoboCAD is a custom software program for designing tool paths, setting print parameters, controlling the axes of the printer, and executing CNC/G-code programs. Figure 3.7 details the graphical user interface (GUI) as seen by the user complete with workspace, design options, and error and design output. As designs are composed, RoboCAD updates the visual representation (workspace) in real-time giving the user the ability to catch possible errors before the part is physically constructed.

Creating models in RoboCAD is done by one of two methods: The first method uses manual drawing tools such as polylines, lines, boxes, or arcs to create parameterized shapes. Models are defined layer-by-layer with layer thickness (Δz) being a function of d_{tip} (i.e., $\Delta z = f \cdot d_{tip} | \ f:[0,1]$). RoboCAD creates compound objects such as polylines, filled boxes and filled arcs, or filled polygons from geometric primitives (i.e., lines and arcs). In G-code, arcs are defined by two vectors: one is directed from the start point of the arc to the center point, the other is directed from the start point to the end point (see figure 3.6(b)). Note: two possible arcs could have the same two vectors, so G2 is used to specify a counterclockwise arc and G3 for a clockwise arc. Open and closed (e.g., polygons) polylines are defined by a set of points or nodes in a data structure. A polyline is created by defining a set of nodes where a straight line connects each node. A line (edge) is thus generated between both nodes and may be repeated until the desired effect is achieved (see figure 3.6).

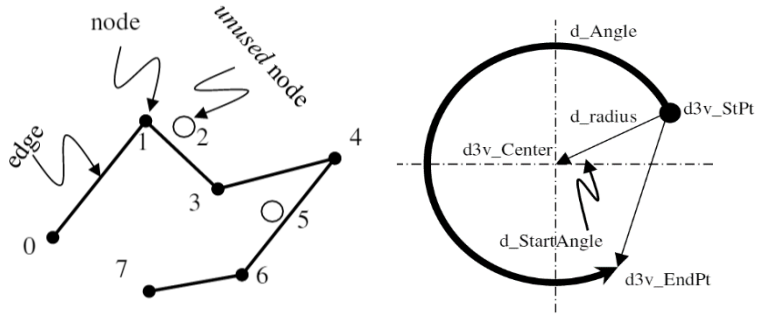


Figure 3.6: Polyline detail with notation of 'node', 'unused node', and 'edge' and (b) arc detail with defining center and end-point vectors

Material composition (i.e. volume percent of material for mixing) can also be defined within the polyline data structure using two approaches: node and edge. Composition is specified in RoboCAD as a normalized, *m*-component vector corresponding to fractional composition values for blending of up to *m* materials. The simplest form of composition vector is that of uniform composition (i.e. material composition is unchanging throughout the polyline). Compositional gradients are introduced by defining composition at both beginning and ending nodes. Edge compositions are uniform and may be manually set by *x*-*y* position or edge number.

To physically achieve compositional gradients, syringe pumps are accelerated and decelerated over a total time (t_f), using a set number of constant time increments (n), and with initial and final velocities ($\dot{x}_0$ and $\dot{x}_{tf}$ respectively). With each time-step increment, velocity changes linearly by $\dot{x}_{n-1} = \ddot{x}\delta t + \dot{x}_{n-2}$ where $\delta t = \frac{t_f}{n}$ and ẍ is acceleration. Acceleration is then defined as $\ddot{x} = \frac{\dot{x}_{t_f} - \dot{x}_0}{t_f}$. While inlet stream flowrates may vary to accommodate composition alterations, the output flowrate must remain constant to meet the deposition rate required by the toolpath.

The second method of model creation is to import tessellated, three-dimensional geometry files (e.g., STL files), which are then subjected to slicing and filling processes. Slicing consists of intersecting the STL file with a set of planes parallel to the substrate and coincident with the desired

layers to be printed. This intersection creates a set of perimeter polygons that must be ordered to create positive (typically CCW oriented polygon) and negative (typically CW oriented polygon) areas. After defining the perimeter polygons, a toolpath to fill the perimeter is generated by either: *i*) calculating a set of offset polygons of fixed distance from the perimeter (i.e., contour filling), *ii*) calculating a set of parallel lines that span the positive areas of the defining perimeter (i.e., raster filling), or *iii*) a combination of *i*) & *ii*). Figures 3.8(a)-(d) illustrate slicing and filling by contour and mixed (i.e., combination of *i* & *ii*) methods. One may also add supporting structures to fill unsupported parts of the STL geometry. Support structures are created using the same methods for slicing and filling.

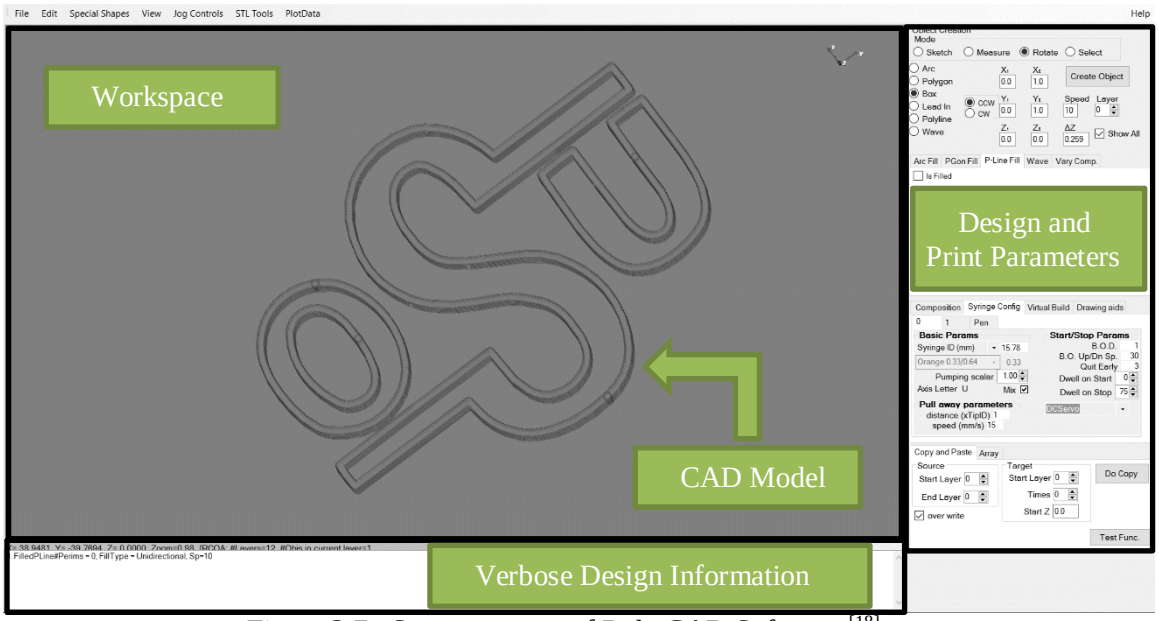
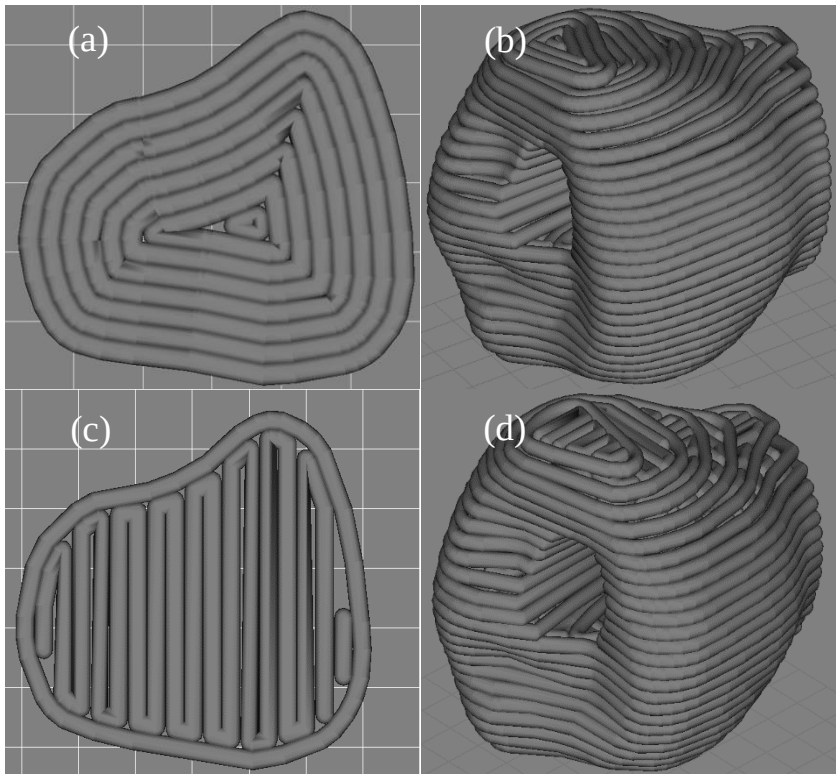


Figure 3.7: Screen capture of RoboCAD Software [18]



Figures 3.8: (a) Cross-sectional view of one layer within a pattern by contour, (b) the entire pattern after slicing and filling, (c) cross-sectional view of one layer using mixed method, and (d) the entire pattern sliced and filled

Table 3.1: Table of sample attributes as used in mixing efficiency study

Sample #	***Layers***	***Composition Ratio***	***Impeller Velocity (in RPM)***	***Dimensions (in mm)***	***Filament Diameter (in mm)***	***Δz (in mm)***	***# of rims***	***Pitch (in mm)***
1	14	[75/25]	1000	25.4 x 25.4 x 3.6	0.330	0.259	1	0.35
2	14	[50/50]	2000	25.4 x 25.4 x 3.6	0.330	0.259	1	0.35

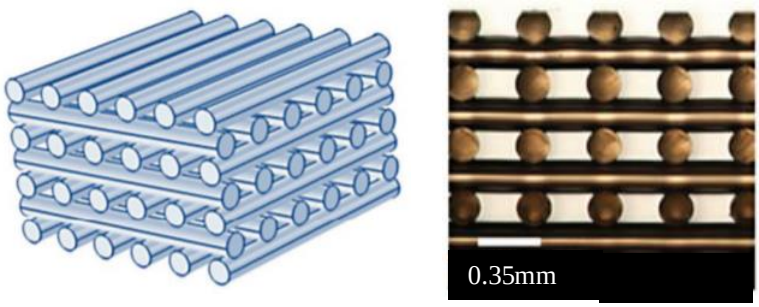


Figure 3.9: (a) gives an isometric view of a Simple Cubic (SC) lattice with (b) illustrating a side view and pitch width [41]

3.3 Experimental and Characterization Techniques

3.3.1 Flowrate Validation

The SE1700 inks used in this work are highly viscous, non-Newtonian fluids that are pumped through a length (~200mm) of polypropylene tubing (i.d.≈2mm) from a syringe reservoir into a nL mixing chamber and out through an extrusion nozzle (d_{tip}=337μm) onto either a rigid substrate or un-cured SE1700 in previous layers. Hence, predicting pressure drop as a function of flow rate is complicated by all possible process variations and positive displacement syringe pumps are used rather than constant pressure driven syringes. Assuming incompressible fluids and rigid mechanical connections, the flow rate from a syringe should equal the cross sectional area of the syringe times the plunger speed. Therefore, both positive displacement pumps are characterized prior to the mixing experiment.

First, a batch of neat SE1700 is loaded into two separate syringes and mounted for printing. The ink streams are plumbed through the tubing to independent 0.337mm printing tips. Each plunger is given a speed based on composition ratios (i.e., 2:1, 1:2, etc.). For example, a 50/50 (1:1) ratio has both plungers extruding material at the same rate. A 2:1 ratio doubled the rate of the first plunger over the second (i.e., if plunger one is moving at 0.02mm/s, plunger two is set to 0.01mm/s). Both materials are then extruded for one minute to achieve a steady flow state. Empty weigh boats

are then placed below each tip and filled for three minutes. Once filled, the collected ink is weighed to determine the average mass flow rate from each syringe. With a 2:1 flowrate ratio, for example, the amount of material extruded from the first plunger should double that of the second plunger.

3.3.2 Rheology

The rheology of neat and tungsten particle laden SE1700 is characterized with a controlled stress-strain rheometer (Bohlin Instruments, Model C-VOR200, East Brunswick, NJ) using a cone and plate (d=40mm with an angle of 4°) measurement geometry under both oscillatory and steady shear conditions. For oscillatory measurements, shear stress (τ) is increased from 20 to 1800Pa at an angular frequency of ω = 1Hz. Steady shear viscometry is performed by increasing shear rate ($\dot{\gamma}$) from 0.0005 to 1800(1/s) over ~40 minutes.

3.3.3 Scanning Electron Microscopy (SEM)

A field emission scanning electron microscope (FE-SEM) (Hitachi S-4800, Santa Clara, CA) is used to verify tungsten particle size as well as particle size distribution. Graphite Conductive Adhesive (Electron Microscopy Sciences, Fort Washington, PA) is applied to the SEM stage and coated with a thin, even layer of tungsten particles. Tungsten exhibits good conductivity so gold sputtering is not necessary for a working sample. One coating of tungsten is analyzed at various magnifications to ascertain sizing information. Particle sizing and distribution is done via ImageJ (Wayne Rasband, National Institute of Health).

3.3.4 Particle Size Distribution

ImageJ is used to determine size distributions. Forty-eight random particle diameters from figure SEM images are estimated manually using the line tool. Given the data set, diameter values are sorted into four diameter size groups (i.e., 0-5μm, 5-10μm, 10-15μm, and 15-20μm). Diameter

values are initially given in pixels but are converted to μm by measuring the scale bar on the SEM micrograph and adjusting accordingly.

3.3.5 Durometer

Sample mechanical properties are obtained by durometer (CheckLine, Model DD-100-OO, Cedarhurst, NY). A durometer extrapolates the value of material hardness by measuring depth of presser foot indentation. Durometer testing is certified under the American Society for Testing and Materials (ASTM) (i.e., ASTM D2240) [42]. The durometer utilizes the Shore scale as a representation of material hardness. For example, the durometer used in this thesis is a Shore OO, indicating its use for soft materials such as gels. Scale hardness ranges from OOO-S to A, where A is applicable towards the hardest materials (e.g., rubber tires, thermoplastic elastomers, etc.). For softer materials, the presser foot (indenter) is rounded so as to not penetrate the sample. Accurate results require that the presser foot be completely submerged within the sample until the metallic base makes contact (see figure 3.10(b)).

Three samples are constructed (50mm x 50mm x 5.8mm) using the active mixer with a tip size of d_{tip}=0.337mm. Samples consisted of various colors as well as graded geometries (see section 2.3.1.1 for a more detailed explanation on grading geometries) given in table 3.2. A custom mounting attachment is assembled allowing the durometer to be affixed against the printers Z-axis gantry. Additionally, a serial cable, attached from durometer to computer, provided necessary communication to integrate custom testing software. A program is written to move the durometer in the X and Y direction while taking measurements at set millimeter increments. It is important to properly place and align the samples for consistent data collections. Samples are placed on a flat ceramic substrate and aligned with one sample edge following the printer X-axis. Alignment is conducted using a Dial Test Indicator (Mitutoyo Series 513-Horizontal Type) to ensure the X-axis matched both sample and printer. The indicator is temporarily mounted to the Z-gantry and moved

laterally along the X-axis. For this data, the durometer took readings every 1mm in X and Y starting from (0,0) (see figure 3.11). The durometer begins at (0,0), moves laterally towards (50,0), moves longitudinally to (50,1) and laterally back to (0,1). At each 1mm step, the durometer descends 3mm vertically to collect measurements for a half-second and ascends back to Z=0 before moving to the next point. This process is repeated until the durometer reaches (50,50). An array of 2500 data points is collected across each sample. The data is visualized with a surface plot using Matlab where durometer hardness is plotted as a function of (X,Y) position. Durometer measurements are conducted once per sample. For future work, a statistical analysis will be done with several durometer tests to derive variance and standard deviation of the data.

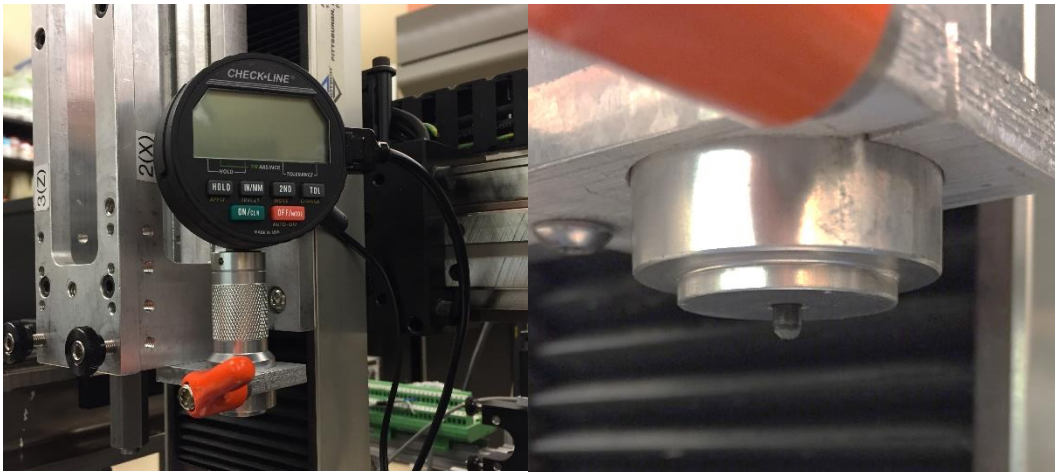


Figure 3.10: Digital image of (a) the durometer mounted to the Z-axis gantry and (b) a close-up of the durometer presser foot (indenter)

Table 3.2: Table of sample attributes as used in durometer study

Sample #	***Layers***	***RPM***	***Dimensions (in mm)***	***Road Width (in mm)***
1	24	1000	50 x 50 x 5.8	0.40 – 0.80
2	24	1000	50 x 50 x 5.8	0.40 – 0.80
3	24	1000	50 x 50 x 5.8	0.32 – 0.90

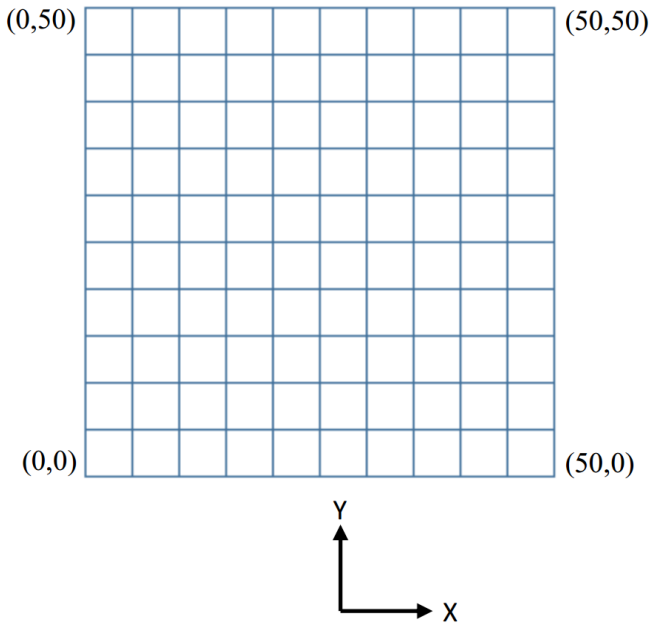


Figure 3.11: Illustration of durometer movement path for data collection

3.3.6 Radial Distribution Function (RDF)

Mixing efficiency/quality is characterized using image analysis of cross sections from MicroCT of samples printed with the active mixing chamber and inks A and B in differing ratios. Using ImageJ and figures 4.8(a) and (b), the normalized radial distribution functions for both samples are calculated and plotted. The RDF determines how density varies as a function of distance to a reference atom. Mathematically,

$$g(r) = \frac{N(r)}{4\pi r^2 \rho} \qquad [3.1]$$

Where g(r) is the probability of finding another particle at a distance *r* from a reference particle, N(r) is the number of particles within the a disc of width Δr (see figure 3.12 for a graphical representation), and ρ represents the number of particles per cm^2. At short distances, RDF is zero. However, as the radius expands beyond one particle diameter, the density of particles increases dramatically. As particle homogeneity is achieved, the RDF (g(r)) converges to unity. Figure 3.12 graphically illustrates the basis of an RDF with a reference atom at the center and an infinitesimal

ring, *dr*, surrounding the atom. As *dr* expands, so too does the density of atoms, eventually (assuming particle dispersion uniformity) converging to one, indicating uniform probability. Should homogeneity not exist, the RDF will not converge.

An ImageJ macro (Appendix B) is applied to aid in calculating the RDF. A noise threshold is first determined to find particle edges. Through trial and error, empty spaces are eliminated from the threshold and subsequent derivations. Once the majority of particles are captured, the macro is executed. Figures 4.8(a) and (b) are used for RDF derivations. For control, figure 4.8(b) is modeled for poor mixing (i.e., particles laden within half of a filament). Filament rod halves are removed by decreasing the brightness so as to appear black. The RDF macro is executed on the modified image and is presented as figure 4.17.

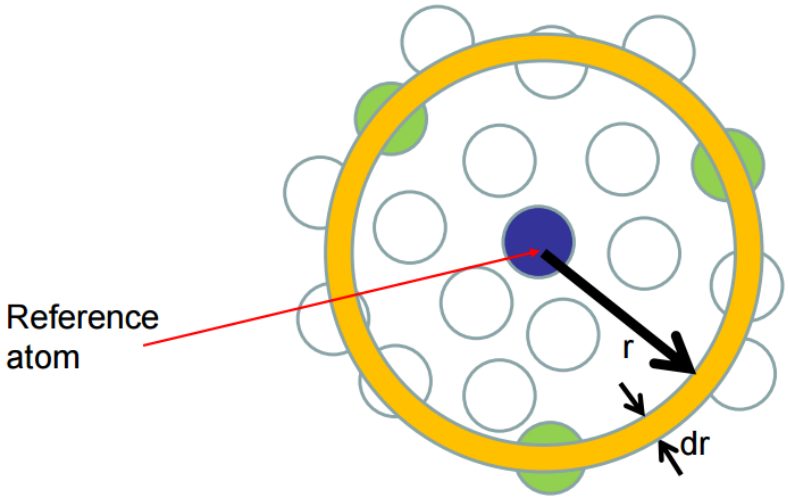


Figure 3.12: Graphical illustration of the RDF [43]

3.3.7 X-ray Micro-Computed Tomography (MicroCT)

Effectively measuring particle dispersion and mixing efficiency is done by MicroCT (Skyscan 1172). MicroCT is a non-destructive imaging technique based on X-ray absorbance of materials. It is capable of generating micron resolution 3D images of complex geometries [44, 45, 46]. Tungsten is a good candidate for X-ray analysis given its dense structure and thus good X-ray absorbance.

A tungsten X-ray source energized at 10kW (80kV and 124μA) is used to probe the sample. A total of 1021 radiographs are taken at a camera resolution of 11.63μm/pixel in 0.4 degree steps and 885ms exposure time. Samples from figure 3.8 are cut into smaller rectangles (~3mm x 3mm x 25mm) and mounted to the CT stage for examination. Figure 3.13 illustrates the sample mounting technique. Overall, MicroCT analysis provided significant data which is detailed in the following chapters.

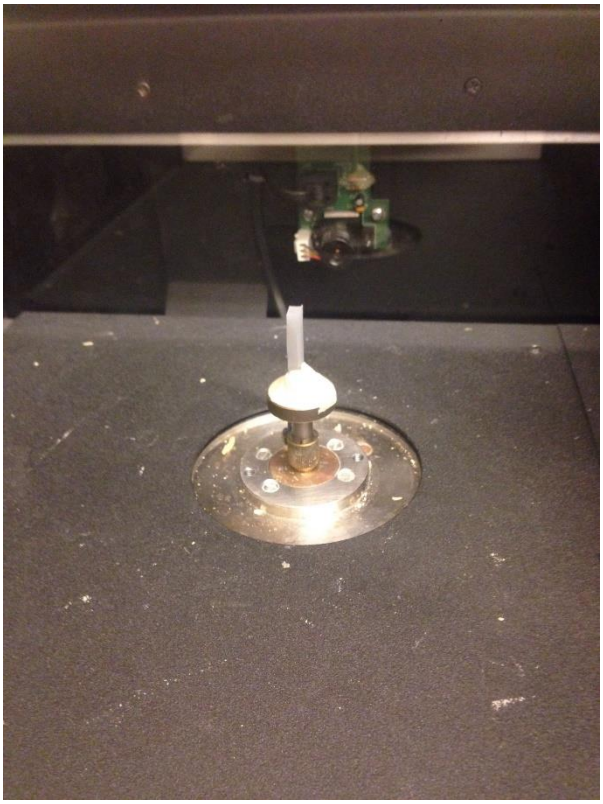


Figure 3.13: MicroCT stage with sample mounted vertically and attached using putty

3.4 Summary

This chapter expanded upon experimental setup and methodology. Results are included from preliminary experiments such as rheological measurements and volumetric flowrate calibrations. Additionally, the active mixing device is discussed at length in terms of its mechanical and electrical properties. The subsequent chapters provide results and a discussion of the findings.

CHAPTER IV: RESULTS

4.1 Introduction

This chapter presents the results from the experimental and characterization techniques described in Chapter 3 in the following order: 4.2 Ink Flowrate Validation (ref., 3.3.1), 4.3 Ink Rheology (ref., 3.3.2), 4.4 Scanning Electron Miscroscopy (ref., 3.3.3), 4.5 Micro Computed Tomography (ref., 3.3.6), 4.6 Radial Distribution Function (ref., 3.3.5) and 4.7 Durometer Hardness (ref., 3.3.4). In this chapter, the results are presented in an organized manner with minor attention to interpretation of the data. Detailed discussion of the significance of the results is reserved until chapter 5.

4.2 Ink Flowrate Validation

Figures 4.1(a), (b), and (c) illustrate the data obtained from the flowrate validation experiment. To represent the data, a theoretical value of mass is determined based upon volumetric flowrate ([3.1] and [3.2]), time duration of extrusion (t=180s), volume extruded ([3.3] and [3.4]), and theoretical mass ([3.5] and [3.6]). Both experimental values of material 'A' and material 'B' are then compared to the theoretical values [3.5] and [3.6] based on respective flowrates. Equations 3.1 and 3.2 yield the volumetric flowrate:

$$\dot{V}_1 = A_p v_1 [3.1]$$
$$\dot{V}_2 = A_p v_2 [3.2]$$

where:

$$A_p = Area\ of\ Plunger \approx 195.5 mm^2$$

$$v_1 = Plunger\ Velocity = 0.01\frac{mm}{s}$$
$$v_2 = Plunger\ Velocity = 0.02\frac{mm}{s}$$

Time duration of extrusion is:

$$t = Time\ Duration\ of\ Extrusion = 180s$$

Calculating the volume is thus:

$$V_1 = \dot{V_1}t \approx 352mm^3 \quad [3.3]$$
$$V_2 = \dot{V_2}t \approx 704mm^3 \quad [3.4]$$

Theoretical mass is then determined by:

$$m_1 = V_1\rho \approx 0.388g \quad [3.5]$$
$$m_2 = V_2\rho \approx 0.777g \quad [3.6]$$

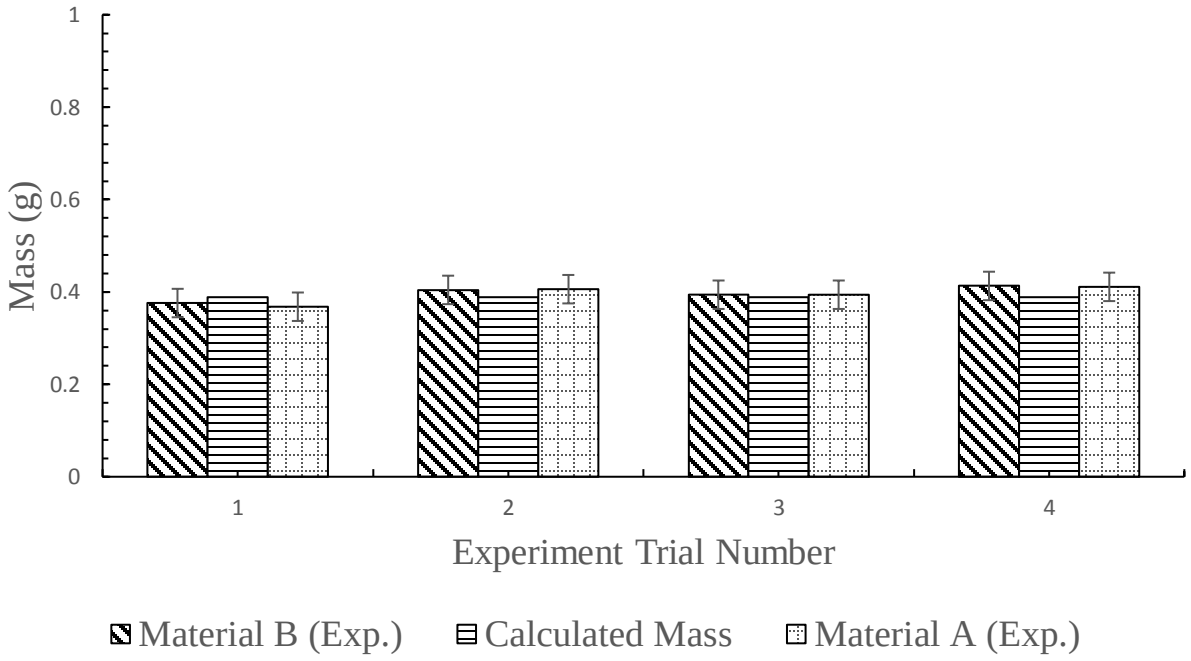


Figure 4.1(a): Plot of material 'A' and material 'B' (1:1 flowrate ratio) against derived theoretical mass [3.5]

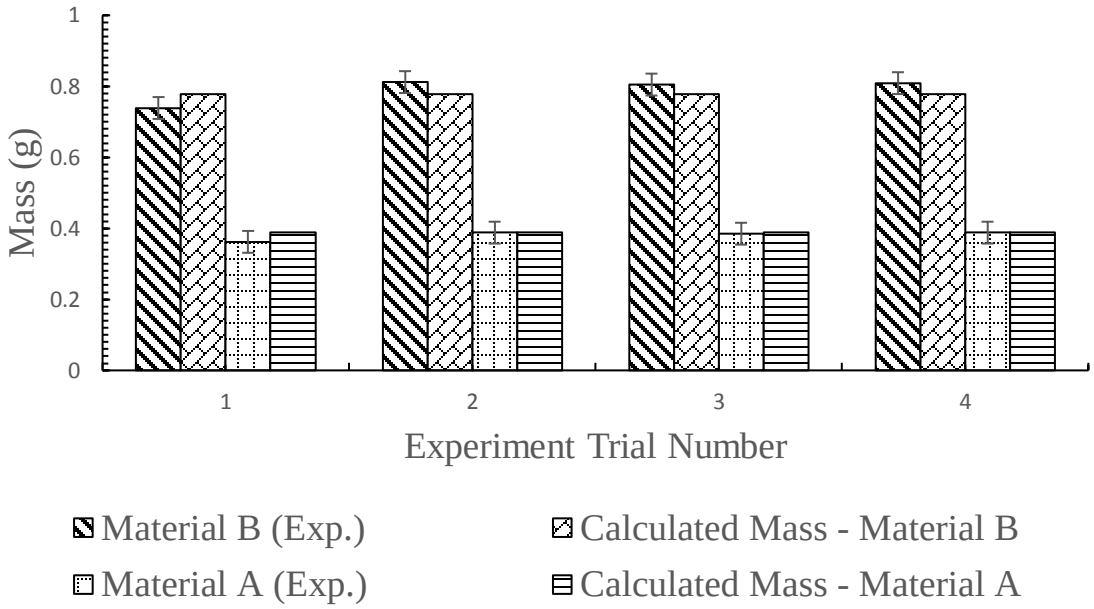


Figure 4.1(b): Plot of material 'A' and material 'B' (1:2 flowrate ratio) against derived theoretical mass [3.5] and [3.6] respectively

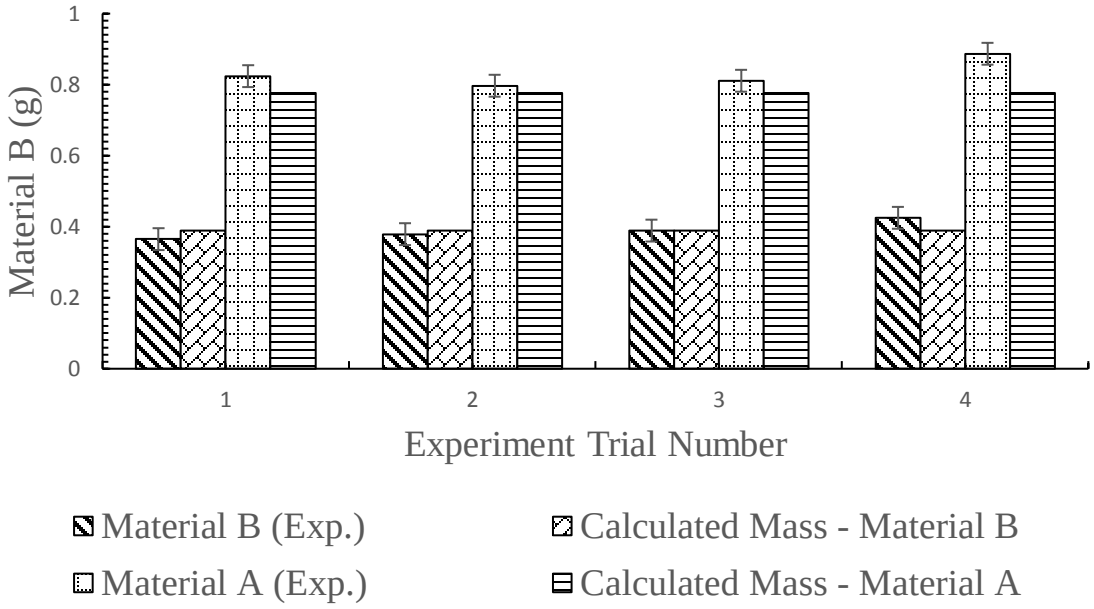


Figure 4.1(c): Plot of material 'A' and material 'B' (2:1 flowrate ratio) against derived theoretical mass [3.6] and [3.5] respectively

For all three experiments, results indicate correct plunger speeds for given flow ratios. An estimated propagation of uncertainty for mass (δm_T) is calculated. Factors such as plunger velocity (u_1=0.01mm/s & δu_1=0.1µm/s : u_2=0.02mm/s & δu_2=0.1µm/s), weigh boat insertion and extraction time (t=180s & δt=0.5s), and syringe diameter (d=15.78mm & δd=0.2mm) are included in the derivation. The δ values indicate the individual uncertainty for velocity, time, and syringe diameter. V is defined as the volume of ink, δV as the total volume uncertainty propagation, δV_1 as the volume uncertainty given u_1=0.01mm/s, δV_2 as the volume uncertainty given u_2=0.02mm/s, δm_1 and δm_2 as the mass uncertainty propagation values, and δm_T is the total mass uncertainty. Equation 4.1 is given below detailing the uncertainty calculation and derivation.

$$\delta V = \sqrt{\left(\frac{\partial V}{\partial d}\delta d\right)^2 + \left(\frac{\partial V}{\partial u}\delta u\right)^2 + \left(\frac{\partial V}{\partial t}\delta t\right)^2} \quad ; \quad V = \frac{\pi}{4}d^2 ut \qquad [4.1]$$

where:

$$\delta V_1 \approx 9.64mm^3 \text{ @ } u_1 = 0.01mm/s$$
$$\delta V_2 \approx 18.3mm^3 \text{ @ } u_2 = 0.02mm/s$$
$$\delta m_1 = \rho \delta V_1 \approx 0.0106g$$
$$\delta m_2 = \rho \delta V_2 \approx 0.0202g$$
$$\delta m_T = \delta m_1 + \delta m_2 = 0.0308g$$

4.3 Ink Rheology

Rheology for both SE1700 laden with tungsten as well as neat SE1700 is conducted. Figure 4.2(a) and (b) shows both viscous and elastic moduli as a function of shear stress. For viscoelastic materials, the elastic modulus is dominant until a certain shear stress is achieved. Once achieved, the viscous modulus becomes dominant, exhibiting liquid-like behavior. The data is plotted in figure 4.3(a) and (b). As shear rate is increased, viscosity is decreased, thus exhibiting a shear-thinning behavior. The shear-thinning behavior is an optimal trait of "printable" inks. Additionally, tan(δ) is also plotted as a function of shear stress in figure 4.4(a) and (b). At a certain shear stress level, the ink moves from elastic-dominant to viscous-dominant, further detailing the shear-thinning nature of the material.

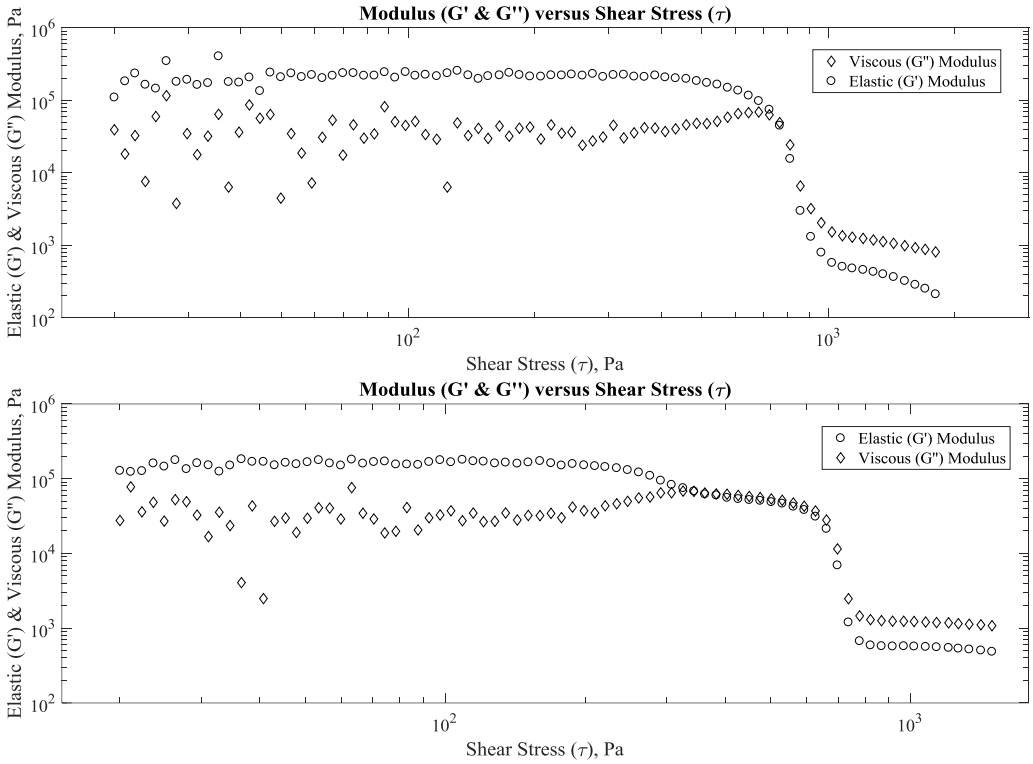


Figure 4.2: (a) Plot of viscous and elastic modulus over shear stress at room temperature for SE1700 laden with tungsten and (b) neat SE1700

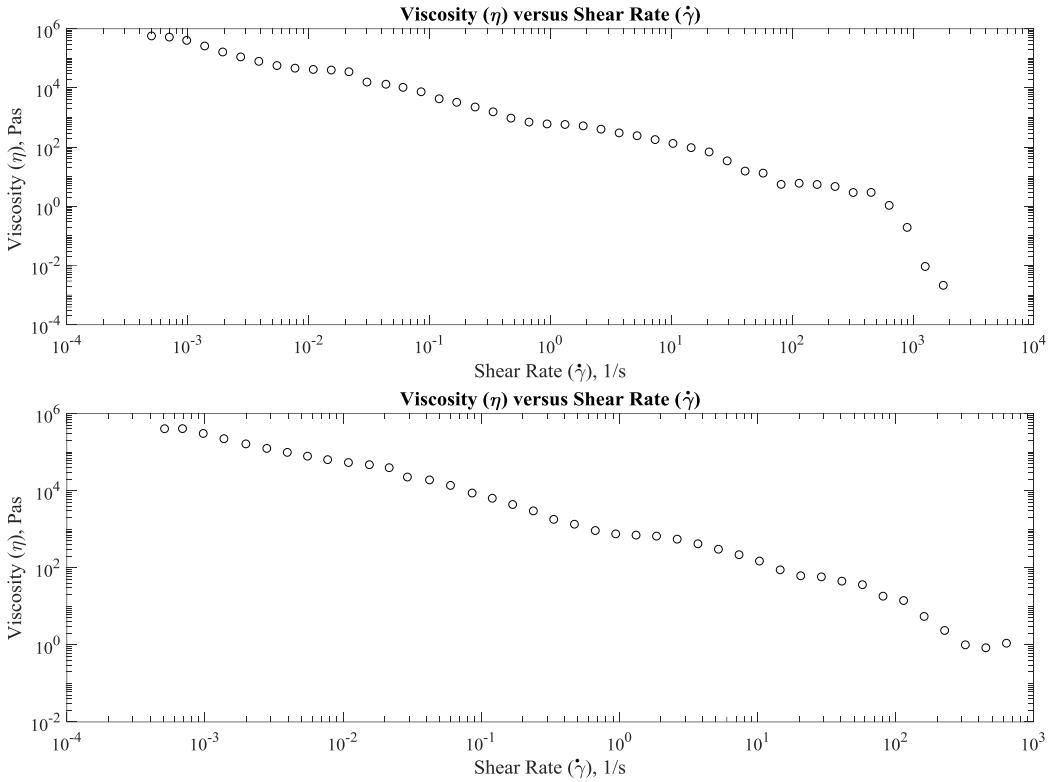


Figure 4.3: (a) Plot of viscosity over shear rate at room temperature for SE1700 laden with tungsten and (b) neat SE1700

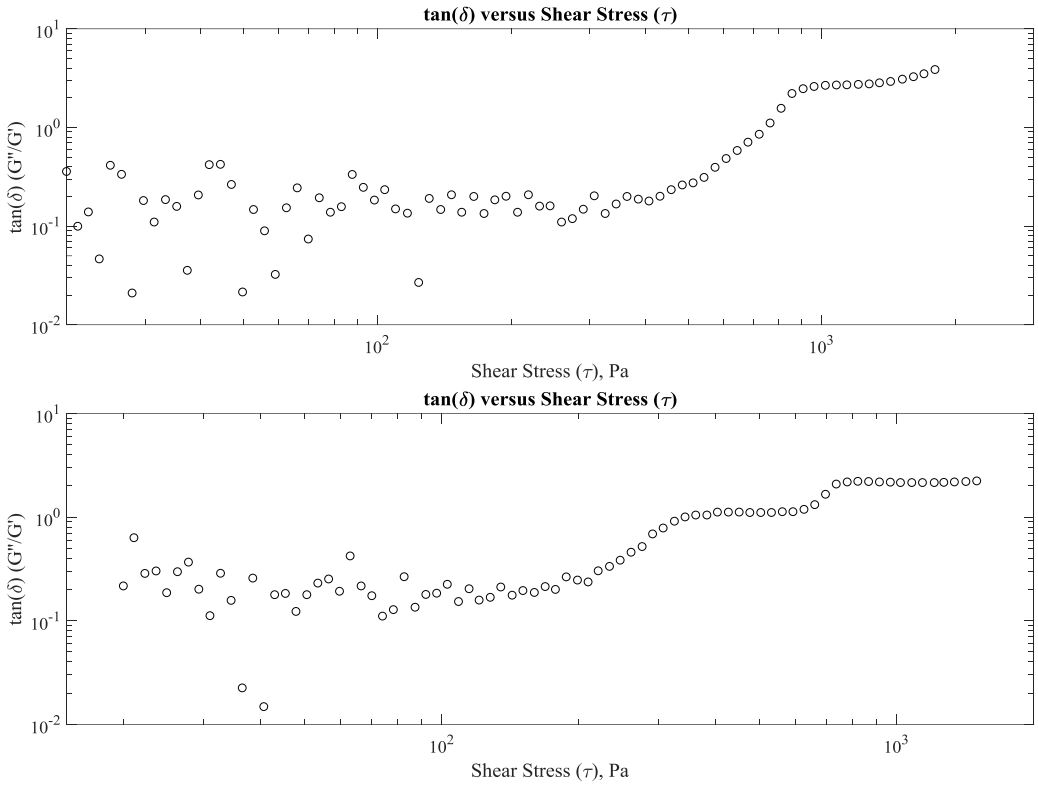


Figure 4.4: (a) Plot of tan(δ) over shear stress at room temperature for SE1700 laden with tungsten and (b) neat SE1700

4.4 Scanning Electron Microscopy (SEM)

Several tungsten SEM micrographs are acquired and are presented in figure 4.5(a)-(e). The particles have a uniformly spherical morphology and a distribution of sizes. Figure 4.6 illustrates particle size distributions for 48 particles with roughly 58% of particles falling between the 5-10μm diameter ranges. According to the powder data sheet (see Tekna W-25 datasheet), there are fewer than 5% of particles greater than 25μm in diameter.

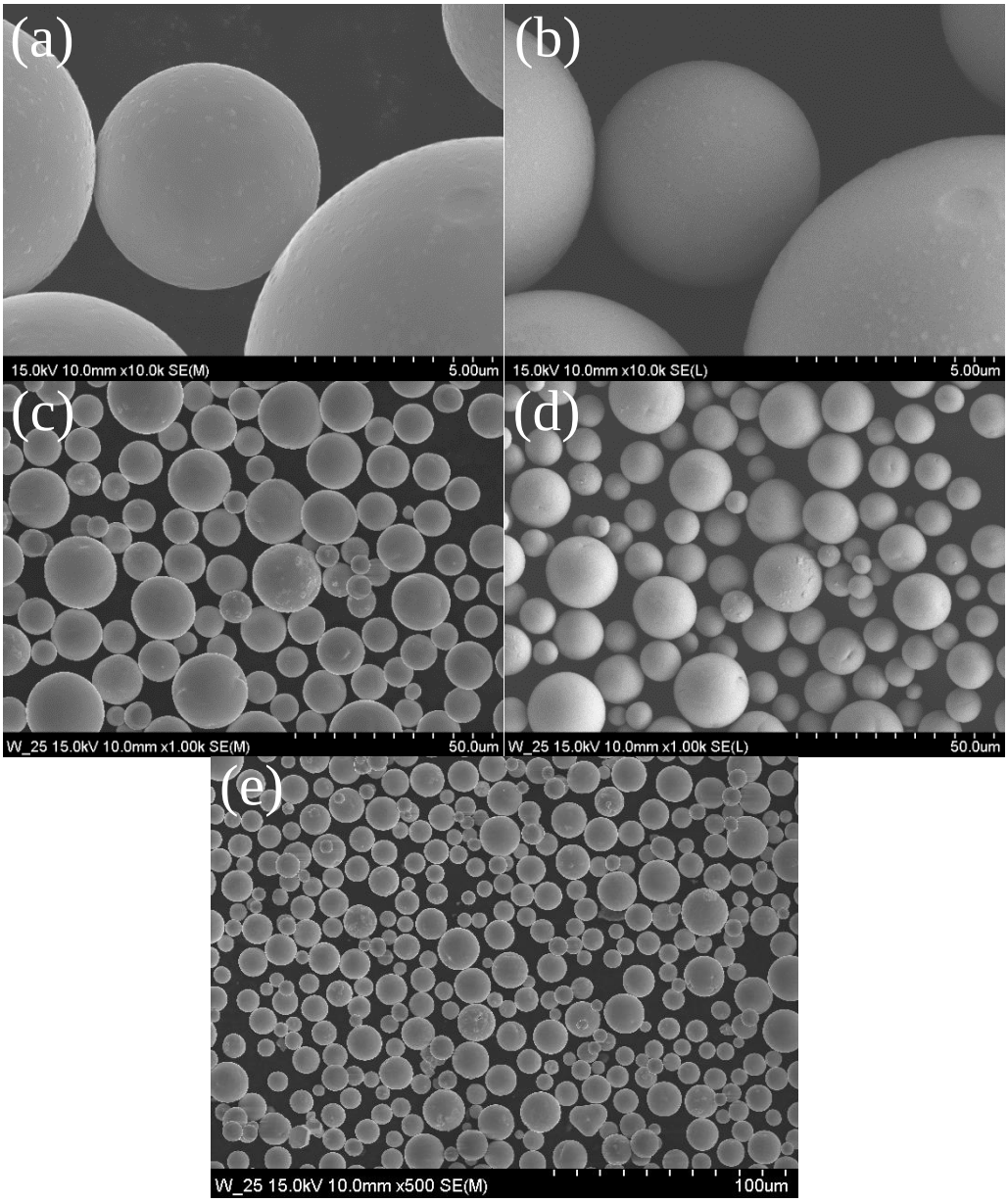


Figure 4.5: SEM images of tungsten particles at (a) 15.0kV at 10,000x magnification (mixed), (b) 15.0kV at 10,000x magnification (lower), (c) 15.0kV at 1,000x magnification (mixed), (d) 15.0kV at 1,000x magnification (lower), and (e) 15.0kV at 500x magnification (mixed)

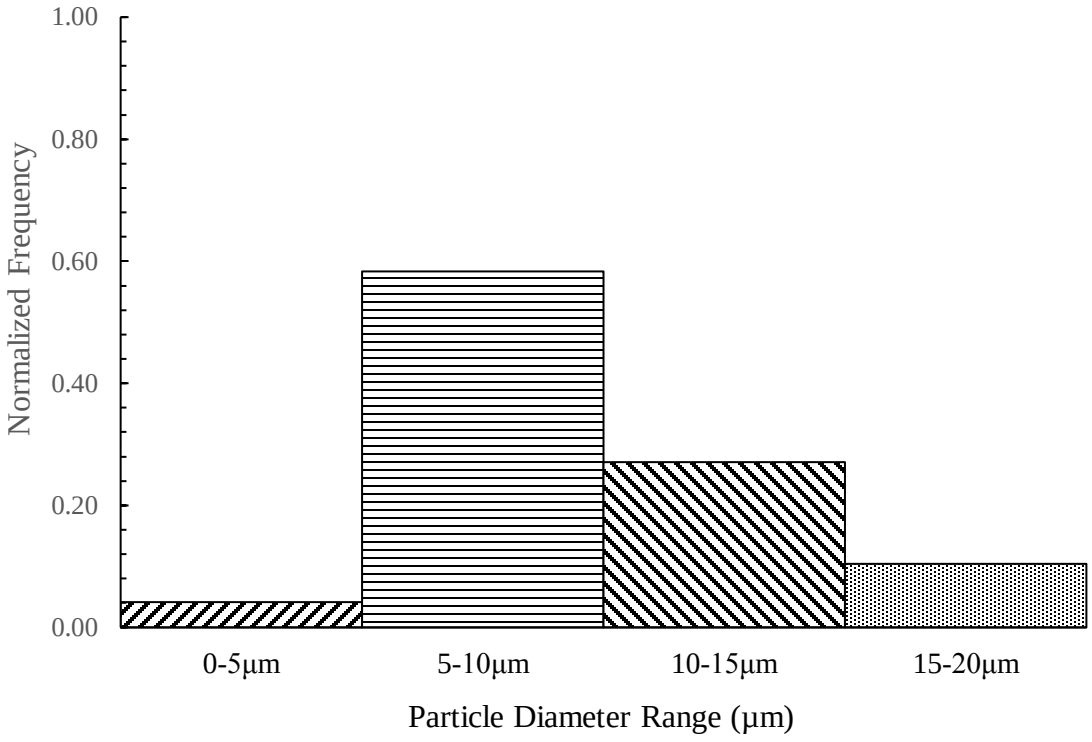


Figure 4.6: Plot of particle size distributions with the majority of data points falling in the 5-10µm category

4.5 X-ray Microcomputed Tomography (MicroCT)

Before subjection to MicroCT, samples from figure 3.6 (i.e., 50/50 composition at 2000 RPM and 75/25 composition at 1000 RPM) are cut into smaller rectangles (~3mm x 25mm x 3mm). This ensures samples properly fit the CT stage for imaging. The images are presented in figures 4.8(a)-(d), 4.9, and 4.10. Figures 4.8(a) and (b) depict radiographs for 75/25 composition at 1000 RPM and 50/50 composition at 2000 RPM respectively. The images represent a vertically oriented cross-section (reference figure 4.7) where the width of the image corresponds to the sample thickness. The white space in figures 4.8(a) and (b) indicate empty space between the filaments. The grey and black indicate the filament rods and tungsten particles respectively. Tungsten particles (black dots) are dispersed across the filaments with an increase seen in the 50/50 composition

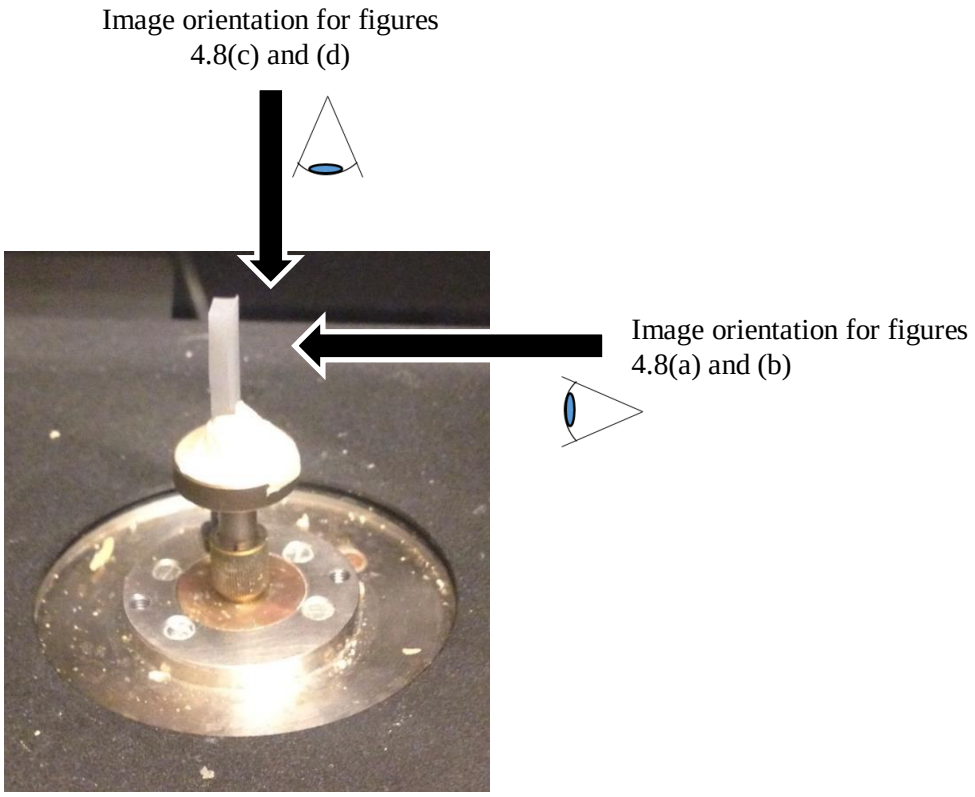
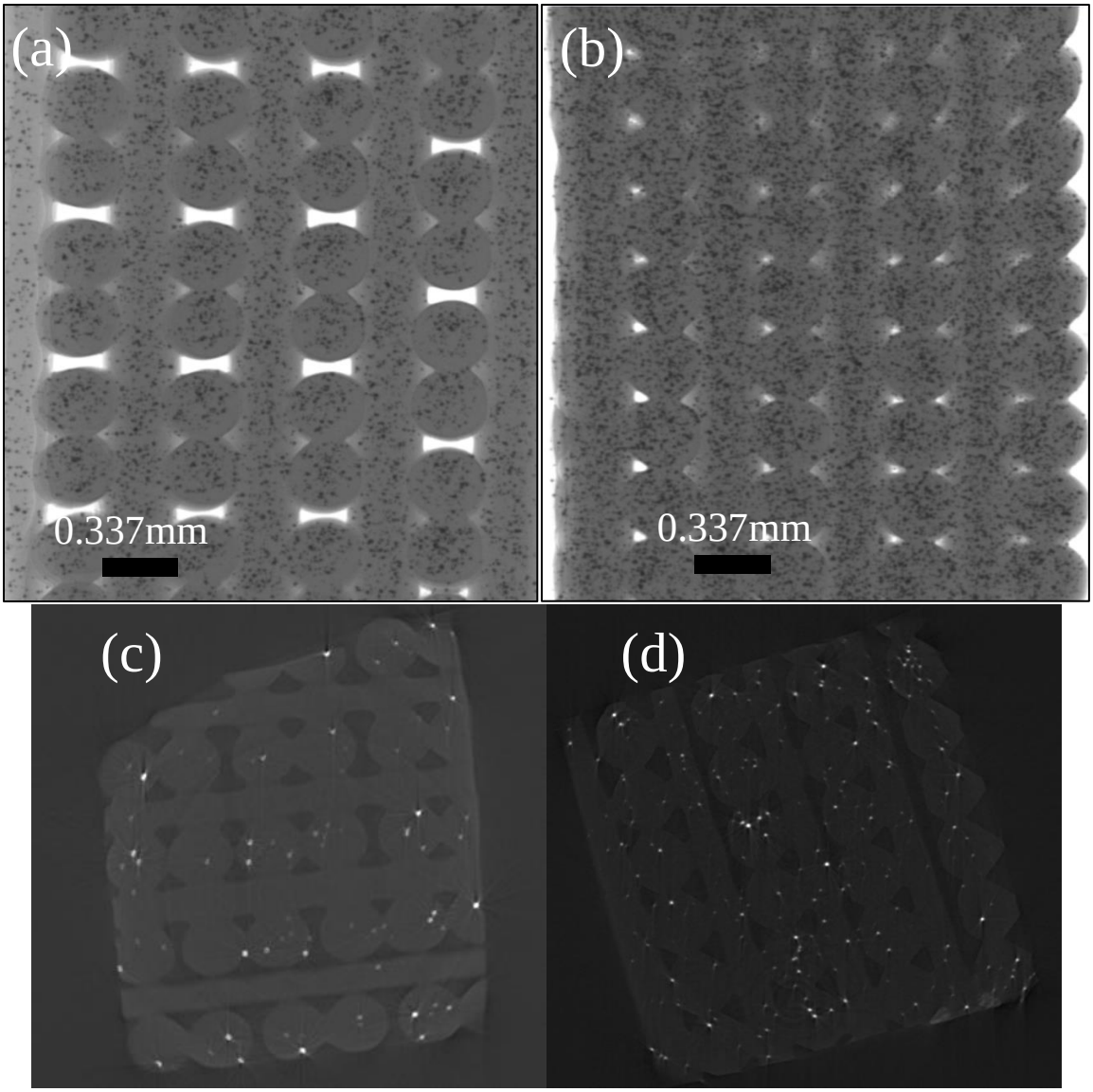


Figure 4.7: Sample orientation for figures 4.8(a)-(d)

Figures 4.8(c) and (d) depict a top-down view of each sample at different layers. The black sections in both figures indicates empty space while the dark grey depicts filament rods. The white dots scattered across the images represent the tungsten particles. Similar to figures 4.8(a) and (b), 4.8(d) demonstrates an increase in tungsten concentration of 4.8(c), the 75/25 sample. The CT images are combined to construct a three-dimensional image of the sample, depicted in figures 4.9 and 4.10. The images are enhanced to remove the silicone filaments thus leaving only tungsten particles.


(a)
0.337mm
(b)
0.337mm
(c)
(d)

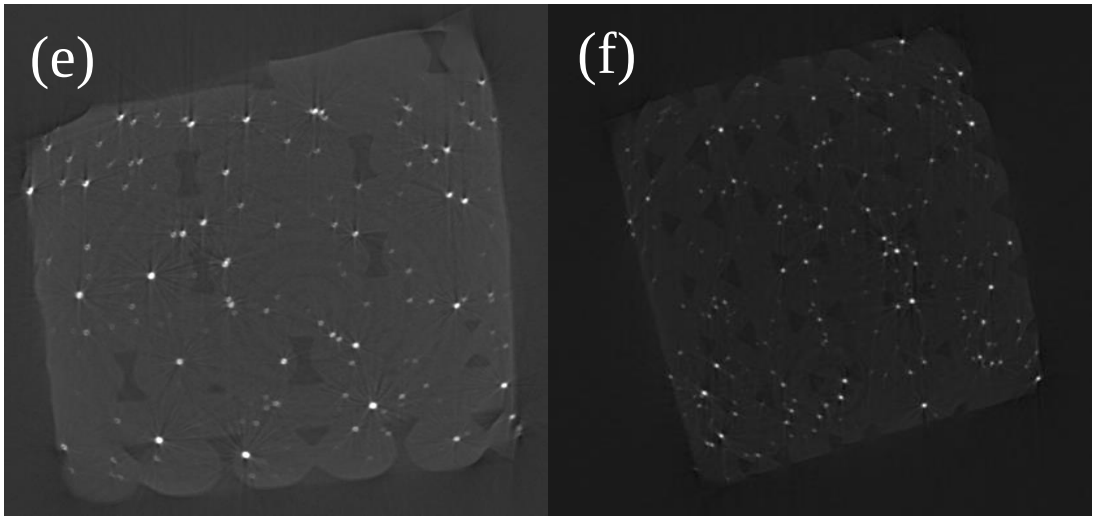


Figure 4.8: Cross-sectional radiographs and MicroCT reconstructed slices of pure SE1700 mixed with tungsten. (a), (c), and (e) is a 75/25 mix and (b), (d), and (f) is a 50/50 mix

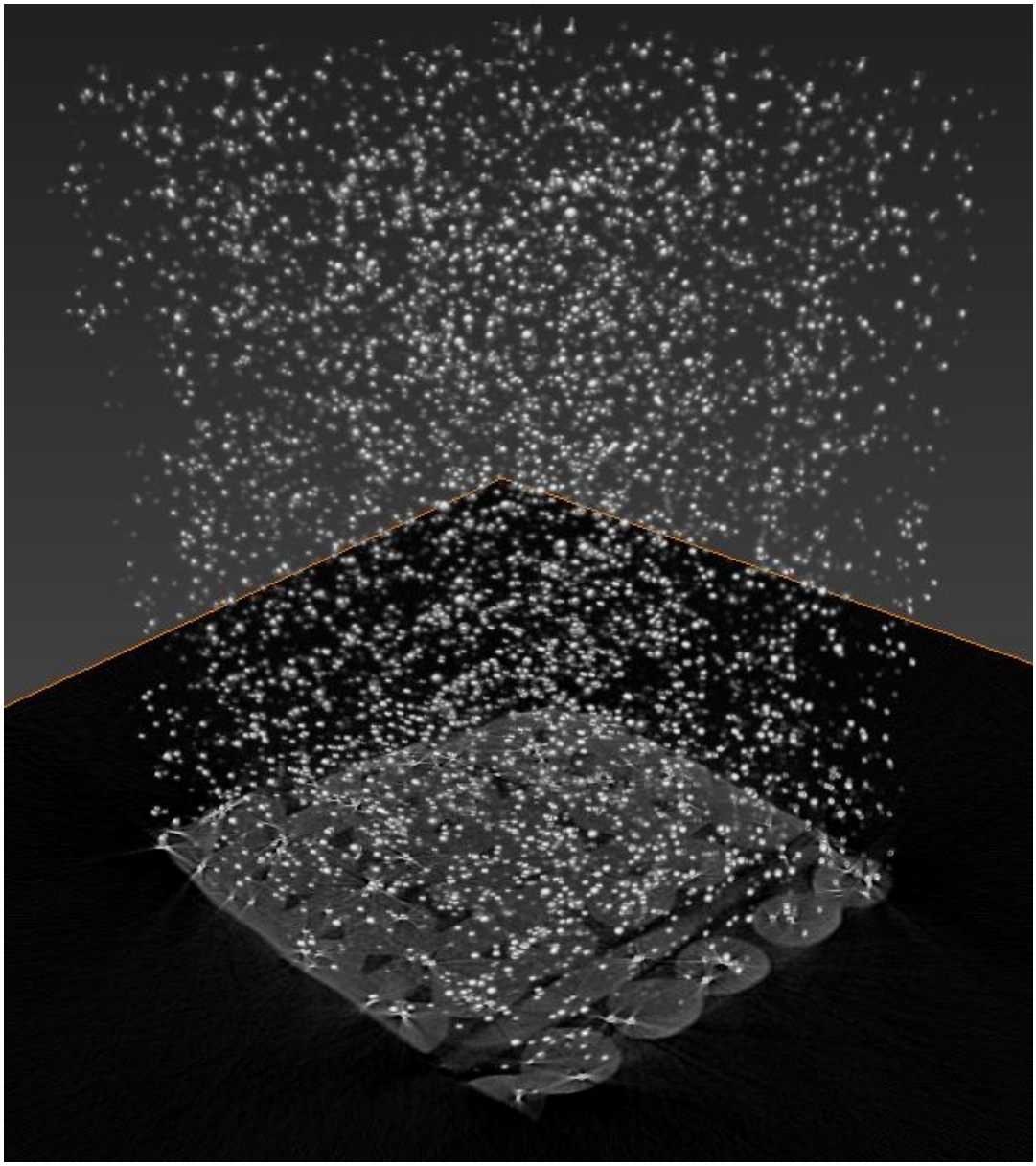


Figure 4.9: MicroCT reconstructed 3D image of tungsten dispersion in the 75/25 at 1000 RPM's sample

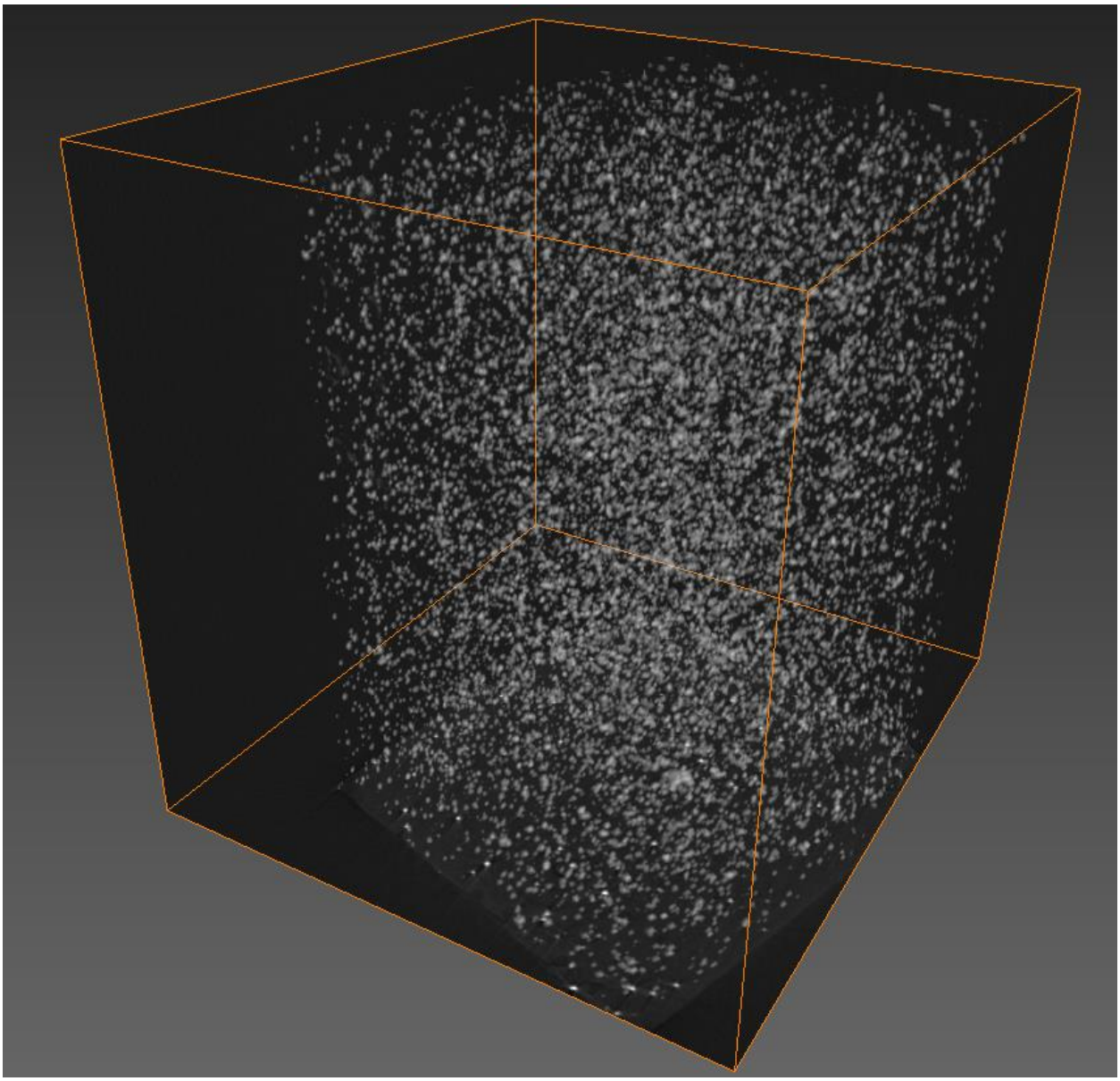


Figure 4.10: MicroCT reconstructed 3D image of tungsten dispersion in the 50/50 at 2000 RPM's sample

4.6 Radial Distribution Function (RDF)

Figures 4.5-4.10 represent data obtained from an RDF of both 50/50 and 75/25 samples based on three random cross-sectional images. Figures 4.11-4.13 show a convergence to unity with small fluctuations around one. Figures 4.14-4.16 show a more gradual approach to convergence,

occurring around 450μm (150 pixels). This is partly due to the empty space between the filaments. The 50/50 sample RDF converges more rapidly to unity over the 75/25 sample. Distances (x-axis) are given in pixels but may be converted to μm by multiplying pixel value by ~3. For control, figure 4.17 illustrates the "poor" mixing regime and its respective Radial Distribution Function.

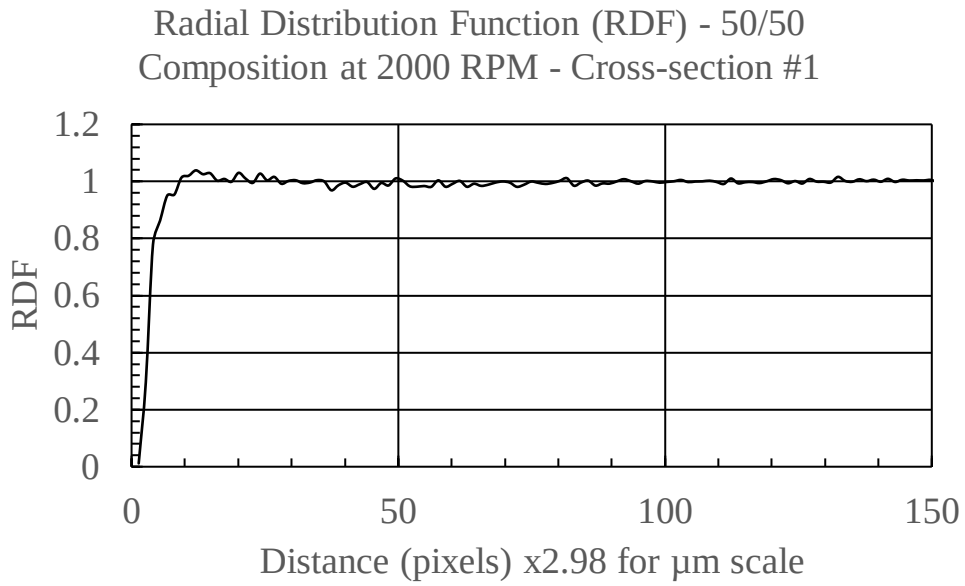


Figure 4.11: Radial Distribution Function for a 50/50 composition at 2000 RPM for cross-section #1

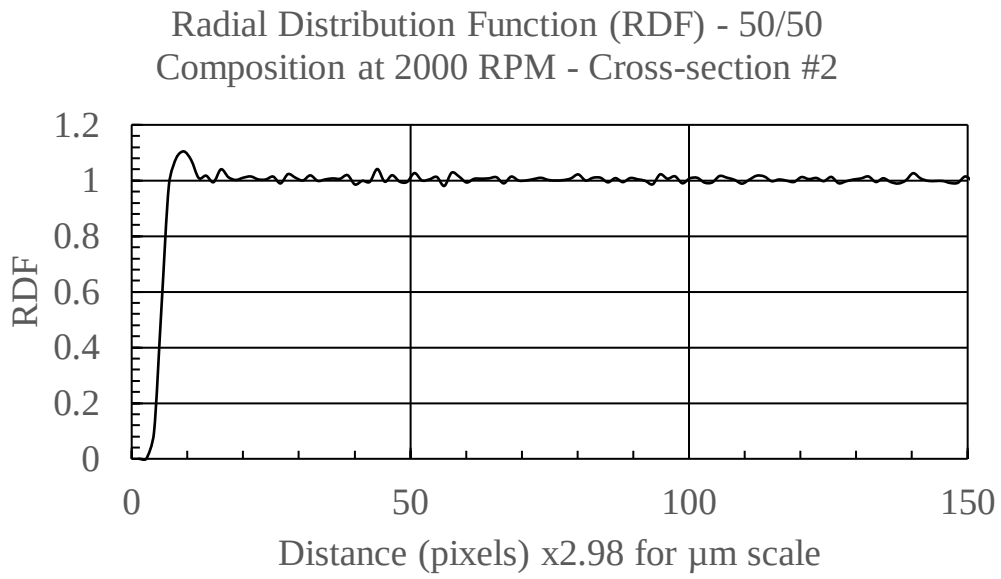


Figure 4.12: Radial Distribution Function for a 50/50 composition at 2000 RPM for cross-section #2

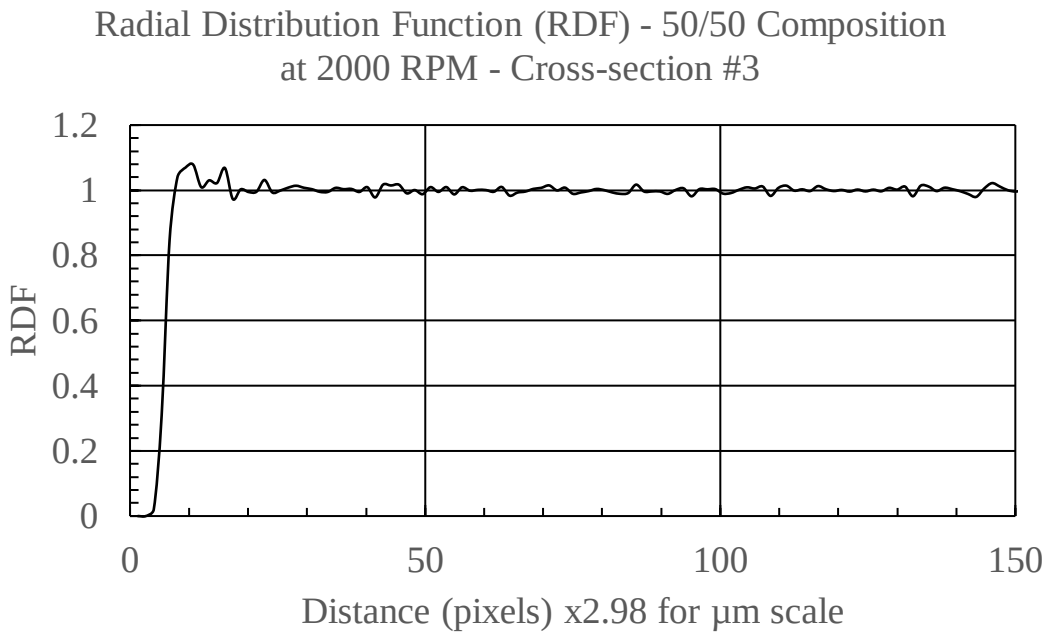


Figure 4.13: Radial Distribution Function for a 50/50 composition at 2000 RPM for cross-section #3

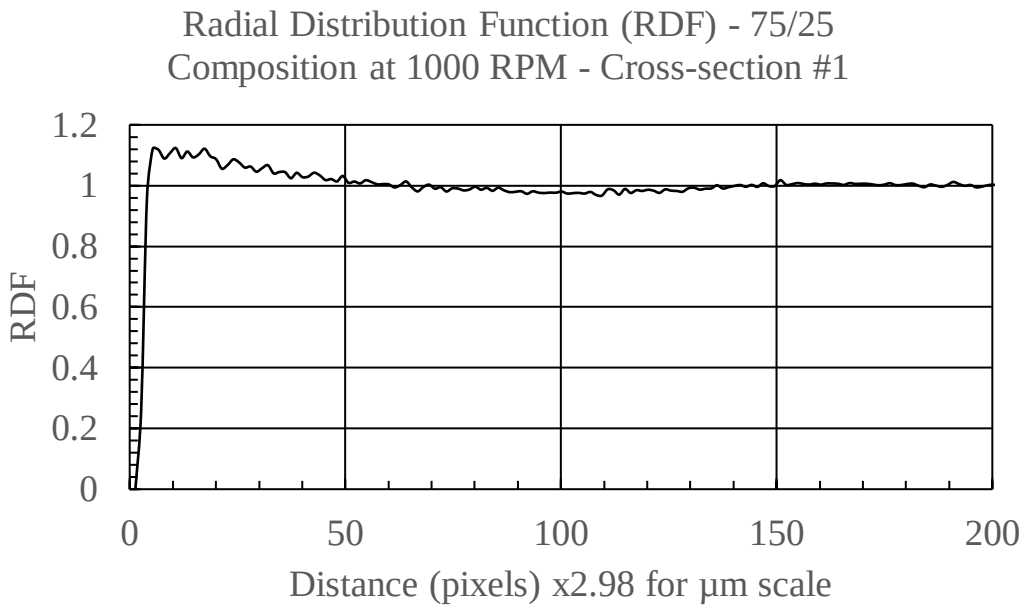


Figure 4.14: Radial Distribution Function for a 75/25 composition at 1000 RPM for cross-section #1

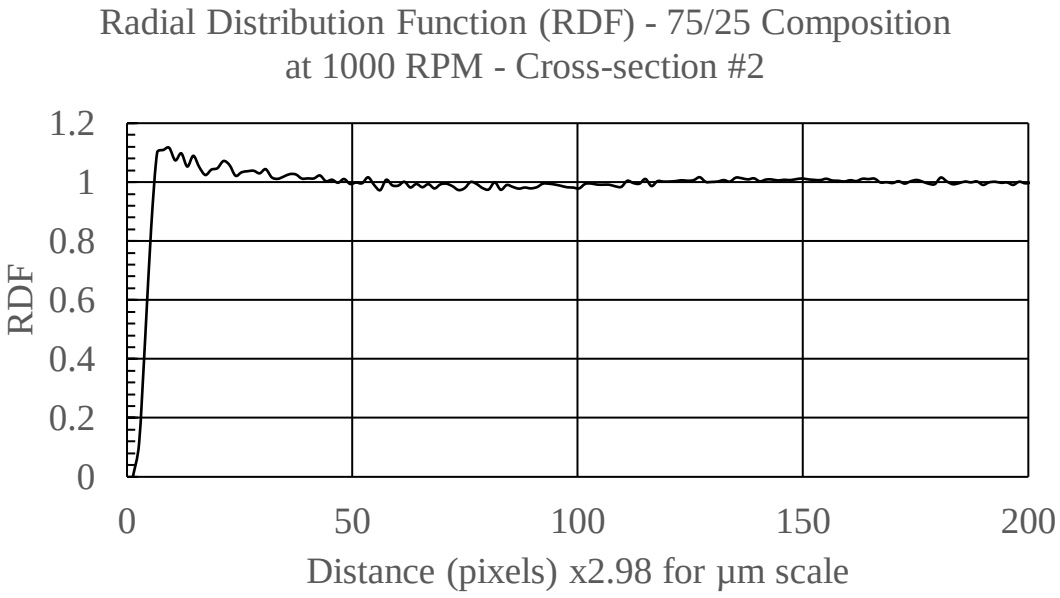


Figure 4.15: Radial Distribution Function for a 75/25 composition at 1000 RPM for cross-section #2

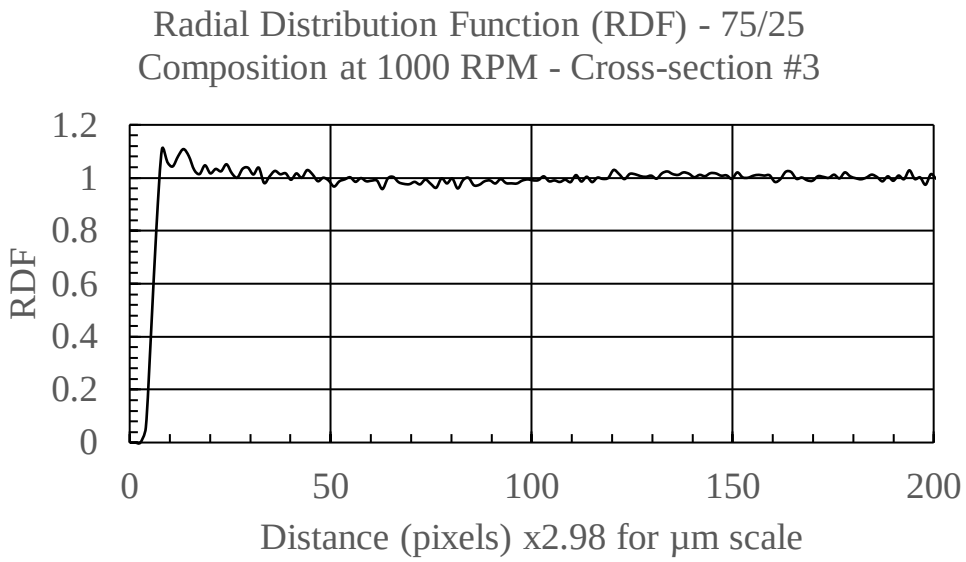


Figure 4.16: Radial Distribution Function for a 75/25 composition at 1000 RPM for cross-section #3

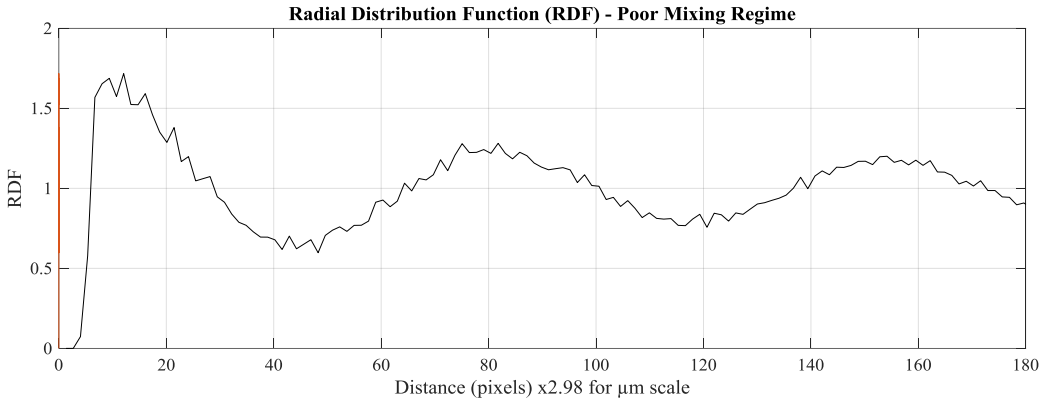


Figure 4.17: Radial Distribution Function control plotted for the poor mixing regime

4.7 Durometer Hardness

Three samples of varying geometry are printed and subjected to durometer testing. The samples exhibited a face-centered tetragonal (FCT) cell structure with rod spacing increasing from one end to another. This geometry is detailed in section 2.3.1. Each layer is rotated by 90° and every third layer offset by 0.5mm. Samples are printed with two outside rims. A total of 2500 data points are collected across the top of each sample. Figures 4.18-20 illustrate sample hardness as a function

of (X,Y) position. The data is analyzed and plotted using Matlab. Two of the three samples constructed are structurally identical (figures 4.18 and 4.19). The third sample (figure 4.20) incorporated a broader range of lattice spacing over the first two samples, beginning with 0.32mm and ending at 0.9mm. Figures 21, 22, and 23 depict the physical samples, showing overall size as well as graded lattice spacing. The graphs depict what is expected: Hardness is at its highest value where lattice spacing is smallest and vice versa.

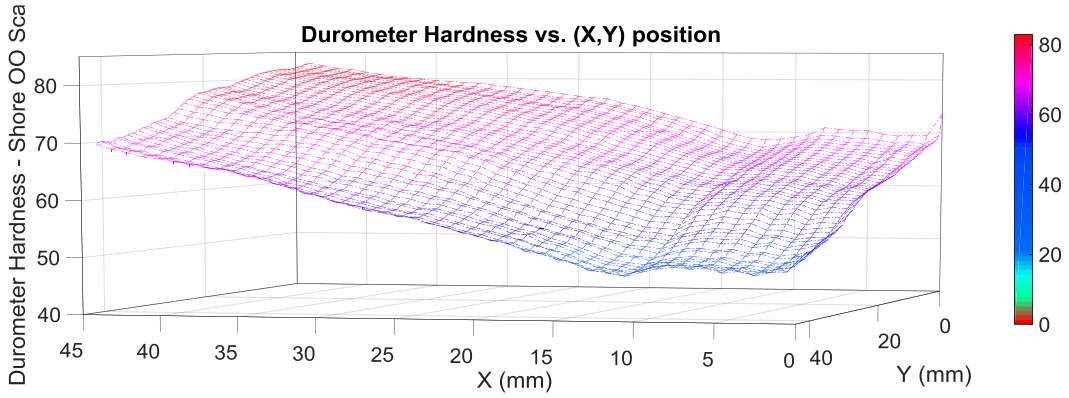


Figure 4.18: Material hardness as a function of (X,Y) position (in mm) based on the Shore OO scale. Rod spacing increased from 0.4mm to 0.8mm with an FCT unit cell structure and two outside rims. A color scale is provided to indicate the hardest and softest points.

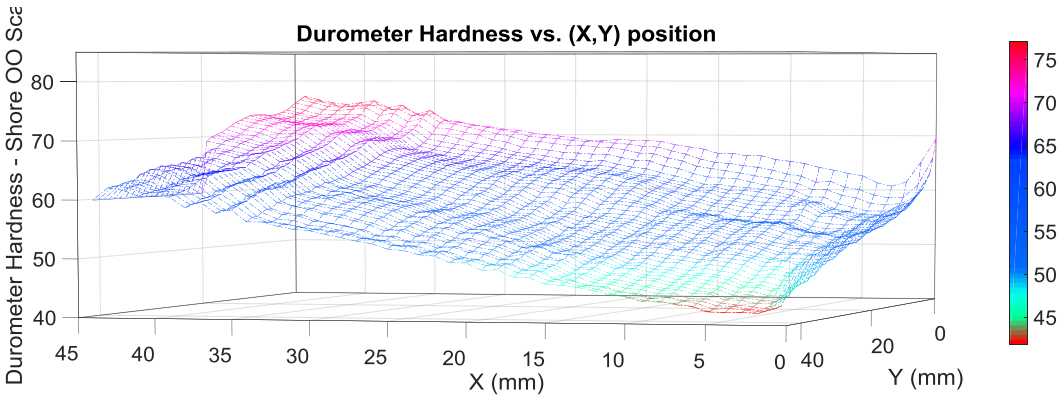


Figure 4.19: Material hardness as a function of (X,Y) position (in mm) based on the Shore OO scale. Rod spacing increased from 0.4mm to 0.8mm with an FCT unit cell structure and two outside rims. A color scale is provided to indicate the hardest and softest points

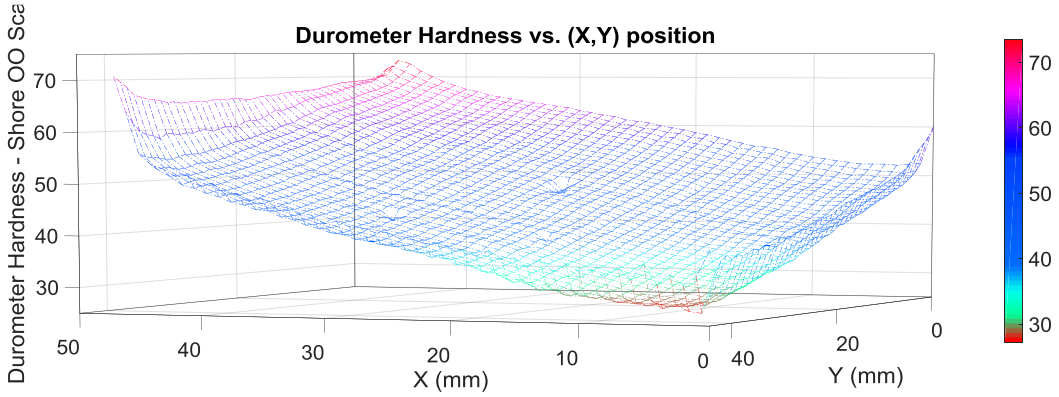


Figure 4.20: Material hardness as a function of (X,Y) position (in mm) based on the Shore OO scale. Rod spacing increased from 0.32mm to 0.9mm with an FCT unit cell structure and two outside rims. A color scale is provided to indicate the hardest and softest points.

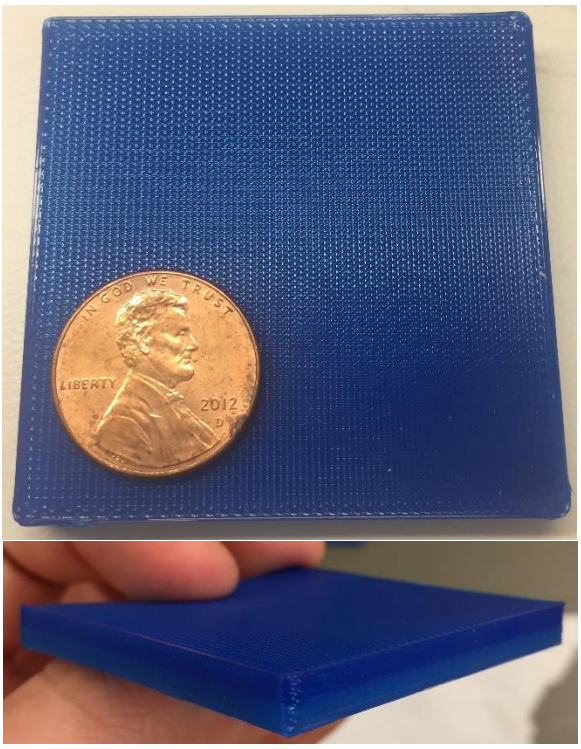


Figure 4.21: Digital image of the first sample with a penny for scale

Figure 4.22: Digital image of the second sample with a penny for scale

Figure 4.23: Digital image of the third sample with a penny for scale

4.8 Summary

Results from all major experiments are presented. Tungsten SEM images showed good particle size and size distribution based on the Tekna datasheet. A Radial Distribution Function is used to determine homogeneity amongst atoms as a function of distance. Samples are also constructed for subjection to mechanical testing by durometer. Initially, the end goal of the durometer study is to determine mechanical properties of a sample with varying geometry and composition. However, because of time constraints, only samples with graded geometry are tested. For future work, durometer tests will be conducted on samples of graded geometry and composition utilizing two silicones of disparate elastic moduli.

CHAPTER V: DISCUSSION AND CONCLUSION

5.1 Introduction

The focus of this chapter is to analyze and discuss the results from the previous chapter. As such, this chapter is separated into two major sections: Characterizing mixing efficiency (Objective #1) and mechanical properties of graded lattice structures (Objective #2). First, MicroCT image results (referring figures 4.3(a)-(f) and 4.11-12) are evaluated and discussed. Second, an elaboration on the qualitative analysis, provided by the Radial Distribution Function (RDF), is provided. Finally, a discussion of the durometer data is given, expanding upon the results acquired.

5.2 Discussion – Mixing Efficiency

Two samples of varying compositional ratios (SE1700:Tungsten) and impeller velocities were constructed through an active mixing device via direct-ink writing. The goal of this study was to characterize "mixedness". Do particles suspended in a viscoelastic material become well mixed when subjected to high shear rate? Subsections of the samples were subjected to microcomputed tomography (MicroCT) evaluations to determine compositional ratios as well as mixing efficiency (i.e., particle dispersion throughout the sample). If the samples were printed within the "poor" mixing regime, as seen in figure 5.1(a), filament rods would consist of a half-diameter of tungsten particles and pure SE1700 as the other half [24].

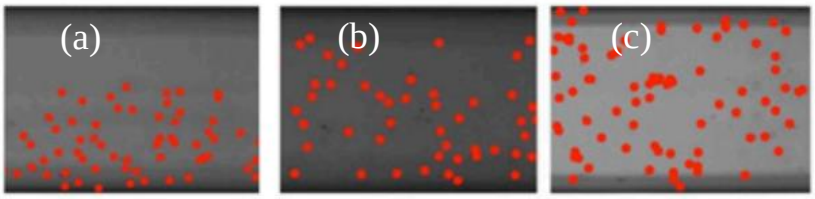


Figure 5.1: Graphical representation of three mixing regimes (a) poor, (b) middle, and (c) good mixing [24]

Qualitative characterization of mixing effectiveness in two dimensions was done using a Radial Distribution Function (RDF) (discussed in section 4.2.1) based on cross-sectional images obtained from MicroCT analysis. The first sample, 50/50 composition at 2000 RPM, demonstrated good homogeneity with few agglomeration spots (see figures 4.3(a), (c), (f), and 4.11). Additionally, the RDF converged to unity with little variation (see figures 4.5-4.7). From this data, good mixing characteristics are seen in the 50/50 sample. The higher impeller velocity may play a role in its more uniform dispersion. The second sample, 75/25 composition at 1000 RPM, exhibited somewhat less uniformity in terms of its RDF, radiographs, and MicroCT snapshots. Tungsten particles were mostly concentrated to the center of the filaments. This may be due to the lower paddle rotational velocity or paddle geometry. More MicroCT and statistical analysis on similar specimens is required to prove this hypothesis.

In terms of particle density, qualitatively, a 50/50 composition of SE1700 to tungsten is more pronounced over the 75/25 ratio. This result demonstrates a good syringe pump calibration and follows the results obtained through the volumetric flowrate experiment. When analyzed in three dimensions (figures 4.11-4.12), to the eye, both samples show particle dispersion uniformity. However, quantitatively, particle dispersion in two and three dimensions was not verified.

Further characterizations include mixing efficiency as a function of impeller geometry and spatial orientation within the mixing chamber. That is, how does the shape of the impeller

improve mixing; and what is the optimal impeller positioning inside the mixing chamber? These questions are saved for future work.

5.3 Discussion – Mechanical Properties of Graded Lattice Structures

The second major study utilized a durometer to measure hardness of a lattice structure with graded geometry. Initially, the overall goal was to construct a specimen with both graded geometry as well as composition of two silicones with disparate elastic moduli. However, because of time constraints and software issues, only structures with graded geometry were analyzed. Three samples were printed using SE1700 and measured via durometer. The first two samples were structurally identical while the third sample exhibited a broader range of lattice spacing. Durometer setup and measurement methodology is explained in section 3.3.4. Data acquired from the study is depicted in figures 4.14-4.16. Hardness was measured as a function (X,Y) position. The graphs follow a predictable trend, that is, shorter filament spacing leads to a high durometer value (harder) and larger spacing corresponds with a lower hardness value (softer). Occasional bumps or waves in the graphs arise from the durometer indenter submerging between filaments (i.e., pressing into a hole). The first two samples overall hardness is greater than that of the third sample (with broader lattice spacing). Hardness levels increase around the edges of the samples because of the outside rims surrounding the lattice.

The results are characterized in terms of a simple supported beam model. A transverse load (durometer foot) is being applied to the center of two-support beam (see figure 5.1). As the supports move away from each other, deflection under load increases. After a certain distance, the beam will succumb to deflection under its own weight.

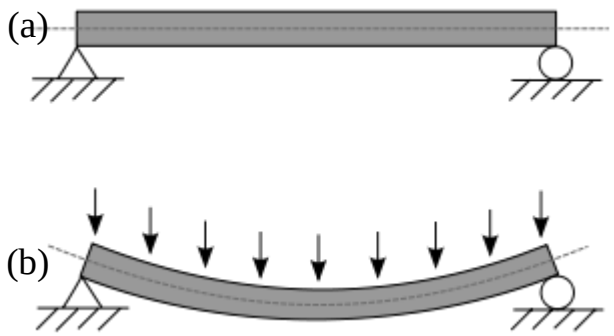


Figure 5.2: Graphical representation of a simple supported beam model with (a) no load and (b) transverse load

While results indicate a straightforward conclusion, a more complex solution (i.e., relating material hardness as a function of position) is the next step. Additionally, relating durometer not only to position but also elastic modulus. Future work will include mixing and printing two silicones with different elastic moduli (graded composition) in combination with varying lattice geometry.

5.4 Summary

The findings of both major studies were analyzed and discussed. It was found that mixing efficiency is greater for the 50/50 sample at 2000 RPM than the 75/25 specimen. Additionally, an increase in tungsten concentration was evident across all radiographs and images. Further characterization and statistical work must be done to evaluate impeller velocity as a key component in mixing efficiency. Results from the durometer study were also presented. As suspected, sample hardness increased with decreased lattice spacing and vice versa. The data was modeled after a simple supported beam where deflection from a transverse load occurs with little force as beam supports move further apart.

REFERENCES

[1] Kruth, J.P., Leu, M.C., and Nakagawa, T. Progress in Additive Manufacturing and Rapid Prototyping. CIRP Annals – Manufacturing Technology 1998, 47, 2, 525-540.

[2] Kunc, Vlastimil, Brian Post, Lonnie Love, John Lindahl, Ahmed Hassen, and Ryan Dehoff. "3D Printed Tool for Building Aircraft Achieves Guinness World Records Title." *3D Printed Tool for Building Aircraft Achieves Guinness World Records Title | ORNL*. Oak Ridge National Laboratory, 29 Aug. 2016. Web. 03 Jan. 2017.

[3] Lewis, J. A. J. Am. Ceram. Soc. 2000, 83 (10), 2341−59.

[4] Tohver, V.; Smay, J. E.; Bream, A.; Braun, P. V.; Lewis, J. A. PNAS 2001, 98, 8950−54.

[5] B. H. Cumpston, S. P. Ananthavel, S. Barlow, D. L. Dyer, J. E. Ehrlich, L. L. Erskine, A. A. Heikal, S. M. Kuebler, I. Y. S. Lee, D. McCord-Maughon, J. Qin, H. Rockel, M. Rumi, X.-L. Wu, S. R., Marder, J. W. Perry, *Nature* **1999**, *398*, 51.

[6] G. M. Gratson, M. Xu, J. A. Lewis, *Nature* **2004**, *428*, 386.

[7] Compton, B. G. and Lewis, J. A. (2014), 3D-Printing of Lightweight Cellular Composites. Adv. Mater., 26: 5930–5935. doi:10.1002/adma.201401804

[8] Scheithauer, Uwe, Eric Schwarzer, Hans-JÃrgen Richter, and Tassilo Moritz. "Thermoplastic 3D Printing-An Additive Manufacturing Method for Producing Dense Ceramics." International Journal of Applied Ceramic Technology 12.1 (2014): 26-31. Web.

[9] B. Utela, D. Storti, R. Anderson, and M. Ganter, "A review of process development steps for new material systems in three dimensional printing (3DP)," Journal of Manufacturing Processes, vol. 10, no. 2, pp. 96–104, 2008.

[10] Lewis, J. A. (2006). Direct ink writing of 3D functional materials. *Advanced Functional Materials, 16*(17), 2193-2204.

[11] Witek, L. (2015). "*Extrusion-based, three-dimensional printing of calcium-phosphate scaffolds"* (Order No. 10157323). Available from Dissertations & Theses @ Oklahoma State University - Stillwater; ProQuest Dissertations & Theses Global. (1837110646).

[12] Bose, S., Vahabzadeh, S., Bandyopadhyay, A. Bone Tissue Engineering Using 3D Printing. Materials Today. 2013; 16(12): 496-504.

[13] A. Safari and E. K. Akdogan, ‘‘Rapid Prototyping of Novel Piezoelectric Composites,’’ Ferroelectrics, 331, 153–79 (2006).

[14] J. E. Smay, J. Cesarano III, B. A. Tuttle, and J. A. Lewis, ‘‘Piezoelectric Properties of 3-X Periodic Pb(ZrxTi1_x)O3-Polymer Composites,’’ J. Appl. Phys., 92, 6119–27 (2002).

[15] Wehner, Michael, Ryan L. Truby, Daniel J. Fitzgerald, Bobak Mosadegh, George M. Whitesides, Jennifer A. Lewis, and Robert J. Wood. "An Integrated Design and Fabrication Strategy for Entirely Soft, Autonomous Robots." *Nature* 536.7617 (2016): 451-55. Web.

[16] Zhu, C. (2010). Shape evolution of three-dimensional periodic structure fabricated by direct-write assembly of concentrated colloidal gels (Order No. 3443620). Available from Dissertations & Theses @ Oklahoma State University - Stillwater; ProQuest Dissertations & Theses Global. (855816504).

[17] Smay, J.E., S.S. Nadkarni, and J. Xu,. “Direct Writing of Dielectric Ceramics and Base Metal Electrodes.” International Journal of Applied Ceramic Technology, 2007. 4(1): p. 47-52.

[18] RoboCAD [Computer Software]. (2016) J.E. Smay.

[19] Hessel, V., Löwe, H., and Friedhelm, S., 2005. Micromixers – a review on passive and active mixing principles. Chemical Engineering Science, vol. 60, pp. 2479-2501, 2005.

[20] Liu, R.H., et al., 2002. Bubble-induced acoustic micromixing. Lab on a Chip 2, 151–157.

[21] Voldman, J., Gray, M.L., Schmidt, M.A., 1998. Liquid mixing studies with an integrated mixer/valve. In: Harrison, J., van den Berg, A. (Eds.) Micro Total Analysis Systems. Kluwer Academic Publishers, Dordrecht, pp. 181–184.

[22] L. Lu, K. S. Ryu and C. Liu, “A Magnetic Microstirrer and Array for Microfluidic Mixing”, Journal of Microelectromechanical Systems, vol. 11, no. 5, pp. 462- 469, 2002.

[23] Glasgow, I., Aubry, N., 2003. Enhancement of microfluidic mixing using time pulsing. Lab on a Chip 3, 114–120.

[24] Ober, Thomas J., Daniele Foresti, and Jennifer A. Lewis. "Active Mixing of Complex Fluids at the Microscale." Proceedings of the National Academy of Sciences 112.40 (2015): 12293-2298. Web.

[25] Active Mixer Design. Mixing Chamber Detail Schematic. J.E. Smay

[26] Active Mixer Design. Overall Schematic. J.E. Smay

[27] Fried, Joel R. "5 - Viscoelasticity and Rubber Elasticity." Polymer Science and Technology. Englewood Cliffs, NJ: Prentice Hall PTR, 1995. 182-228. Print.

[28] "Dow Corning Cookie Policy." Basics of Silicone Elastomers – Properties, Applications & Condensation Chemistry - Dow Corning. N.p., n.d. Web. 05 Jan. 2017.

[29] "Dow Corning Cookie Policy." DOW CORNING® SE 1700. N.p., n.d. Web. 05 Jan. 2017.

[30] Hardin, James O., Thomas J. Ober, Alexander D. Valentine, and Jennifer A. Lewis. "3D Printing: Microfluidic Printheads for Multimaterial 3D Printing of Viscoelastic Inks (Adv. Mater. 21/2015)." *Advanced Materials* 27.21 (2015): 3278. Web.

[31] Stroock, Abraham D., Stephan Dertinger, Armand Ajdari, Igor Mezic, Howard Stone, and George Whitesides. "Chaotic Mixer for Microchannels." *Science* 295.5555 (2002): 647-51. Web.

[32] Therriault, Daniel, Scott R. White, and Jennifer A. Lewis. "Chaotic Mixing in Three-dimensional Microvascular Networks Fabricated by Direct-write Assembly." *Nature Materials* 2.4 (2003): 265-71. Web.

[33] Williams, Manda S., Kenneth J. Longmuir, and Paul Yager. "A Practical Guide to the Staggered Herringbone Mixer." *Lab on a Chip* 8.7 (2008): 1121. Web.

[34] Mensing, G. A., T. M. Pearce, M. D. Graham, and D. J. Beebe. "An Externally Driven Magnetic Microstirrer." *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences* 362.1818 (2004): 1059-068. Web.

[35] Meijer, Han E.h., Mrityunjay K. Singh, Tae Gong Kang, Jaap M.j. Den Toonder, and Patrick D. Anderson. "Passive and Active Mixing in Microfluidic Devices." *Macromolecular Symposia*279.1 (2009): 201-09. Web.

[36] Choi, Jin-Woo, Chien-Chong Hong, and Chong H. Ahn. "An Electrokinetic Active Micromixer."*Micro Total Analysis Systems 2001* (2001): 621-22. Web.

[37] Young, Robert J., and P. A. Lovell. *Introduction to Polymers*. Boca Raton: CRC, 2011. Print.

[38] Vincent, Julian. "Basic Elasticity and Viscoelasticity." *Structural Biomaterials*. N.p.: Princeton UP, 2012. N. pag. Print.

[39] "Mold Making & Casting Materials | Rubbers, Plastics, Foams & More!" *Smooth-On, Inc.* N.p., n.d. Web. 04 Apr. 2017.

[40] "Materials Engineering." *Main_page [SubsTech]*. N.p., 10 July 2016. Web. 04 Apr. 2017.

[41] Duoss, Eric B., Todd H. Weisgraber, Keith Hearon, Cheng Zhu, Ward Small, Thomas R. Metz, John J. Vericella, Holly D. Barth, Joshua D. Kuntz, Robert S. Maxwell, Christopher M. Spadaccini, and Thomas S. Wilson. "Cellular Solids: Three-Dimensional Printing of Elastomeric, Cellular Architectures with Negative Stiffness (Adv. Funct. Mater. 31/2014)."*Advanced Functional Materials* 24.31 (2014): 5020. Web.

[42] ASTM D2240-15, Standard Test Method for Rubber Property—Durometer Hardness, ASTM International, West Conshohocken, PA, 2015, www.astm.org

[43] Buehler, Markus. "Continuum and Particle Methods." MIT - Lecture 4. Massachusetts, Cambridge. Web.

[44] V. Rohit, "Structural analysis of poly ethylene terephthalate bottles using the finite element method", Oklahoma State University, ProQuest, UMI Dissertations Publishing, 1513395, (2012).

[45] J.C. Hanan, "Plastic container having sidewall ribs with varying depth", United States patent, US 2013/0140264 A1

[46] J.C. Hanan, "Preform extended finish for processing light weight ecologically beneficial bottles", United States patent, US 2012/0263902A1

APPENDIX A: C++ CODE

```
/*-------------------------------------------------------------------------
  Arduino library to determine the Arduino models and features,
  as well as the SDK version.

  Most features can be accessed via static variables.
  You must instantiate if you want to know if the name of the board
  or if specific features such exist, for example multiple serial
  connections on the Arduino Mega.

  This list may be neither comprehensive nor up to date

  A full list of boards and processor names are available on Wikipedia:
  https://en.wikipedia.org/wiki/List_of_Arduino_boards_and_compatible_systems

  @author Tony Gaitatzis backupbrain@gmail.com
  @date 2015-12-10

  -------------------------------------------------------------------------
  This file is part of the Arduino Board Manager library

  NeoPixel is free software: you can redistribute it and/or modify
  it under the terms of the GNU Lesser General Public License as
  published by the Free Software Foundation, either version 3 of
  the License, or (at your option) any later version.

  NeoPixel is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public
  License along with NeoPixel.  If not, see
  <http://www.gnu.org/licenses/>.
  -------------------------------------------------------------------------*/

#include "ArduinoBoardManager.h"

ArduinoBoardManager::ArduinoBoardManager() {
  // clear the features
  memset(FEATURES, false, NUM_FEATURES);
```

```
static const unsigned long BOARD_UNKNOWN = 0x0;
static const unsigned long BOARD_UNO = 0x01;
static const unsigned long BOARD_ZERO = 0x02;
static const unsigned long BOARD_DUE = 0x03;
static const unsigned long BOARD_MICRO= 0x04;
static const unsigned long BOARD_YUN_400 = 0x05;
static const unsigned long BOARD_LEONARDO = 0x06;
static const unsigned long BOARD_MEGA = 0x07;
static const unsigned long BOARD_NANO = 0x08;
static const unsigned long BOARD_NANO_3 = 0x09;
static const unsigned long BOARD_LILYPAD = 0x08;
static const unsigned long BOARD_TRINKET = 0x09;


switch (ArduinoBoardManager::BOARD) {
  case ArduinoBoardManager::BOARD_UNO:
    strcpy(BOARD_NAME, "UNO");
    strcpy(CPU_NAME, "ATmega328P");
  break;
  case ArduinoBoardManager::BOARD_ZERO:
    strcpy(BOARD_NAME, "Zero");
    strcpy(CPU_NAME, "ATSAMD21G18A");
    FEATURES[ArduinoBoardManager::FEATURE_ANALOG_OUT] = true;
  break;
  case ArduinoBoardManager::BOARD_DUE:
    strcpy(BOARD_NAME, "Due");
    strcpy(CPU_NAME, "ATSAM3X8E");
    FEATURES[ArduinoBoardManager::FEATURE_ANALOG_OUT] = true;
  break;
  case ArduinoBoardManager::BOARD_MICRO:
    strcpy(BOARD_NAME, "Micro");
    strcpy(CPU_NAME, "Atmega32U4");
  break;
  case ArduinoBoardManager::BOARD_YUN_400:
    strcpy(BOARD_NAME, "Yun");
    strcpy(CPU_NAME, "AR9331");
  break;
  case ArduinoBoardManager::BOARD_LEONARDO:
    strcpy(BOARD_NAME, "Leonardo");
    strcpy(CPU_NAME, "ATmega16U4");
  break;
  case ArduinoBoardManager::BOARD_MEGA:
    strcpy(BOARD_NAME, "Mega");
    strcpy(CPU_NAME, "ATmega1280");
    FEATURES[ArduinoBoardManager::FEATURE_MULTIPLE_SERIAL] = true;
  break;
  case ArduinoBoardManager::BOARD_NANO:
    strcpy(BOARD_NAME, "Nano");
    strcpy(CPU_NAME, "ATmega168");
  break;
  case ArduinoBoardManager::BOARD_NANO_3:
    strcpy(BOARD_NAME, "Nano");
    strcpy(CPU_NAME, "ATmega328");
  break;
  case ArduinoBoardManager::BOARD_LILYPAD:
    strcpy(BOARD_NAME, "Lilypad");
    strcpy(CPU_NAME, "ATmega168V");
  break;
```

```
    case ArduinoBoardManager::BOARD_LILYPAD_2:
      strcpy(BOARD_NAME, "Lilypad");
      strcpy(CPU_NAME, "ATmega328V");
    break;
    case ArduinoBoardManager::BOARD_TRINKET:
      strcpy(BOARD_NAME, "Trinket");
      strcpy(CPU_NAME, "ATTiny85");
    break;
    case ArduinoBoardManager::BOARD_101:
      strcpy(BOARD_NAME, "101");
      strcpy(CPU_NAME, "ARCv2EM");
      FEATURES[ArduinoBoardManager::FEATURE_BLUETOOTH_4] = true;
      FEATURES[ArduinoBoardManager::FEATURE_ACCELEROMETER] = true;
      FEATURES[ArduinoBoardManager::FEATURE_GYROSCOPE] = true;
    break;
    default:
      strcpy(BOARD_NAME, "Unknown");
      strcpy(CPU_NAME, "Unknown");

  }

}


bool ArduinoBoardManager::featureExists(uint8_t feature) {
  if ((feature < ArduinoBoardManager::NUM_FEATURES) &&
     (ArduinoBoardManager::FEATURES[feature]))
      return true;
  return false;
}

/*-------------------------------------------------------------------------
  Arduino library to determine the Arduino models and features,
  as well as the SDK version.

  Most features can be accessed via static variables.
  You must instantiate if you want to know if the name of the board
  or if specific features such exist, for example multiple serial
  connections on the Arduino Mega.

  This list may be neither comprehensive nor up to date

  A full list of boards and processor names are available on Wikipedia:
  https://en.wikipedia.org/wiki/List_of_Arduino_boards_and_compatible_systems

  @author Tony Gaitatzis backupbrain@gmail.com
  @date 2015-12-10

  -------------------------------------------------------------------------
  This file is part of the Arduino Board Manager library

  NeoPixel is free software: you can redistribute it and/or modify
  it under the terms of the GNU Lesser General Public License as
  published by the Free Software Foundation, either version 3 of
  the License, or (at your option) any later version.

  NeoPixel is distributed in the hope that it will be useful,
  but WITHOUT ANY WARRANTY; without even the implied warranty of
```

```
  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
  GNU Lesser General Public License for more details.

  You should have received a copy of the GNU Lesser General Public
  License along with NeoPixel.  If not, see
  <http://www.gnu.org/licenses/>.
  -----------------------------------------------------------------------*/

#if ARDUINO >= 100
  #include <Arduino.h>
#else
  #include <WProgram.h>
#endif

class ArduinoBoardManager {
 public:
  /**
   * Arduino IDE/SDK Version
   */
  static const uint16_t SDK_VERSION = ARDUINO;        /**< Arduino SDK Version */

  /**
   * Board Name
   */
  static const uint8_t MAX_BOARD_NAME_LENGTH = 16;
  static const uint8_t MAX_CPU_NAME_LENGTH = 16;
  char BOARD_NAME[MAX_BOARD_NAME_LENGTH];                       /**< When
instantiated, this is the board name, eg "UNO" */
  char CPU_NAME[MAX_CPU_NAME_LENGTH];                         /**< When
instantiated, this is the cpu name, eg "__AVR_ATmega328P__" */

  /**
   * Known Board Models
   */
  static const uint8_t BOARD_UNKNOWN = 0x0;
  static const uint8_t BOARD_UNO = 0x01;
  static const uint8_t BOARD_ZERO = 0x02;
  static const uint8_t BOARD_DUE = 0x03;
  static const uint8_t BOARD_MICRO= 0x04;
  static const uint8_t BOARD_YUN_400 = 0x05;
  static const uint8_t BOARD_LEONARDO = 0x06;
  static const uint8_t BOARD_MEGA = 0x07;
  static const uint8_t BOARD_NANO = 0x08;
  static const uint8_t BOARD_NANO_3 = 0x09;
  static const uint8_t BOARD_LILYPAD = 0x0a;
  static const uint8_t BOARD_LILYPAD_2 = 0x0b;
  static const uint8_t BOARD_TRINKET = 0x0c;
  static const uint8_t BOARD_101 = 0x0d;

  /**
   * Known Arduino Features
   */
  static const uint8_t NUM_FEATURES = 1;
  static const uint8_t FEATURE_MULTIPLE_SERIAL = 0x00;
  static const uint8_t FEATURE_BLUETOOTH_4 = 0x01;
  static const uint8_t FEATURE_ACCELEROMETER = 0x02;
  static const uint8_t FEATURE_GYROSCOPE = 0x03;
  static const uint8_t FEATURE_ANALOG_OUT = 0x04;
```

```
  /**
   * CPU speed
   */
  static const unsigned long MAX_MHZ = F_CPU;

  boolean FEATURES[NUM_FEATURES];

  /**
   * CPU Specifications
   */
#if defined(__AVR_ATmega328P__) // uno, fio
  static const uint8_t BOARD = 0x01;           /**< UNO board */
  static const uint8_t NUM_BITS = 8;            /**< 8-bit processor */
  static const uint16_t CPU = __AVR_ATmega328P__;     /**< 16Mhz */
  static const unsigned long SRAM_SIZE = 2000;       /**< 2kb of sram */
  static const unsigned long EEPROM_SIZE = 1000;     /**< 1kb eeprom */
  static const unsigned long FLASH_SIZE = 32000;     /**< 32k flash storage */
#elif defined(__AVR_ATSAMD21G18A__) // zero
  static const uint8_t BOARD = 0x02
  static const uint8_t NUM_BITS = 8;
  static const uint16_t CPU = __AVR_ATSAMD21G18A__;
  static const unsigned long SRAM_SIZE = 32000;
  static const unsigned long EEPROM_SIZE = 16000;
  static const unsigned long FLASH_SIZE = 256000;
#elif defined(__AVR_ATSAM3X8E__) // Due
  static const uint8_t BOARD = 0x03;
  static const uint8_t NUM_BITS = 8;
  static const uint16_t CPU = __AVR_ATSAMD21G18A__;
  static const unsigned long SRAM_SIZE = 96000;
  static const unsigned long EEPROM_SIZE = 0;
  static const unsigned long FLASH_SIZE = 512000;
#elif defined(__AVR_Atmega32U4__) // Yun 16Mhz, Micro, Leonardo, Esplora
  static const uint8_t BOARD = 0x04;
  static const uint8_t NUM_BITS = 8;
  static const uint16_t CPU = __AVR_Atmega32U4__;
  static const unsigned long SRAM_SIZE = 2500;
  static const unsigned long EEPROM_SIZE = 1000;
  static const unsigned long FLASH_SIZE = 32000;
#elif defined(_AVR_AR9331__) // Yun 400Mhz
  static const uint8_t BOARD = 0x05;
  static const uint8_t NUM_BITS = 8;
  static const uint16_t CPU = _AVR_AR9331__;
  static const unsigned long SRAM_SIZE = 64000000;
  static const unsigned long EEPROM_SIZE = 0;
  static const unsigned long FLASH_SIZE = 16000000;
#elif defined(__AVR_ATmega16U4__) // leonardo
  static const uint8_t BOARD = 0x06;
  static const uint8_t NUM_BITS = 8;
  static const uint16_t CPU = __AVR_ATmega16U4__;
  static const unsigned long SRAM_SIZE = 2560;
  static const unsigned long EEPROM_SIZE = 1000;
  static const unsigned long FLASH_SIZE = 32000;
#elif defined(__AVR_ATmega1280__) || defined(__AVR_ATmega2560__) // mega, Mega ADK
  static const uint8_t BOARD = 0x07;
  static const uint8_t NUM_BITS = 8;
  static const uint16_t CPU = __AVR_ATmega2560__;
  static const unsigned long SRAM_SIZE = 8000;
```

```
  static const unsigned long EEPROM_SIZE = 4000;
  static const unsigned long FLASH_SIZE = 256000;
#elif defined(_AVR_ATmega328__) // Nano >= v3.0 or Arduino Pro, pro328, ethernet
  static const uint8_t BOARD = 0x08;
  static const uint8_t NUM_BITS = 8;
  static const uint16_t CPU = _AVR_ATmega328__;
  static const unsigned long SRAM_SIZE = 2000;
  static const unsigned long EEPROM_SIZE = 1000;
  static const unsigned long FLASH_SIZE = 32000;
#elif defined(_AVR_ATmega168__) // Nano < v3.0 or uno, pro
  static const uint8_t BOARD = 0x09;
  static const uint8_t NUM_BITS = 8;
  static const uint16_t CPU = _AVR_ATmega168;
  static const unsigned long SRAM_SIZE = 1000;
  static const unsigned long EEPROM_SIZE = 500;
  static const unsigned long FLASH_SIZE = 16000;
#elif defined(_AVR_ATmega168V__) // LilyPad
  static const uint8_t BOARD = 0x0a;
  static const uint8_t CPU = _AVR_ATmega168V__;
  static const unsigned int NUM_BITS = 8;
  static const unsigned long SRAM_SIZE = 1000;
  static const unsigned long EEPROM_SIZE = 500;
  static const unsigned long FLASH_SIZE = 14000
#elif defined(_AVR_ATmega328V__) // LilyPad 2
  static const uint8_t BOARD = 0x0b;
  static const uint8_t CPU = _AVR_ATmega328V__;
  static const unsigned int NUM_BITS = 8;
  static const unsigned long SRAM_SIZE = 1000;
  static const unsigned long EEPROM_SIZE = 500;
  static const unsigned long FLASH_SIZE = 14000
#elif defined(_AVR_ATTiny85__) // trinket
  static const uint8_t BOARD = 0x0c;
  static const uint8_t NUM_BITS = 8;
  static const uint16_t CPU = _AVR_ATTiny85__;
  static const unsigned long SRAM_SIZE = 500;
  static const unsigned long EEPROM_SIZE = 500;
  static const unsigned long FLASH_SIZE = 2500;
#elif  defined(__AVR_ARCv2EM__) || (__CURIE_FACTORY_DATA_H_) // Intel Curie/101
  static const uint8_t BOARD = 0x0d;
  static const uint8_t NUM_BITS = 32;
  static const uint16_t CPU = __AVR_ARCv2EM__;
  static const unsigned long SRAM_SIZE = 24000; // might be 80k?
  static const unsigned long EEPROM_SIZE = 0;
  static const unsigned long FLASH_SIZE = 384000;
#else
  static const uint8_t BOARD = 0x00;
  static const uint8_t NUM_BITS = 0;
  static const uint16_t CPU = 0;
  static const unsigned long SRAM_SIZE = 0;
  static const unsigned long EEPROM_SIZE = 0;
  static const unsigned long FLASH_SIZE = 0;
#endif

  /**
   * Instantiate board Manager
   */
  ArduinoBoardManager();
```

```
  /**
   * Ask if a specific feature exists, e.g. Arduino::FEATURE_MULTIPLE_SERIAL
   *
   * @param feature  an unsigned int, e.g. FEATURE_MULTIPLE_SERIAL
   * @return true if feature exists, false if feature does not exist
   */
  bool featureExists(uint8_t feature);

};

#include "UserTimer.h"
#include "QuadEncoder.h"
#include "PID_Loop.h"

#ifndef DCServoMotor_H
#define DCServoMotor_H
class DCServoMotor
{
public:
       volatile int n_FastPWMPin;  //the PWM control pin for the DC motor
controlled by fast PWM on timer 3
       volatile int n_DirPin;  //high/low to INV pin on motor driver board to
control direction
       volatile int n_Loops=0;
       volatile double d_TargetRPM=0;  //set by user with "SP xx.xx"
       volatile double d_CurrentRPM=0;
       volatile double d_RPMErr=0;
  volatile double d_Revs=0;
  volatile double d_Degrees=0;
  volatile double d_ArcMin=0;
  volatile double d_ArcSec=0;
       volatile bool o_ControlSpeed=true;
       PID_Loop * PIDL_Vel; //PID control loop for velocity
       UserTimer * PID_Clock;
       QuadEncoder * Enc;

       DCServoMotor();
       DCServoMotor(int EncoderPinA, int EncoderPinB, int QuadSteps, int ChAMode,
int ChBMode, int FastPWMPin, int DirPin, double PIDClockCycle);
       void Setup();
       void SetupTimers(double dDesiredPeriod);
       void Reset();
       void CalcCurrentRPM();
       void EchoCurrentRPM();
       void SetTargetRPM(double dNewRPM);
  void SetDir(int nDir);
       void EchoTargetRPM();
       void EchoDir();
       void EchoSetupInfo();
       void EchoSpeedControlPin();
       void EchoTimerSetup(int nTimer);
       void EchoDirPin();
       void EchoPID();
       void EchoMixerInfo();
       void ControlSpeed();
};
#endif
```

```
#include <Arduino.h>
#include <String.h>
#include "QuadEncoder.h"
#include "UserTimer.h"
#include "DCServoMotor.h"
#include "PID_Loop.h"
#include "MYSERIAL.h"

DCServoMotor::DCServoMotor()
{
	n_FastPWMPin = 9;  //the PWM control pin for the DC motor controlled by
fast PWM on timer 3
	n_DirPin = 10;  //high/low to INV pin on motor driver board to control
direction
	PID_Clock = new UserTimer(0.01);
	Enc= new QuadEncoder();
	PIDL_Vel = new PID_Loop();
}

DCServoMotor::DCServoMotor(int EncoderPinA, int EncoderPinB, int QuadSteps, int
ChAMode, int ChBMode, int FastPWMPin, int DirPin, double PIDClockCycle)
{
	Enc = new QuadEncoder(EncoderPinA, EncoderPinB, QuadSteps, ChAMode,
ChBMode);
	PID_Clock= new UserTimer(PIDClockCycle);
	PIDL_Vel = new PID_Loop();
	n_FastPWMPin = FastPWMPin;
	n_DirPin = DirPin;
	Setup();
}

void DCServoMotor::Reset()
{
	Enc->Reset();
	PID_Clock->Reset();
	PIDL_Vel->Reset();
	Setup();
}

void DCServoMotor::Setup()
{
	//set pin as output for FastPWM
	pinMode(n_FastPWMPin, OUTPUT);
	//set pin as output for Direction
	pinMode(n_DirPin, OUTPUT);
}

void DCServoMotor::CalcCurrentRPM()
{
	long l_Delta_ms = PID_Clock->delta_ms();
	d_CurrentRPM=l_Delta_ms>0?(Enc->GetDeltaRev() / (1.0*l_Delta_ms)) *
60000.0:d_CurrentRPM;
	if(d_CurrentRPM!=0.0&&l_Delta_ms>0)
	{
		Enc->ZeroDeltaCnts();
		PID_Clock->st_ms = millis();
	}
	d_RPMErr = d_CurrentRPM- d_TargetRPM;
```

```
        PIDL_Vel->d_Signal = d_CurrentRPM;
}

void DCServoMotor::EchoCurrentRPM()
{

        MYSERIAL.print("DEC=(");
        MYSERIAL.print(Enc->l_EncCnts);
        MYSERIAL.print("-");
        MYSERIAL.print(Enc->l_PrevEncCnts);
        MYSERIAL.print(")=");
        MYSERIAL.print(Enc->l_DeltaCnts);
        MYSERIAL.print("[");
        MYSERIAL.print(Enc->GetDeltaRev(),5);
        MYSERIAL.println("]");
        MYSERIAL.print("Dt=(");
        MYSERIAL.print(millis());
        MYSERIAL.print("-");
        MYSERIAL.print(PID_Clock->st_ms);
        MYSERIAL.print(")=");
        MYSERIAL.println(PID_Clock->delta_ms());
        CalcCurrentRPM();
        MYSERIAL.print("Current RPM=");
        MYSERIAL.println(d_CurrentRPM);
}

void DCServoMotor::SetTargetRPM(double dNewRPM)
{
        d_TargetRPM = dNewRPM;
        PIDL_Vel->ChangeSetPoint(d_TargetRPM);
        //EchoTargetRPM();
}

void DCServoMotor::EchoTargetRPM()
{
        MYSERIAL.print("Target RPM=");
        MYSERIAL.println(PIDL_Vel->d_SetPt);
}

void DCServoMotor::EchoDir()
{
        MYSERIAL.print("Direction=");
        MYSERIAL.println(Enc->o_Dir?"CCW":"CW");
}

void DCServoMotor::EchoSetupInfo()
{
        MYSERIAL.println("DC Servo Config:");
        MYSERIAL.print("\tFastPWMPin=:");
        MYSERIAL.println(n_FastPWMPin);
        MYSERIAL.print("\tDirPin=:");
        MYSERIAL.println(n_DirPin);
        Enc->EchoSetupInfo();
}

void DCServoMotor::EchoSpeedControlPin()
{
        MYSERIAL.print("FastPWMPin=");
```

```
        MYSERIAL.println(n_FastPWMPin);
}

void DCServoMotor::SetDir(int nDir)
{
  digitalWrite(n_DirPin,nDir);
  Enc->o_Dir=nDir;
}

void DCServoMotor::EchoDirPin()
{
        MYSERIAL.print("DirPin=");
        MYSERIAL.println(n_DirPin);
}

void DCServoMotor::EchoTimerSetup(int nTimer)
{
        PID_Clock->EchoSetupInfo(nTimer);
}

void DCServoMotor::EchoPID()
{
    //some feedback that is useful

    MYSERIAL.print("Delta cnts = ");
    MYSERIAL.print(Enc->l_EncCnts);
    MYSERIAL.print(" - ");
    MYSERIAL.print(Enc->l_PrevEncCnts);
    MYSERIAL.print(" = ");
    MYSERIAL.println(Enc->l_DeltaCnts);


         PIDL_Vel->EchoPID();

    MYSERIAL.print("Timer 3 Period=");
    MYSERIAL.println(PID_Clock->d_TimerPeriod, 4);

    EchoSetupInfo();
}

void DCServoMotor::EchoMixerInfo()
{
        //echo a single line "," delimited information about mixer
        //order is: CurrentRPM, AvgRPM, RPMSetPoint, Direction, KP, KI, KD, PGain,
IGain, DGain, d_DutyCycle, n_DutyCycle, n_FPWMLevels, d_FPWMFrequency,
d_Timer1Freq, l_Cnts, l_Pos, n_ChAPin, n_ChBPin, n_FPWMPin, n_DirPin, n_ChAMode,
nChBMode, d_CurrentErr, d_ErrAccum, d_ErrSlope, n_CntsPerRev
        String stOut="MixerInfo:";
        stOut += String(d_CurrentRPM, 2) + ", ";
        stOut += String(PIDL_Vel->d_AvgSignal, 2) + ", ";
        stOut += String(d_TargetRPM, 2) + ", ";
        stOut += String(Enc->o_Dir) + ", ";
        stOut += String(PIDL_Vel->d_KP, 6) + ", ";
        stOut += String(PIDL_Vel->d_KI, 6) + ", ";
        stOut += String(PIDL_Vel->d_KD, 6) + ", ";
        stOut += String(PIDL_Vel->d_PGain, 6) + ", ";
        stOut += String(PIDL_Vel->d_IGain, 6) + ", ";
        stOut += String(PIDL_Vel->d_DGain, 6) + ", ";
```

```
        stOut += String(PIDL_Vel->d_DutyCycle, 3) + ", ";
        stOut += String(PIDL_Vel->n_DutyCycle) + ", ";
        stOut += String(PIDL_Vel->n_FPWMLevels) + ", ";
        stOut += String(PIDL_Vel->d_FPWMFreq, 3) + ", ";
        stOut += String(1.0/PID_Clock->d_TimerPeriod, 3) + ", ";
        stOut += String(Enc->l_EncCnts) + ", ";
        stOut += String(Enc->l_EncPos) + ", ";
        stOut += String(Enc->n_ChA) + ", ";
        stOut += String(Enc->n_ChB) + ", ";
        stOut += String(n_FastPWMPin) += ", ";
        stOut += String(n_DirPin) += ", ";
        stOut += String(Enc->n_ChAMode) + ", ";
        stOut += String(Enc->n_ChBMode)+ ", ";
  stOut += String(PIDL_Vel->d_CurrentErr,6)+", ";
  stOut += String(PIDL_Vel->d_ErrAccum,6)+", ";
  stOut += String(PIDL_Vel->d_ErrSlope,6)+", ";
        stOut += String(Enc->n_CntsPerRev);
        MYSERIAL.println(stOut);
  //stOut="[";
  //for(int nI=0;nI<PIDL_Vel->n_PIDArrayLen;nI++){stOut+=String(PIDL_Vel-
>d_SignalArray[nI],2)+", ";}
  //stOut+="]";
  //MYSERIAL.println(stOut);
}

void DCServoMotor::ControlSpeed()  //this gets triggered upon each compare of
timer3
{
  if(!o_ControlSpeed){return;}
  CalcCurrentRPM();
  PIDL_Vel->d_Signal = d_CurrentRPM;
  PIDL_Vel->Control();
}

void DCServoMotor::SetupTimers(double dDesiredPeriod)
{
       cli(); //disables global interrupts
                //I'm using timer 3 to adjust the PWM duty cycle on timer 1
channel A
                //The frequency of timer 3 is about 100Hz after this setup.
                /*About Timers:  TCCR stands for "Timer/Counter Control
Register."
                * TCCR's are configured by setting values in two, 8-bit registers
A&B.  So TCCR3 is for timer 3.  TCCR3A is register A for timer 3.  TCCR3B is
register B for timer 3.
                * The bits of register A are labeled as follows: [bit
num]:defined constant name,read/write,initial value
                * [0]:WGMn0, R/W, 0
                * [1]:WGMn1, R/W, 0
                * [2]:COMnC0, R/W, 0
                * [3]:COMnC1, R/W, 0
                * [4]:COMnB0, R/W, 0
                * [5]:COMnB1, R/W, 0
                * [6]:COMnA0, R/W, 0
                * [7]:COMnA1, R/W, 0
                * For register B:
                * [0]:CSn0, R/W, 0
                * [1]:CSn1, R/W, 0
```

```
                * [2]:CSn2, R/W, 0
                * [3]:WGMn2, R/W, 0
                * [4]:WGMn3, R/W, 0
                * [5]:not used
                * [6]:ICES1, R/W, 0
                * [7]:ICNC1, R/W, 0
                *
                * The timer can be set to count as fast as the CPU cycles (16MHz
on Mega2560 chip) or slowed down by setting CS12, CS11, and CS10 bits.
                * Timer 3 is 16bit, so it can count from 0 to 2^16=65536.  The
count value at any time depends on how fast you are counting and if you
                * reset at the max value or something less than that.  If you
count at 16MHz and fill the counter it should take 65536/16x10^6=0.004096s.
                * You can make the timer tick slower by using a prescalar: e.g.
1024/16MHz=0.000064s cycle time.  Your prescalar can be 1, 8, 64, 256, 0r 1024.
                * If I use a 1024 prescaler and count until 16bits is filled:
T=65536*0.000064=4.194s
                * If I use a 256 prescaler and count until 16bits is filled:
T=65536*0.000016=1.0486s
                * If I use a 64 prescaler and count until 16bits is filled:
T=65536*0.000004=0.26214s
                * If I use a 8 prescaler and count until 16bits is filled:
T=65536*0.0000005=0.0327s
                */
                //****
                //First, setup timer 3 for my motor control PID
        TCCR3A = 0; //clear the setup register A
        TCCR3B = 0; //clear the setup register B

        //set prescalar to best match dDesiredPeriod
        int nCase;
  PID_Clock->d_DesiredPeriod=dDesiredPeriod;
        nCase = dDesiredPeriod < (4.194) ? 1024 : 1024;
        nCase = dDesiredPeriod < (1.0486) ? 256 : nCase;
        nCase = dDesiredPeriod < (0.2621) ? 64 : nCase;
        nCase = dDesiredPeriod < (0.0327) ? 8 : nCase;
        nCase = dDesiredPeriod < (0.0041) ? 1 : nCase;
        switch (nCase)
        {
        case 1:
                TCCR3B |= (1 << CS30);
                break;
        case 8:
                TCCR3B |= (1 << CS31);
                break;
        case 64:
                TCCR3B |= (1 << CS30) | (1 << CS31);
                break;
        case 256:
                TCCR3B |= (1 << CS32);
                break;
        case 1024:
                TCCR3B |= (1 << CS30) | (1 << CS32);
                break;
        }

        //Using an overflow compare method to interrupt when timer counts to this
value.
```

```
        OCR3A = (int)(dDesiredPeriod / (nCase / 16000000.0));
        PID_Clock->d_TimerPeriod = (double)OCR3A / (16000000.0 / nCase); //Should
be close to dDesiredPeriod
  //set to clear timer on compare match (CTC) mode:
        TCCR3B |= (1 << WGM32);

        //turn on overflow compare interrupt in timer interrupt mask for 3
        TIMSK3 |= (1 << OCIE3A);
        //****

        PIDL_Vel->SetupFPWM();
        sei();
}

#include "DCServoMotor.h"
#ifndef InterruptConfig_H
#define InterruptConfig_H
void SetupInterrupts();
void Interrupt_ChA();
void Interrupt_ChB();
void do_DutyCycle(char const * Input, int Len, int Start);
#endif
#include <Arduino.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "InterruptConfig.h"
#include "DCServoMotor.h"
#include "StringFunctions.h"
#include "QuadENcoder.h"
#include "MYSERIAL.h"

#include <stdio.h>
#ifndef _MYSERIAL_h
#define _MYSERIAL_h
#ifndef MYSERIAL_H
#define MYSERIAL_H
#define MYSERIAL Serial
#endif
#endif

#include "MYSERIAL.h"

#ifndef PID_Loop_H
#define PID_Loop_H

class PID_Loop
{
public:
        static const int n_PIDArrayLen= 100;
        volatile int n_DataPointer = 0;  //points to position in array of last
input data
        volatile int n_FPWMLevels=1024;
        volatile double d_FPWMFreq = 1000;
        volatile double d_SignalArray[n_PIDArrayLen];  //used in PID to calcualte
the average signal
  volatile double d_Sum=0.0;
        volatile double d_AvgSignal=0.0;  //the average signal value
```

```
        volatile double d_CurrentErr = 0.0;
        volatile double d_ErrArray[n_PIDArrayLen];  //used in PID loop to calculate
the average error
        volatile double d_AvgErr=0.0;
        volatile double d_ErrAccum=0.0;  //the accumulated error in the PID loop
  volatile double d_ErrSlope=0.0;
        volatile double d_DutyCycle = 0; //0 to 100%
        volatile int n_DutyCycle=0;  //the duty cycle value where output switches
from high to low

        volatile double d_SetPt=0.0;
        volatile double d_Signal=0.0;

        volatile double d_KP = 1;  //the proportional gain in the PID loop
        volatile double d_KI = 0.01;  //the integral gain in the PID Loop
        volatile double d_KD = 0.1;  //the derivative gain in the PID loop

        volatile double d_PGain = 0;
        volatile double d_IGain = 0;
        volatile double d_DGain = 0;

        volatile double d_Divider = 1.0;  //a factor to divide the PID gain
        volatile bool o_PIDFeedback = false;

        void Reset();
        void EchoKP();
        void EchoKI();
        void EchoKD();
        void EchoPID();
        void EchoDutyCycle();
  double ConstrainDouble(double val, double Min, double Max);
        int ConstrainInt(int val, int Min, int Max);
        void SetupFPWM();
        void SetDutyCycle();
  void ChangeSetPoint(double dNewSP);
        void Control();
};
#endif // !PID_Loop_H

#include "PID_Loop.h"
#include "MYSERIAL.h"
#include <arduino.h>

void PID_Loop::EchoKP()
{
        MYSERIAL.print("KP=");
        MYSERIAL.println(d_KP,5);
}

void PID_Loop::EchoKI()
{
        MYSERIAL.print("KI=");
        MYSERIAL.println(d_KI,5);
}

void PID_Loop::EchoKD()
{
        MYSERIAL.print("KD=");
```

```
        MYSERIAL.println(d_KD,5);
}

void PID_Loop::EchoDutyCycle()
{
        MYSERIAL.print("DutyCycle=");
        MYSERIAL.println(d_DutyCycle, 5);
        MYSERIAL.print("OCR1A=");
        MYSERIAL.println(OCR1A);
        MYSERIAL.print("OCR1B=");
        MYSERIAL.println(OCR1B);
}

void PID_Loop::EchoPID()
{
        //some feedback that is useful
        MYSERIAL.print("Current RPM = ");
        MYSERIAL.print(d_Signal);
        MYSERIAL.print("[");
        MYSERIAL.print(d_SetPt);
        MYSERIAL.println("]");

        MYSERIAL.print("Current Error = ");
        MYSERIAL.println(d_CurrentErr);

        MYSERIAL.print("Accumulated Error = ");
        MYSERIAL.println(d_ErrAccum);

  MYSERIAL.print("Error Slope = ");
  MYSERIAL.println(d_ErrSlope);

        MYSERIAL.print("P Gain=");
        MYSERIAL.print(d_PGain, 5);

        MYSERIAL.print(", I Gain=");
        MYSERIAL.print(d_IGain, 5);

        MYSERIAL.print(", D Gain=");
        MYSERIAL.println(d_DGain, 5);

        MYSERIAL.print("Duty Cycle = ");
        MYSERIAL.print(n_DutyCycle);
        MYSERIAL.print(" [");
        MYSERIAL.print(n_FPWMLevels);
        MYSERIAL.println("]");
}

double PID_Loop::ConstrainDouble(double val, double Min, double Max)
{
  double dOut=val<Min?Min:val;
  return dOut>Max?Max:dOut;
}

int PID_Loop::ConstrainInt(int val, int Min, int Max)
{
  int nOut=val<Min?Min:val;
  return nOut>Max?Max:nOut;
}
```

```
void PID_Loop::SetupFPWM()
{
	//Now, setup fast PWM on Timer 1
	TCCR1B = 0; //clear register B
	TCCR1A = 0; //clear register A

	//setting up 10bit fast pwm
	TCCR1A |= (1 << WGM10) | (1 << WGM11); //set to 10 bit fast PWM mode
	TCCR1B |= (1 << WGM12);
	TCCR1B |= (1 << CS10); //set prescalar to 1 so counts at full speed
	d_FPWMFreq = 16000000.0 / (1 << 10)/1.0; //Frequency=16MHz/1024=15.6kHz;
	TCCR1A |= (1 << COM1A1); //enable n_FastPWMPin for fast PWM (pin 9 on
Arduino micro)
	n_FPWMLevels = (1<<10); //set up 10 bit fast PWM, total count is 1024 (0-
1023).
	OCR1A = 0; //I will adjust value of OCR1A to control the fast pwm on
control pin A.
}

void PID_Loop::SetDutyCycle()
{
	d_DutyCycle=ConstrainDouble(d_DutyCycle,0.0,100.0);
	n_DutyCycle = (d_DutyCycle/ 100.0) * (n_FPWMLevels);
	n_DutyCycle=ConstrainInt(n_DutyCycle,0,n_FPWMLevels-1);
	OCR1A = n_DutyCycle;
	//EchoDutyCycle();
}

void PID_Loop::Reset()
{
	n_DataPointer = 0;  //points to position in array of last input data
	d_AvgSignal=0.0;  //the average signal value
  d_Sum=0;
	d_CurrentErr = 0.0;
	d_AvgErr=0.0;
	d_ErrAccum=0.0;  //the accumulated error in the PID loop
	d_DutyCycle = 0.0; //0 to 100%
	n_DutyCycle=0;  //the duty cycle value where output switches from high to
low

	d_SetPt=0.0;
	d_Signal=0.0;

	d_PGain = 0.0;
	d_IGain = 0.0;
	d_DGain = 0.0;

	SetDutyCycle();
}

void PID_Loop::ChangeSetPoint(double dNewSP)
{
 d_SetPt=dNewSP;
 d_ErrAccum=0.0;
}

void PID_Loop::Control()
```

```
{
	d_CurrentErr = (d_Signal-d_SetPt);  //calculates error
  d_CurrentErr*=abs(d_SetPt)>0.0?1.0/d_SetPt:1.0; //calculate error as fraction of
setpoint
  double d_AbsErr=abs(d_CurrentErr);
	double d_KPMult=d_AbsErr>0.2?1.0:0.1; //multiplies Kp by 10 if error
greater than 10%
  d_KPMult=d_AbsErr>0.4?10.0:d_KPMult; //multiplies Kp by 100 if error greater
than 25%
  double d_KIMult=1.0/1000.0;
	//double d_KDMult=abs(d_CurrentErr)>0.02?1.0:0.1;

	d_ErrAccum+=d_CurrentErr;

	d_ErrArray[n_DataPointer] = d_CurrentErr;
  d_Sum+=d_Signal-d_SignalArray[n_DataPointer];
	d_AvgSignal= d_Sum / n_PIDArrayLen;
	d_SignalArray[n_DataPointer] = d_Signal;

	double d_PrevErr = n_DataPointer == 0 ? d_ErrArray[n_PIDArrayLen-1] :
d_ErrArray[n_DataPointer - 1];
  d_ErrSlope=d_CurrentErr-d_PrevErr;
	n_DataPointer == (n_PIDArrayLen-1) ? n_DataPointer = 0 : n_DataPointer++;

	double dPID = 0;
	//proportional gain
	d_PGain = d_KP*d_CurrentErr*d_KPMult;
	//integral gain
	d_IGain = d_KI*d_ErrAccum;
	//dirivitive gain
	d_DGain = d_KD*d_ErrSlope;
	//a scale factor
	dPID = -(d_PGain + d_IGain + d_DGain);

  dPID=ConstrainDouble(dPID,-1.0,1.0);
	d_DutyCycle += dPID;
	SetDutyCycle();
}

#include "DCServoMotor.h"

#ifndef ProcessSerial_H
#define ProcessSerial_H
class SerialProcessor
{
public:
	char c_MYSERIALInBuffer[128]{};
	int n_BytesRead = 0;

	void SerialProcessor::Echo(char * c_array);
	void SerialProcessor::Echo(char * c_bytesRead, int nLen);
	void SerialProcessor::LookForRCADCmds(char * InBuf, int nLen, DCServoMotor
* DCSM);
	void SerialProcessor::ProcessSerial(DCServoMotor * DCSM);
};
#endif

#include "StringFunctions.h"
```

```
#include "InterruptConfig.h"
#include "string.h"
#include "MYSERIAL.h"
#include "ProcessSerial.h"
#include "ArduinoBoardManager.h"
#include <Arduino.h>

void SerialProcessor::Echo(char * c_bytesRead, int nLen)
{
      if(nLen<=0){return;} //zero length
      MYSERIAL.print("Echo[");
      MYSERIAL.print(nLen);
      MYSERIAL.print("]:");
      for(int nI=0;nI<nLen;nI++)
      {MYSERIAL.print(c_bytesRead[nI]);}
      MYSERIAL.println("");
}

void SerialProcessor::Echo(char * c_array)
{
       int nL = 0;
       while (nL<65)
       {
              if (c_array[nL] == '\0') { break; }
              nL++;
       }
       Echo(c_array, nL);
}

/*
 * Commands:
 * Timer - Echo TimerSetup
 * Hand - Echo "Shake"
 * Enc - Echo Encoder Setup Info
 * SC - 0 or 1 to turn off or on speec control
 * Type - Echo ABM.BOARD_NAME
 * DW - digital write (pin, value)
 * DR - digital read (pin)
 * AW - analog write (pin, value)
 * DC - set duty cycle on fast PWM pin
 * KI - set KI value; ?KI - echo KI value
 * KD - set KD value; ?KD - echo KD value
 * KP - set KP value; ?KP - echo KP value
 * CP - set FastPWM pin; ?CP - echo FastPWM pin;
 * PID - echo PID information
 * SP - set RPM; ?SP - echo current RPM
 * DCSMConfig - echo some setup information
 * DIR - echo direction
 * MI - echo mixer information
 */
void SerialProcessor::LookForRCADCmds(char * InBuf, int nLen, DCServoMotor * DCSM)
{
     //Echo(InBuf,nLen);

        //Look for mixer info
        int n_MI = InStr("MI", InBuf, 2, nLen);
        if (n_MI >= 0)
        {
```

```
            DCSM->EchoMixerInfo();
    }

 //Look for timer info
 int n_Timer=InStr("Timer", InBuf,5,nLen);
 if(n_Timer>=0)
 {
              DCSM->EchoTimerSetup(3);
 }


//Look for Hand
int n_Hand=InStr("Hand",InBuf,4,nLen);
if(n_Hand>=0)
{
  MYSERIAL.println("Shake");
  SetupInterrupts();
}

//echo encoder setup
int n_Enc=InStr("Enc",InBuf,3,nLen);
if(n_Enc>=0)
{
  DCSM->Enc->EchoSetupInfo();
}

//turn on/off PID loop
int n_SC=InStr("SC",InBuf,2,nLen);
if(n_SC>=0)
{
  DCSM->o_ControlSpeed=(bool)GetNextInt(InBuf, nLen, n_SC);
  MYSERIAL.print("Speed Control On=");MYSERIAL.println(DCSM->o_ControlSpeed);
}

//echo board name
int n_Type=InStr("Type", InBuf,5,nLen);
if(n_Type>=0)
{
       ArduinoBoardManager ABM = ArduinoBoardManager();
  MYSERIAL.println(ABM.BOARD_NAME);
}

//look for digital write command
int n_DW=InStr("DW", InBuf,2,nLen);
if(n_DW>=0)
{
  do_dw(InBuf, nLen, n_DW+2);
}

//look for digital read command
int n_DR=InStr("DR", InBuf,2,nLen);
if(n_DR>=0)
{
  do_dr(InBuf, nLen, n_DR+2);
}

//look for analog write command
int n_AW=InStr("AW", InBuf,2,nLen);
```

```
if(n_AW>=0)
{
  do_aw(InBuf, nLen, n_AW+2);
}

//set duty cycle on FPWM pin
int n_DC=InStr("DC", InBuf,2,nLen);
if(n_DC>=0)
{
  do_DutyCycle(InBuf, nLen, n_AW+2);
}

//set KI for PID control of servo
int n_KI=InStr("KI", InBuf,2,nLen);
if(n_KI>=0&&InBuf[n_KI-1]=='?')
{
            DCSM->PIDL_Vel->EchoKI();
}
     else if(n_KI>=0)
{
  double d_NewKI=GetNextDouble(InBuf, nLen, n_KI);
     DCSM->PIDL_Vel->d_KI = d_NewKI;
}

//set FPWM Control Pin for PID control of servo
int n_CP=InStr("CP", InBuf,2,nLen);
if(n_CP>=0&&InBuf[n_CP-1]=='?')
{
          DCSM->EchoSpeedControlPin();
}
     else if(n_CP>=0)
{
  int n_NewCP=GetNextInt(InBuf, nLen, n_CP);
  DCSM->n_FastPWMPin=n_NewCP;
}

//set dir Control Pin
int n_DP=InStr("DP", InBuf,2,nLen);
if(n_DP>=0&&InBuf[n_DP-1]=='?')
{
DCSM->EchoDirPin();
}
else if(n_DP>=0)
{
  int n_NewDP=GetNextInt(InBuf, nLen, n_DP);
  DCSM->n_DirPin=n_NewDP;
}

//set enc CHA Pin
int n_CHA=InStr("CHA", InBuf,3,nLen);
if(n_CHA>=0&&InBuf[n_CHA-1]=='?')
{

}
else if(n_CHA>=0)
{
  int n_NewCHA=GetNextInt(InBuf, nLen, n_CHA);
  DCSM->Enc->n_ChA=n_NewCHA;
```

```
  DCSM->Enc->Reset();
}

//set enc CHB Pin
int n_CHB=InStr("CHB", InBuf,3,nLen);
if(n_CHB>=0&&InBuf[n_CHB-1]=='?')
{

}
else if(n_CHB>=0)
{
  int n_NewCHB=GetNextInt(InBuf, nLen, n_CHB);
  DCSM->Enc->n_ChB=n_NewCHB;
  DCSM->Enc->Reset();
}

//set KP for PID control of servo
int n_KP=InStr("KP", InBuf,2,nLen);
if(n_KP>=0&&InBuf[n_KP-1]=='?')
{
          DCSM->PIDL_Vel->EchoKP();
}
     else if(n_KP>=0)
{
  double d_NewKP= GetNextDouble(InBuf, nLen, n_KP);
     DCSM->PIDL_Vel->d_KP = d_NewKP;
}

//set KD for PID control of servo
int n_KD=InStr("KD", InBuf,2,nLen);
if(n_KD>=0&&InBuf[n_KD-1]=='?')
{
          DCSM->PIDL_Vel->EchoKD();
}
     else if(n_KD>=0)
{
  double d_NewKD= GetNextDouble(InBuf, nLen, n_KD);
       DCSM->PIDL_Vel->d_KD = d_NewKD;
}

//get PID feedback
double n_PID=InStr("PID", InBuf,3,nLen);
if(n_PID>=0)
{
    DCSM->EchoPID();
}

//change speed
     int n_SP=InStr("SP", InBuf,2,nLen);
if(n_SP>=0&&InBuf[n_SP-1]=='?')
{
            DCSM->EchoCurrentRPM();
}
else if(n_SP>=0)
{
            double d_NewRPM=GetNextDouble(InBuf, nLen,n_SP);
  if(DCSM->PID_Clock->d_DesiredPeriod>0.001){DCSM->SetupTimers(0.001);}
            DCSM->SetTargetRPM(d_NewRPM);
```

```
    }

    //set direction
    int n_DIR=InStr("DIR", InBuf,2,nLen);
    if(n_DIR>=0&&InBuf[n_DIR-1]=='?')
    {
        DCSM->EchoDir();
    }
    else if(n_DIR>=0)
    {
      int nDir=GetNextInt(InBuf, nLen, n_DIR);
      DCSM->SetDir(nDir);
    }

    int n_DCSMConfig=InStr("DCSMConfig", InBuf,10,nLen);
    if(n_DCSMConfig>=0&&InBuf[n_DCSMConfig-1]=='?')
    {
              DCSM->EchoSetupInfo();
    }

}

void SerialProcessor::ProcessSerial(DCServoMotor * DCSM)
{
    bool o_EOL=false;
    char byteRead;
    while(MYSERIAL.available())
    {
      byteRead=MYSERIAL.read();
      if(byteRead=='\n'||byteRead=='\r')
      {
        o_EOL=true;
        break;
      }
      c_MYSERIALInBuffer[n_BytesRead]=byteRead;
      n_BytesRead++;
    }
    if(!o_EOL){return;}
    LookForRCADCmds(c_MYSERIALInBuffer,n_BytesRead, DCSM);

    n_BytesRead=0;
}

// QuadEncoder.h
#ifndef QuadEncoder_H
#define QuadEncoder_H
class QuadEncoder
{
public:
       //FIELDS
       //define the encoder channel pins
       volatile int n_ChA=0;
       volatile int n_ChB=0;
       volatile int n_CntsPerRev=1024;
       volatile int n_Divider=1;
       volatile int n_QuadCounts=1024;
       volatile int n_ChAMode=0; //(0=off, (1<<0)=RisingEdge, (1<<1)=FallingEdge,
(1<<2)=Rising&Falling Edges
```

```
        volatile int n_ChBMode=0; //(0=off, (1<<0)=RisingEdge, (1<<1)=FallingEdge,
(1<<2)=Rising&Falling Edges

        volatile bool A_Set=true;
        volatile bool A_Change=true;
        volatile bool B_Set=false;

        volatile long l_EncPos=0;  //the encoder position in counts (+ or -)
        volatile long l_EncCnts=0;  //the total encoder counts in the current timer
interval
        volatile long l_PrevEncCnts=0;  //the total encoder counts in the last
timer interval
        volatile long l_DeltaCnts=0;
        volatile bool o_Dir=true;

        //unsigned long l_StartMS;  //
        //unsigned long l_ElapsedMS;  //

enum ChMode { ChM_None = 0, ChM_Rising = (1 << 0), ChM_Falling = (1 << 1),
ChM_Both = (1 << 2) };

        //CONSTRUCTOR
        QuadEncoder();
        QuadEncoder(int EncoderPinA, int EncoderPinB, int QuadSteps, int ChAMode,
int ChBMode);

  //METHODS
        void Setup(int EncoderPinA, int EncoderPinB, int QuadSteps, int ChAMode,
int ChBMode);
        void Reset();
        void EchoSetupInfo();
        void doEncA();
        void doEncB();
        void ZeroDeltaCnts();
        double GetDeltaRev();
};
#endif

#include <Arduino.h>
#include "QuadEncoder.h"
#include "MYSERIAL.h"

QuadEncoder::QuadEncoder()
{
        //define the encoder channel pins
        Setup(4, 5, 512, (1<<0), 0);
        A_Change = true;
}

QuadEncoder::QuadEncoder(int EncoderPinA, int EncoderPinB, int QuadSteps, int
ChAMode, int ChBMode)
{
        Setup(EncoderPinA, EncoderPinB, QuadSteps, ChAMode, ChBMode);
        A_Change = true;
}

void QuadEncoder::Setup(int EncoderPinA, int EncoderPinB, int QuadSteps, int
ChAMode, int ChBMode)
```

```
{
	n_ChA = EncoderPinA;
	n_ChB = EncoderPinB;
	n_QuadCounts = QuadSteps;
	n_ChAMode = ChAMode;
	n_ChBMode = ChBMode;
	Reset();
}

void QuadEncoder::Reset()
{
	int n_Edges=0;
	n_Edges+=(n_ChAMode&ChM_Rising)>0?1:0;
	n_Edges+=(n_ChAMode&ChM_Falling)>0?1:0;
	n_Edges+=(n_ChAMode&ChM_Both)>0?2:0;
	n_Edges+=(n_ChBMode&ChM_Rising)>0?1:0;
	n_Edges+=(n_ChBMode&ChM_Falling)>0?1:0;
	n_Edges+=(n_ChBMode&ChM_Both)>0?2:0;

	n_CntsPerRev = n_QuadCounts*n_Edges/4.0;
	MYSERIAL.print("n_Edges=");
  MYSERIAL.println(n_Edges);
	MYSERIAL.print("b_QuadCounts=");
  MYSERIAL.println(n_QuadCounts);
  MYSERIAL.print("n_CntsPerRev=");
  MYSERIAL.println(n_CntsPerRev);
	//configure encoder channels for input
	pinMode(n_ChA, INPUT);
	pinMode(n_ChB, INPUT);

	A_Set = digitalRead(n_ChA);
	B_Set = digitalRead(n_ChB);

	l_EncPos = 0;
	l_EncCnts = 0;
	l_PrevEncCnts = 0;
	//EchoSetupInfo();
}

//interrupt for channel A of encoder
//A: HHHHLLLLHHHHLLLLHHHHH
//B: LLHHHHLLLLHHHHLLLLL
void QuadEncoder::doEncA()
{
	l_EncCnts++;
  switch (n_ChAMode)
  {
    case ChM_Rising:
      A_Set=1;
    break;
    case ChM_Falling:
      A_Set=0;
    break;
    case ChM_Both:
      A_Set=!A_Set;
    break;
  }
  o_Dir?l_EncPos++:l_EncPos--;
```

```
}

//interrupt for channel B of encoder
void QuadEncoder::doEncB()
{
  /*
        l_EncCnts++;
        if (B_Set)
        {
                if ((n_ChBMode&ChM_Falling)>0) //caught a falling edge of ChB
                {
                        B_Set = !B_Set;
                        A_Set = digitalRead(n_ChA);
                        A_Set ? l_EncPos-- : l_EncPos++;
                        o_Dir = !A_Set;
                }
        }
        else
        {
                if ((n_ChBMode&ChM_Rising)>0) //caught a rising edge of ChB
                {
                        B_Set = !B_Set;
                        A_Set = digitalRead(n_ChA);
                        A_Set ? l_EncPos++ : l_EncPos--;
                        o_Dir = A_Set;
                }
        }
 */
}

double QuadEncoder::GetDeltaRev()
{
        l_DeltaCnts = (l_EncCnts - l_PrevEncCnts);
        return l_DeltaCnts*1.0/n_CntsPerRev;
}

void QuadEncoder::ZeroDeltaCnts() { l_PrevEncCnts = l_EncCnts; }

void QuadEncoder::EchoSetupInfo()
{
    MYSERIAL.println("\t<Encoder Config>");
    MYSERIAL.print("\tEncoder Pins: A=");
    MYSERIAL.print(n_ChA);
    MYSERIAL.print(", B=");
    MYSERIAL.println(n_ChB);
    MYSERIAL.print("\tQuadrature Counts per rev=");
    MYSERIAL.println(n_QuadCounts);
    MYSERIAL.print("\tChAMode=");
    MYSERIAL.print(n_ChAMode);
    MYSERIAL.print(", ChBMode=");
    MYSERIAL.println(n_ChBMode);
    MYSERIAL.print("\tn_CntsPerRev=");
    MYSERIAL.println(n_CntsPerRev);
    MYSERIAL.print("l_EncCnts=");MYSERIAL.println(l_EncCnts);
    MYSERIAL.print("l_PrevEncCnts=");MYSERIAL.println(l_PrevEncCnts);
    MYSERIAL.print("l_EncPos=");MYSERIAL.println(l_EncPos);
    MYSERIAL.println("\t</Encoder Config>");
}
```

```
#include <Arduino.h>
#include <String.h>
#ifndef  StringFunctions_H
#define StringFunctions_H
#define CharArrayLen(array) (sizeof(array)/sizeof(char))
int InStr(char const * Sought, char const * Input, int nLenSought, int nLenInput);
int InStr(char const * Sought, char const * Input, int nStart, int nLenSought, int
nLenInput);
double GetNextDouble(char const * Input, int Len, int Start);
int GetNextInt(char const * Input, int Len, int Start);
int FindDelimeterPos(char Delim, char const * Input,int Len, int Start);
void do_dw(char const * Input, int Len, int Start);
bool do_dr(char const * Input, int Len, int Start);
void do_aw(char const * Input, int Len, int Start);
#endif // ! StringFunctions_H

#include <avr/io.h>
#include <Arduino.h>
#include <String.h>
#include "MYSERIAL.h"
#ifndef StringFunctions_CPP
#define StringFunctions_CPP
bool SameLtr(char c1, char c2, bool CaseSensitive)
{
  if(CaseSensitive){return c1==c2;}
  if(c1<97 )//capital letter
  {
    return c1==c2||(c1+32)==c2;
  }
  return c1==c2||(c1-32)==c2;
}


int InStr(char const * Sought, char const * Input, int nLenSought, int nLenInput)
{
/*Look for the first element of Sought in Input.
If found, compare remaining elements of Sought to Input
after the initial location in Input.
*/
       int nInput=nLenInput;
       int nSought=nLenSought;
       int nLocation=-1;
       for (int nJ=0;nJ<nInput;nJ++)
       {
              if(SameLtr(Sought[0],Input[nJ],false))
              {
                           nLocation = nJ;
                           break;
              }
       }
       if(nLocation<0){return -1;}
       for(int nJ=1;nJ<nSought;nJ++)
       {
              if(!SameLtr(Sought[nJ],Input[nJ+nLocation],false)){return -1;}
       }
       return nLocation;
}
```

```
int InStr(char const * Sought, char const * Input, int Start, int nLenSought, int
nLenInput)
{
	int nInput=nLenInput;
	int nSought=nLenSought;
	int nLocation=Start-1;
	for (int nJ=0;nJ<nSought;nJ++)
	{
		for(int nI=nLocation+1; nI<nInput;nI++)
		{
			if(SameLtr(Sought[nJ],Input[nI],false))
			{
				if(nJ>0&&nI>nLocation+2)
				{return -1;}
				nLocation = nJ;
				break;
			}
		}
	}
	return nLocation>Start?nLocation:-1;
}

double GetNextDouble(char const * Input, int Len, int Start)
{
  double d_Whole=0.0;
  double d_Fract=0.0;
  bool o_Radix=false;
  double d_Sign=1.0;
  bool o_Sign=false;  //can only have one sign character in a number and it must
be first character
  int n_Rad=1;
  int n_Start=-1;

	for(int nI=Start;nI<Len;nI++)
	{
		//check for negative sign
		if(Input[nI]=='-')
		{
			if(!o_Sign)
			{
				o_Sign=true;
				d_Sign=-1.0;
			}
			else
			{return d_Sign*(d_Whole+d_Fract);}
		}
		//find a radix; there can be only one
		else if(Input[nI]=='.')
		{
			if (!o_Radix)
			{
				o_Radix = true;
			}
			else
			{return d_Sign*(d_Whole + d_Fract);}
		}
		//see if input is a number; ASCII characters 47 to 58 are 0-9
```

```
                else if(Input[nI]>47&&Input[nI]<58)
                {
                        if (!o_Sign) { o_Sign - true; }
                        if(n_Start<0){n_Start=nI;}
                        if(!o_Radix){d_Whole=10.0*d_Whole+Input[nI]-48.0;}
                        if(o_Radix)
                        {
                                d_Fract=d_Fract+(Input[nI]-48.0)/(pow(10.0,n_Rad));
                                n_Rad++;
                        }
                }
                else if(Input[nI]==' '||Input[nI]=='='||Input[nI]==',')
                {}
                else if(n_Start>=0)
                {
                        return d_Sign*(d_Whole+d_Fract);
                }
        }
        //MYSERIAL.print("d_Whole=");
        //MYSERIAL.print(d_Whole,0);
        //MYSERIAL.print("d_Fract=");
        //MYSERIAL.println(d_Fract,5);
        return d_Sign*(d_Whole+d_Fract);
}

int GetNextInt(char const * Input, int Len, int Start)
{
    int n_Whole=0;
    int n_Sign=1;
    bool o_Sign=false;
    bool o_Start=false;
    for(int nI=Start;nI<Len;nI++)
    {
      //check for negative sign
      if(Input[nI]=='-')
      {
        if(!o_Sign)
        {
          o_Sign=true;
          n_Sign=-1;
        }
        else{return n_Sign*n_Whole;}
      }
      else if(Input[nI]>47&&Input[nI]<58)
      {
              if (!o_Sign) { o_Sign = true; }
              n_Whole=10*n_Whole+Input[nI]-48;
        o_Start=true;
      }
      else if(o_Start)
      {
        MYSERIAL.println("non-numeric found");
        return n_Sign*n_Whole;
      }
      else if(!o_Start)
      {
        n_Sign=1;
        o_Sign=false;
```

```
      }
    }
        n_Whole = n_Sign*n_Whole;
    //MYSERIAL.println(n_Whole);
    return n_Whole;
}

int FindDelimeterPos(char Delim, char const * Input,int Len, int Start)
{
  for(int nI=Start; nI<Len;nI++)
  {
    if(Input[nI]==Delim){return nI;}
  }
  return -1;
}

void do_dw(char const * Input, int Len, int Start)
{
    int n_Pin=GetNextInt(Input,Len, Start);
    int Start2=FindDelimeterPos(',',Input, Len, Start)+1;
    int n_State=GetNextInt(Input, Len, Start2);
    if(n_State==0){digitalWrite(n_Pin,LOW);}
    if(n_State>0){digitalWrite(n_Pin,HIGH);}
    MYSERIAL.print("digitalWrite(");
    MYSERIAL.print(n_Pin);
    MYSERIAL.print(",");
    MYSERIAL.print(n_State);
    MYSERIAL.print(")");
}

bool do_dr(char const * Input, int Len, int Start)
{
    int n_Pin=GetNextInt(Input,Len, Start);

    MYSERIAL.print("digitalRead(");
    MYSERIAL.print(n_Pin);
    MYSERIAL.print(",");
    bool o_State=digitalRead(n_Pin)   ;
    MYSERIAL.print(o_State);
    MYSERIAL.print(")");
    return o_State;
}

void do_aw(char const * Input, int Len, int Start)
{
    int n_Pin=GetNextInt(Input,Len, Start);
    int Start2=FindDelimeterPos(',',Input, Len, Start)+1;
    int n_State=GetNextInt(Input, Len, Start2);
    analogWrite(n_Pin,n_State);
    MYSERIAL.print("analogWrite(");
    MYSERIAL.print(n_Pin);
    MYSERIAL.print(",");
    MYSERIAL.print(n_State);
    MYSERIAL.print(")");
}

#endif // !StringFunctions_CPP
```

```
// UserTimer.h
#ifndef UserTimer_H
#define UserTimer_H
class UserTimer
{

public:
	volatile double d_TimerPeriod=0.01;  //the period of the PID loop in timer
1
	volatile double d_DesiredPeriod = 0.01;
	volatile long st_ms;

	UserTimer(double dDesiredPeriod);
	UserTimer();

	long delta_ms();
	void Reset();
	void EchoSetupInfo(int nTimer);
};
#endif

#include <Arduino.h>
#include <avr/io.h>
#include <avr/interrupt.h>
#include "UserTimer.h"
#include "MYSERIAL.h"
/*
Timers are somewhat complicated to setup.  They can be configured to just count
and cause an interrupt or
to control fast pulse width modulation for controlling devices.
I am using Timer 3 for counting for the PID control of a DC servo motor.
I am using Timer 1 for fast PWM.

Arduino Mega2560has the following pins controlled by the various timers:
timer 0 (controls pin 13, 4)
timer 1 (controls pin 12, 11)
timer 2 (controls pin 10, 9)
timer 3 (controls pin 5, 3, 2)
timer 4 (controls pin 8, 7, 6)

Arduino Micro (Atmel mega32u4 chip) has the following pins controlled by the
various timers:
timer 0 (controls pin A=11, B=3)
timer 1 (controls pin A=9, B=10, C=11)
timer 2 (controls pin A=?, B=?)
timer 3 (controls pin A=5, B=?)
timer 4 (controls pin A=13, B=?)
*/
UserTimer::UserTimer(const double dDesiredPeriod)
{
	d_DesiredPeriod = dDesiredPeriod;
	st_ms = millis();
}

UserTimer::UserTimer()
{
	d_DesiredPeriod = 0.1;
	st_ms = millis();
```

```
}

long UserTimer::delta_ms() { return millis() - st_ms; }

void UserTimer::Reset(){EchoSetupInfo(3);}

void UserTimer::EchoSetupInfo(int n_Num)
{
	switch (n_Num)
	{
	case 1:
		break;
	case 2:
		break;
	case 3:
		MYSERIAL.print("Timer");
		MYSERIAL.print(n_Num);
		MYSERIAL.println(" Setup:");
		MYSERIAL.print("\tTCCR3A=");
		MYSERIAL.println(TCCR3A);
		MYSERIAL.print("\tTCCR3B=");
		MYSERIAL.println(TCCR3B);
		MYSERIAL.print("\tOCR3A=");
		MYSERIAL.println(OCR3A);
		MYSERIAL.print("\td_TimerPeriod=");
		MYSERIAL.println(d_TimerPeriod,5);
		break;
	}
}
```

APPENDIX B: RADIAL DISTRIBUTION FUNCTION MACRO

```
// ImageJ macro to calculate the Radial Distribution Function (RDF) of particle centers
//
// Version 2011-08-22
//
// Input: Binary or 8-bit input image/stack with dark particles on light background.
// Grayscale/RGB images are OK as long as "Find Maxima" works reliably on them.
// For binary images/stacks, the macro does not care whether "black background"
// is selected in Process>Binary>Options.
//
// Output: Normalized RDF plot with distance in pixels. For stacks the mean is plotted.
//
// Known Issues, Updates and Examples at:
// http://imagejdocu.tudor.lu/doku.php?id=macro:radial_distribution_function
//
// Requirements: A working install of "Radial Profile" plugin is required. Get it at
// http://rsb.info.nih.gov/ij/plugins/radial-profile.html
//
// Limitations:
// - Particle positions are rounded to full pixel nearest to particle intensity maximum
```

```
// - RDF output distances are in pixels, irrespective of any spatial calibration of the image
// - RDF range is 0.3x the smallest dimension of the image
// - Particles touching the edge will be ignored; this will limit the accuracy
//    if the particles are not much smaller than the image size.
// - Do not extend the image size for avoiding edge effects; the macro takes care of this.
//
////////////////////////////////////////////////////////////////////////////////


macro "Radial Distribution Function [f5]" {
        run("Select None");
        doStack=false;
        //User dialog
        Dialog.create('RDF Options');
        Dialog.setInsets(0,0,0)
        Dialog.addMessage("Radial Distribution Function Macro \nby Michael Schmid & Ajay
Gopal \n(v.2011-08-21)");
        if (nSlices()>1) {
                Dialog.addMessage("Selected file is a stack. \nUncheck below to analyze \nonly
the current slice.");
                Dialog.addCheckbox("Use all slices in stack?", true);
        }
        Dialog.addMessage("Particle Detection Noise Threshold \nHint: test image/s first with
\nImageJ>Process>Find Maxima \nto verify that below threshold \ngives accurate particle
centers.");
        Dialog.addNumber("     Noise Threshold", 10);
        Dialog.addMessage("Default output is RDF plot with \noptions to list, save & copy data.
\nCheck below to output extra \nwindow with RDF data table.");
        Dialog.addCheckbox("Output RDF data table ", false);
        Dialog.show;
         //Preliminary checks
```

```
	if (nSlices()>1) {doStack = Dialog.getCheckbox;}
	noiseThr = Dialog.getNumber;
	showList = Dialog.getCheckbox;
	setBatchMode(true);
	firstSlice=getSliceNumber();
	lastSlice=getSliceNumber();
	if (doStack) {
		firstSlice=1;
		lastSlice=nSlices();
	}
	width=getWidth;
	height=getHeight;
	//maxRadius may be modified, should not be larger than 0.3*minOf(width, height);
	maxRadius=0.3*minOf(width, height);
	minFFTsize=1.3*maxOf(width, height);
	title=getTitle();
	size=4;
	while(size<minFFTsize) size*=2;
	//Main processing loop
	for (slice=firstSlice; slice<=lastSlice; slice++) {
		//Make autocorrelation of particle positions
		if (doStack) setSlice(slice);
		run("Find Maxima...", "noise="+noiseThr+" output=[Single Points] light
exclude");
		tempID=getImageID();
		tempTitle="temp-"+random();
		rename(tempTitle);
		run("Canvas Size...", "width="+ size+" height="+ size+" position=Center zero");
```

```
		run("FD Math...", "image1=["+tempTitle+"] operation=Correlate
image2=["+tempTitle+"] result=AutoCorrelation do");

		psID=getImageID();

		selectImage(tempID);

		close();

		//Make autocorrelation reference to correct finite image size effects

		newImage("frame", "8-bit White", width, height, 1);

		run("Set...", "value=255");

		tempID=getImageID();

		rename(tempTitle);

		run("Canvas Size...", "width="+ size+" height="+ size+" position=Center zero");

		run("FD Math...", "image1=["+tempTitle+"] operation=Correlate
image2=["+tempTitle+"] result=AutoCorrReference do");

		refID=getImageID();

		imageCalculator("Divide", psID,refID);

		selectImage(refID);

		close();

		selectImage(tempID);

		close();

		//Prepare normalized power spectrum for radial averaging

		selectImage(psID);

		makeRectangle(size/2, size/2, 1, 1);

		run("Set...", "value=0");

		run("Select None");

		circleSize=2*floor(maxRadius)+1;

		run("Specify...", "width="+circleSize+" height="+circleSize+" x="+(size/2+0.5)+"
y="+(size/2+0.5)+" oval centered");

		getRawStatistics(nPixels, mean);

		run("Select None");
```

```
                run("Divide...", "value="+mean);

                run("Specify...", "width="+circleSize+" height="+circleSize+" x="+(size/2+0.5)+"
y="+(size/2+0.5)+" oval centered");

                run("Radial Profile", "x="+(size/2+0.5)+" y="+(size/2+0.5)+"
radius="+floor(maxRadius)-1);

                rename("RDF of "+title);

                rdfID=getImageID();

                selectImage(psID);

                close();

                //Averaging of RDFs for stacks

                if (doStack) {

                        selectImage(rdfID);

                        Plot.getValues(x, y);

                        if (slice==firstSlice) ySum = newArray(y.length);

                        for (i=0; i<y.length; i++)

                        ySum[i]+ = y[i] / lastSlice;

                        close();

                }

        }//End Processing Loop


        //Create output plots with annotated titles and options

        if (doStack) {

                Plot.create("RDF of "+title+" (stack)", "Distance (pixels)", "RDF", x, ySum);

                if (showList) {

                        run("Clear Results");

                        for (i=0; i<x.length; i++) {

                                setResult("R", i, x[i]);

                                setResult("RDF", i, ySum[i]);
```

```
                }
                updateResults();
            }
        }
        else {
            selectImage(rdfID);
            Plot.getValues(x, y);
            Plot.create("RDF of "+title+" (slice"+lastSlice+")", "Distance (pixels)", "RDF", x,
y);
            if (showList) {
                run("Clear Results");
                for (i=0; i<x.length; i++) {
                    setResult("R", i, x[i]);
                    setResult("RDF", i, y[i]);
                }
                updateResults();
            }
            close();
        }//End Output
        setBatchMode("exit and display");// Comment this out if you get duplicate RDF outputs
} //End Macro
```

VITA

Eric Allan Drake

Candidate for the Degree of

Master of Science

Thesis: CHARACTERIZATION OF VISCOELASTIC MATERIALS THROUGH AN ACTIVE MIXER BY DIRECT-INK WRITING

Major Field: Materials Science and Engineering

Biographical:

Education:

Completed the requirements for the Master of Science in Materials Science and Engineering at Oklahoma State University, Tulsa, Oklahoma in May, 2017.

Completed the requirements for the Bachelor of Science in Electrical Engineering at Oklahoma State University, Stillwater, Oklahoma in May, 2014.

Experience:
National Aeronautics and Space Administration (NASA), Houston, TX, (Internship – Summer 2017)
Research Assistant, Oklahoma State University, Tulsa, OK (2015-2017)
Controls Engineer, Zeeco, Broken Arrow, OK (2014-2015)
Engineering Technician II, PCES, Tulsa, OK (Internship – Summer 2013)

Professional Memberships:
IEEE Member