Advanced Methods to Produce and Characterize Transport Properties of Novel Nanoporous Polymers

Andrei Nicolae

A dissertation
submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy

University of Washington
2018

Committee:

Vipin Kumar
John Weller
Devin MacKenzie
Nicholas Boechler
Brian Flinn

Program Authorized to Offer Degree:
Mechanical Engineering

University of Washington

Abstract

Advanced Methods to Produce and Characterize Transport Properties of Novel Nanoporous Polymers

Andrei Nicolae

Chair of the Supervisory Committee:
Vipin Kumar
Department of Mechanical Engineering

In this work, experimental techniques, equipment, apparatuses, models, algorithms, and software codes were developed to explore the nanoporous structure of polyetherimide (PEI) created using the solid-state foaming process. First reported in 2001, researchers have wondered about the transport properties of this skin-covered nanoporous structure but no one had successfully developed the means to reliably produce, access, or characterize it.

In this work, I developed techniques and equipment to produce flat, uniform, and dimensionally stable specimens with low variance. Then, I developed two unique methods to selectively and robustly remove the solid skin without damaging the underlying porous nanostructure. The resulting geometry brought about a fluid problem with peculiar boundary conditions, which has nontrivial solutions. Therefore, I solved the partial differential equations for these

geometries in order to compute transport properties from experimental data. The full solution to this problem made use of analytic and numerical methods, software algorithms, and image processing.

After developing the complete experimental and theoretical tool-sets, I obtained data on a wide range of nanoporous PEI structures, including permeability, diffusivity, tortuosity, and thermal conductivity. Permeability was found to be in the range of $10^{-16} - 10^{-15}\ m^2$, comparable to that of limestone, volcanic rock, and sandstone. Diffusivities were in the range 0.01 - 0.025, comparable to very dense soil. Tortuosity, a property often wondered about by researchers beholding scanning electron micrographs (SEM) of this nanostructure, was computed from diffusivity and porosity data to be in the range 20 - 40. These are the first results on transport properties, some 15 years after the discovery of the nanostructure.

During this doctoral work, several major discoveries were made. First, I found that PEI's nanostructure is impermeable to liquid water, however it allows gasses and wetting liquids (such as acetone and isopropyl alcohol) to freely flow through the structure. This phenomenon is similar to that exploited in the commercial polytetrafluoroethylene membrane *GoreTEX*. This waterproof-but-breathable property has many practical applications, especially given that PEI's nanostructure was shown to be waterproof at pressures of over 50 atmospheres, whereas *GoreTEX* can withstand less than 3 atmospheres. Additionally, I developed a new method to achieve a completely novel nanostructure in PEI by using a two-state foaming technique. These new nanostructures have a very unique appearance and resemble a hybrid of closed-cell and open-porous morphologies. Perhaps most notably, I also discovered a processing condition which produces PEI foams that are transparent. This is the first known occurrence of polymer foams with cells so small that light scattering is diminished to the point of optical clarity. The

cells are so small that our SEM techniques were inadequate in imaging the structure. Only a few voids on the order of 10 nanometers were visible. However, the structure was found to be open-porous as well as optically transparent, similar to Aerogel, but a polymer.

Lastly, I performed a feasibility study on 3D printing polymer bubbles. The idea was to create a nozzle that jets a stream of individual liquid polymer bubbles at high frequency, and deposit them in layers to create a 3D object. Although further work is necessary, I demonstrated the vision using liquid latex, butter, and paraffin wax as working materials, having successfully developed nozzles and techniques to create and release bubbles encapsulated by these substances.

The most significant aspect of my work isn't in the fact that I was first in 15 years to obtain transport property data on nanoporous PEI, but in creating the experimental and theoretical tools as a platform for much broader future research in this field.

Contents

List of Figures

List of Tables

Chapter 1

Nanoporous Polyetherimide

1.1 Background

The microcellular plastics laboratory at the University of Washington has performed research in the area of solid state foaming for almost 30 years. The first efforts in the early 1990s were to study the process in a variety of polymers such as polycarbonate (PC), polyvinyl chloride (PVC), polystyrene (PS), polyethylene terephthalate (PET), and others [1–3] . These were processed by saturating them with carbon dioxide at elevated pressure (usually 3-5 MPa) and room temperature, then foaming in hot liquid baths [4]. The structures produced were closed cell with cell sizes on the order of 10 to 250 microns [5, 6]. Since this initial research, the efforts were focused into other directions, one of which was to reduce the cell size. By the early 2000s, other researchers in the field had begun to experiment with more exotic thermoplastic polymers such as polyphenylsulfone (PPSU), polyimide (PI), and polyetherimide (PEI) [7–10]. It was observed that in these high temperature plastics, the cell sizes produced with the same saturating conditions were on the order of 1 to 20 microns, significantly smaller than previously obtained. Research showed that higher gas concentrations (more CO_2 dissolved in the polymer) yields higher cell nucleation densities. Efforts were made to increase the amount of dissolved gas in the polymer, most obviously by increasing the saturation pressure. By the early 2010s, there were a variety of nanocellular structures created in various polymers, including polymethyl-methacrylate (PMMA) [11,12]. More recently, the microcellular plastics laboratory has performed and published work that kept the saturation pressure at 5 MPa but reduced the saturating temperature down to -30°C. At lower temperatures, the solubility of gas in the polymer increases significantly, and so does the nucleation density. This results in decreased cell sizes, below 10 nanometers. During this pursuit of reducing cell size, an interesting observation was made in polyetherimide — the structure began to transition from closed-cell to open-cell. Several basic experiments showed that this structure allows fluids to pass through. However, nobody has done an extensive investigation on the properties of this open cell structure in regards to its fluid permeation characteristics, until now. This is the core of my doctoral work.

1.2 Polyetherimide and Open-cell Porous Structure

Porous materials with nanometer sized channels are of great interest in the fields of filtration, controlled release systems, microfluidics, supercapacitors, battery separators, scaffolds, templates, and others [13, 14]. A technique for creating a bicontinuous nanoporous polyetherimide (PEI) film was reported by Krause et. al [7, 15]. This technique is based on the solid state foaming process first described by Martini et. Al [16] which involves gas saturation of a polymer in the solid phase and expansion by heating to its glass transition temperature. The polymer type and saturation conditions (pressure and temperature) are critical to obtaining an interconnected structure. Figure 1.1 shows SEM micrographs of PEI samples that were saturated at different pressures. At lower saturation pressure, the cells are on the order of 10 μm and are isolated. At higher saturation pressure, the structure is open porous with cellular features on the nanometer scale.

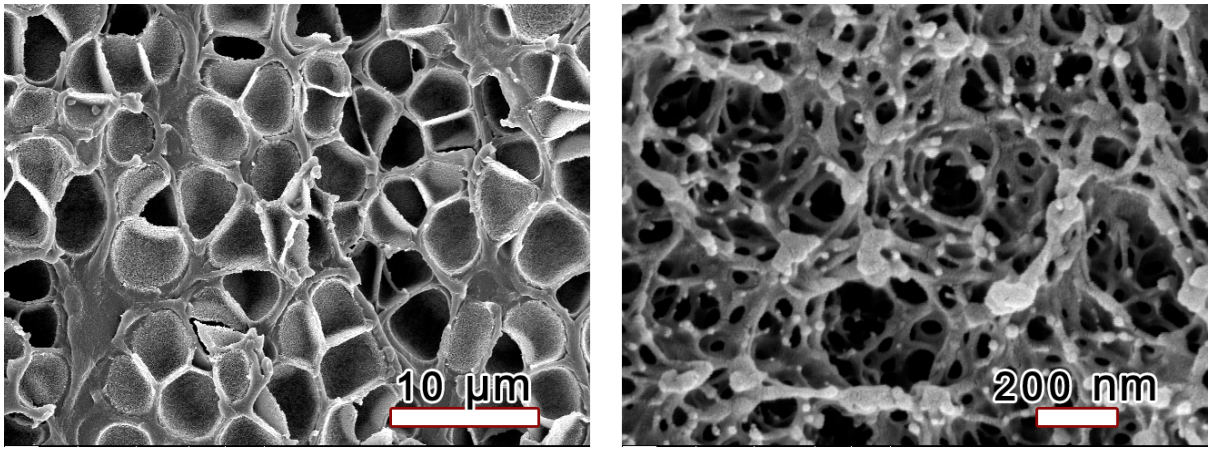

Figure 1.1: PEI foam structures made with different saturation pressures. Left: 1.0 MPa, Right: 5.0 MPa.

This solid-state foaming process, which is inherently characterized by molecular diffusion, yields an unavoidable solid skin on the surfaces, as shown in figure (1.2), which is not porous and therefore is not fluid permeable. To test fluid permeability, Krause et. al reported a technique in which samples were freeze fractured in liquid nitrogen and helium/nitrogen gas was flowed through the porous cross section by applying a pressure differential.

Later, Miller et. al. [10] described the processing conditions that produced PEI samples which appeared to be interconnected in scanning electron micrographs (SEM). However, flow tests were not performed. Building on Miller's work, Aher [17] attempted to remove the solid skin of PEI samples by end milling and sanding/polishing in order to expose large areas of the porous structure. The results were very inconsistent; however, Aher showed that an acetone/dye solution was absorbed into the core of the nanofoam by placing a drop on the face of a sample after the skins had been abraded by polishing.

During my work, I developed a new method for removing the skin without destroying the underlying pores. It consists of selectively removing the solid skin by machining either holes or channels such that the porous structure is exposed in these areas. The rest of the skin remains and serves as structural support, increasing rigidity and protecting the nanoporous surface when handling. Using this technique, the first fluid flow tests through a PEI nanofoam's thickness

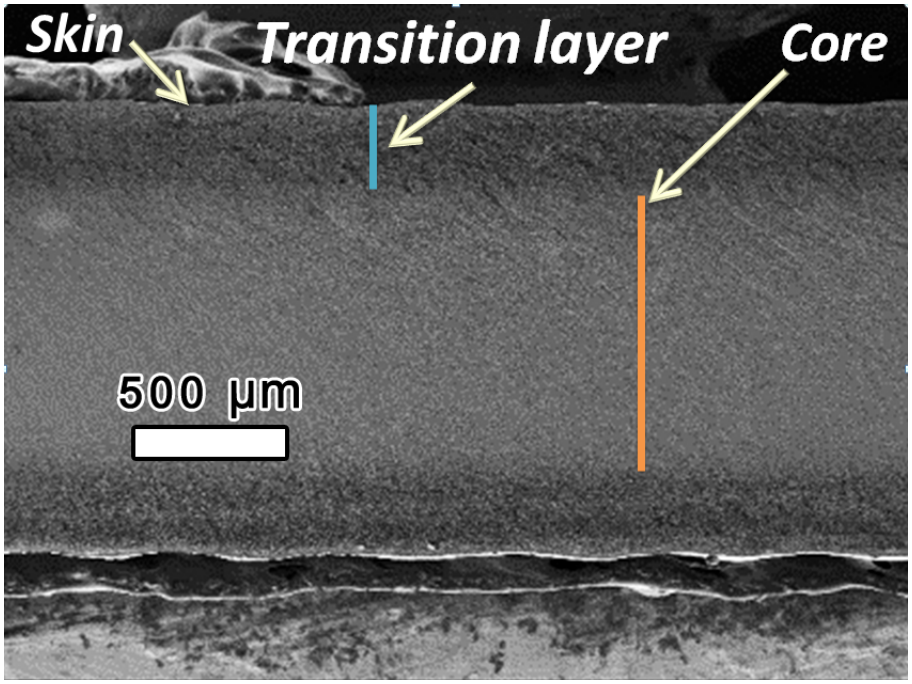

Figure 1.2: Cross-sectional micrograph of a nanoporous polyetherimde sample foamed using the solid state process. The porous structure is enclosed in a solid *skin* which is fluid impermeable.

were performed, again by applying a pressure differential and measuring the volumetric flow rate. To translate this experimental data on volume flow rates into a material property known as *permeability*, a mathematical fluid model was established and used, thus the PEI nanofoams could be quantitatively characterized.

1.3 Characterizing the Porous Structure

The motivation behind studying the properties of the open cell structure is to better understand what practical applications it could best serve. For filtration applications, it is required to know what the permeability is (to compute the pressure drop across the filter for a given flow rate). It is also desired to know what size particles it could filter and which would pass through. For slow-release and chemical applications, the diffusivity must be known - how the structure inhibits diffusion of gasses through the structure's open channels. The tortuosity of the structure is also a useful parameter to know the mean path length that a fluid particle must travel when traversing the material. These material properties will be defined in more detail in the following sections which set the mathematical background for experimental work. The formulas derived therein can translate raw experimental data into the aforementioned material properties.

1.3.1 Measurement of Fluid Permeability

The characteristic pore size d of a nanoporous PEI foam is of the order of 10^{-8} m, making the Reynolds number always small

$$Re = \frac{Ud}{v} << 1 \tag{1.1}$$

In this case, inertial forces become negligible and the fluid dynamics (creeping flow) for a Newtonian fluid can be modeled by the Stokes equations (ignoring body forces) along with the continuity condition [18, 19]

$$\mu\nabla^2\mathbf{u} - \nabla p + \mathbf{f} = 0 \tag{1.2}$$

$$\frac{\partial\rho}{\partial t} + \nabla\cdot(\rho u) = 0. \tag{1.3}$$

Solving the Stokes equations inside the pores/channels is impossible due to lack of information about the geometry. Therefore, a model is adopted which treats the pore structure as a property of the continuum. One such model is known as *Darcy*'s law which states that [20]

$$\mathbf{u} = -\frac{K}{\mu}(\nabla p - \rho\mathbf{g}) \tag{1.4}$$

where K is known as the "permeability" of the medium and μ the dynamic viscosity, assumed to be constants. This equation has been experimentally verified for a large number of porous media such as sand beds and is used extensively in geomechanics. A more general model allows for material anisotropy and arbitrary body forces f_i

$$u_i = -\frac{1}{\mu}K_{ij}(p_{,j} - \rho f_i) \tag{1.5}$$

Here, K_{ij} are the components of second order tensor $\mathbf{K}$ which is a function of pore geometry [21]. The continuity equation in this case includes the *porosity* Φ

$$\Phi\frac{\partial\rho}{\partial t} + \text{div}(\rho\mathbf{u}) = 0 \tag{1.6}$$

A general partial differential equation for the pressure can therefore be derived. For convenience, a Cartesian coordinate system with orthonormal basis $\{\mathbf{e_1}, \mathbf{e_2}, \mathbf{e_3}\}$ for Euclidian 3-space is employed:

$$\mathbf{u} = -\frac{1}{\mu}\mathbf{K}(\mathbf{grad}\, p - \rho\mathbf{f}) \tag{1.7}$$

$$\mathbf{K} = K_{pq}\mathbf{e_p}\otimes\mathbf{e_q} \tag{1.8}$$

$$\mathbf{grad}\, p = \frac{\partial p}{\partial x_r}\mathbf{e_r} \tag{1.9}$$

It follows

$$\text{div}(\mathbf{K}\,\mathbf{grad}\, p) = \text{div}(K_{pq}\mathbf{e_p}\otimes\mathbf{e_q}\frac{\partial p}{\partial x_r}\mathbf{e_r}) = \text{div}(K_{pq}\delta_{rq}\frac{\partial p}{\partial x_r}\mathbf{e_p}) = \frac{\partial}{\partial x_p}(K_{pr}\frac{\partial p}{\partial x_r}) \tag{1.10}$$

Similarly,

$$\text{div}(\mathbf{K}\,\mathbf{f}) = \frac{\partial}{\partial x_i}(K_{ij} f_j) \tag{1.11}$$

Using the well-known identity

$$\text{div}(\rho\mathbf{u}) = \mathbf{u} \cdot \mathbf{grad}\,\rho + \rho\,\text{div}\,\mathbf{u} \tag{1.12}$$

the continuity equation can be written as

$$\Phi\frac{\partial \rho}{\partial t} + u_s\frac{\partial \rho}{\partial x_s} - \frac{\rho}{\mu}\left(\frac{\partial}{\partial x_p}\left(K_{pr}\frac{\partial p}{\partial x}\right) - \frac{\partial}{\partial x_i}(\rho K_{ij} f_j)\right) = 0 \tag{1.13}$$

Using the identities

$$\text{div}(\mathbf{Tu}) = \text{trace}(\mathbf{T}\mathbf{grad}\,u) + \mathbf{u} \cdot \text{div}(\mathbf{T}) \tag{1.14}$$

$$\mathbf{grad}(\phi u) = \phi\,\mathbf{grad}\,u + u \otimes \mathbf{grad}\,\phi \tag{1.15}$$

the continuity can be written as

$$\Phi\frac{\partial \rho}{\partial t} + u\cdot\mathbf{grad}\,\rho - \frac{\rho}{\mu}(tr\{\mathbf{K}(\mathbf{grad}\,(\mathbf{grad}\,p) - p\,\mathbf{grad}\,\mathbf{f} + f \otimes \mathbf{grad}\,\rho)\} + (\mathbf{grad}\,p + \rho\mathbf{f})\cdot\text{div}\,\mathbf{K} = 0 \tag{1.16}$$

This equation has three unknowns: ρ, $\mathbf{K}$, p. To solve it, a constitutive relation must be used to express $\rho = \hat{\rho}(p)$, and $\mathbf{K}$ must be determined, if possible, by experiments. For further insight into the permeability tensor, suppose $\mathbf{K} \neq 0$, $\mathbf{grad}\ p \neq \mathbf{0}$; then the second law of thermodynamics requires

$$-\mathbf{u} \cdot \mathbf{grad}\,p = \mathbf{K}\,\mathbf{grad}\,p \cdot \mathbf{grad}\,p > 0 \tag{1.17}$$

Since $\mathbf{grad}$ p is arbitrary, $\mathbf{K}$ must be positive-definite and therefore have positive trace, $\text{tr}(\mathbf{K}) > 0$. Even if the flow is assumed irrotational, the above are not sufficient conditions to assert $\mathbf{K}$ is symmetric, contrary to a claim published in a paper by Liakopoulos [22]. For example, let $p = ax + by + cz$, $a, b, c \in \mathbb{R}$, and let $\mathbf{K}$ (in basis $\{\mathbf{e_1}, \mathbf{e_2}, \mathbf{e_3}\}$) be

$$\mathbf{K} = \begin{pmatrix} 1 & 1 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 1 \end{pmatrix} \tag{1.18}$$

Clearly, here $\mathbf{K}$ is positive definite and $\mathbf{grad}\,p =$ constant, therefore $\mathbf{K}\,\mathbf{grad}\,p$ has zero curl and divergence without $\mathbf{K}$ being symmetric. As it turns out, the permeability tensor indeed must be symmetric to satisfy the Stokes equations, however the proof is much more involved [21].

However, when looking at scanning electron micrographs of the foam's nanostructure it is apparent that the geometry is uniform in the planar direction and also invariant under rotations about the specimen' s normal direction $\hat{\mathbf{z}}$. This is expected since the foaming process is stochastic and the raw material isotropic. Due to this radial symmetry, $\hat{\mathbf{z}}$ is an eigenvector of $\mathbf{K}$ with eigenvalue k_1. Also, every vector parallel to $\hat{z}$ is an eigenvector of $\mathbf{K}$ with eigenvalue k_2.

These eigenvalues are functions of coordinate z and are approximately constant except in the foam's transition region.

If a pressure gradient is applied only in the direction of $\mathbf{z}$ (by applying different pressures at the specimen's faces) the flow equation becomes

$$|\mathbf{u}| = \frac{k_1(z)}{\mu}|\mathbf{grad}\, p| \tag{1.19}$$

In one dimension, $|\mathbf{u}|$ is a constant and another constant k can be defined such that

$$|\mathbf{u}| = \frac{k}{\mu}\Delta p \tag{1.20}$$

Practically, k can be thought of as the effective permeability of the material, which would equal k_1 if $\partial k_1/\partial z = 0\ \forall z$.

A method of measuring permeability is given by ASTM D6539-13 and involves a simple apparatus which creates a fixed pressure boundary condition on a sample while measuring the volume flow rate of a gas through the sample. In this setup, the effective permeability k for a rectangular sample of dimensions $a \times b \times c$ (thickness c) can be computed by

$$k = \frac{\mu Q c}{ab\Delta p} \tag{1.21}$$

1.3.2 Measurement of Tortuosity

Envisage a solid material of rectangular prism shape, i.e. the set
$S = \{(x, y, z)|0 < x < a, 0 < y < b, 0 < z < c\}$ that is then drilled straight through the thickness c with hole diameter d and total number of holes n, forming a subset $M \subset S$. The average density of this material *M* is

$$\rho = \frac{m}{V} = \frac{m - \rho_0 n\pi d^2 c/4}{V} = \rho_0\left(1 - \frac{n\pi d^2 c}{4abc}\right) = \rho_0\left(1 - \frac{A}{A_0}\right) \tag{1.22}$$

where ρ_0 is the solid material density and A, A_0 are interpreted as follows: suppose that a section cut is made by a plane $D = \{(x, y, z)|z \in [0, c], x, y \in R\}$. Then $A_0 = ab$ is simply the total area enclosed by the perimeter of the cross section, and A is the total area of the hole cross-sections, that is, the area of the set $\underline{A} = D \cap (S\backslash M)$. Solving for A

$$A = \frac{A_0(\rho_0 - \rho)}{\rho_0} = A_0\Phi \tag{1.23}$$

where $\Phi \in (0, 1]$ again is the porosity of the material.

If a fluid is to flow through the thickness of this material, assuming $d/c \ll 1$, the distance that each fluid particle must travel through the material is very close to c. Now suppose that instead of a material with straight holes there are curved channels through the thickness but the porosity Φ remains the same. Consequently, a passing fluid particle must travel through the material, on average, a distance $l = \tau c$ where $\tau \geq 1$ is defined as the *tortuosity* of the material, a factor independent of Φ.

In this case, A will not be a constant but generally a piecewise continuous function $A(z)$. However,

$$\frac{1}{abc}\int_0^c A(z)dz = \Phi = \frac{\overline{A}}{ab} \tag{1.24}$$

The question now arises how τ can be determined considering that tracking fluid particles through a medium is experimentally difficult. One method is to fill the voids with a fluid of resistivity χ and measure the resistance between top and bottom faces of the material. From Pouillet's law, a lower bound on resistance is easily deduced:

$$R \geq \int_0^c \chi dz/A(z) \tag{1.25}$$

The resistance is a minimum if the pore structure is that of ***M***. In general, the pore structure isn't such; let $\tilde{M} \subset S$ denote this general porous solid. R can be greater due to tortuosity, or from geometric reasons. However, satisfying certain conditions suggests that R isn't too far from the minimum due to the later. Physically, it means that the paths through $\tilde{M}$ have no "dead ends" or isolated pores, and have a rather well defined direction. Formally, this is stated in the following way: let $s = \{(x,y,z)|a_1 < x < a_2, b_1 < y < b_2, 0 < z < c, (x,y,z) \subset S\}$. Let $\underline{A_j} = D \cap (s \backslash \tilde{M})$ and A_j the area of $\underline{A_j}$. Let $\eta \in Z$ denote the number of closed connected regions in $\underline{A_j}$. In general, $\eta(z)$ is not constant and therefore discontinuous; let z_j denote an arbitrarily small finite neighborhood of points around each z where η is discontinuous. R is a minimum for a given tortuosity only if

$$A_j \cap \partial s \neq \varnothing \, \forall z_j \tag{1.26}$$

$$A_j \in C^0([0,c], R) \tag{1.27}$$

which are necessary but not sufficient conditions. A pore structure could be constructed to satisfy these conditions without yielding a minimum R for a given tortuosity, although it wouldn't differ greatly unless cleverly designed. If the pore structure is produced by a stochastic process (such as by a blowing agent) there is a very low probability of this special case occurring. Regardless, it is difficult to measure whether a real foam satisfies these conditions at all. For this reason, an effective tortuosity can be defined in the following way: suppose that

$$\frac{1}{A}\frac{dA}{dz} \ll 1 \therefore A \approx \bar{A} \tag{1.28}$$

This condition is readily seen in nanoporous PEI micrographs where the core is "uniform" and pores are very small. It follows that

$$R_{min} \approx \frac{\chi c}{\bar{A}} = \frac{\chi c}{ab\Phi} \tag{1.29}$$

Then, the effective tortuosity τ_e is defined by

$$\tau_e \equiv \frac{R_{measured}}{R_{min}} = \frac{ab\phi R_{measured}}{\chi c} \tag{1.30}$$

Note that for a pore structure such as M, the tortuosity and effective tortuosity are identical. Experimentally, it becomes a matter of thoroughly intruding the pores with an electrolyte with known resistivity and measuring the resistance between two faces of a porous rectangular prism.

1.3.3 Determining a Characteristic Pore Diameter

Scanning electron microscope (SEM) images of nanofoam cross sections reveal odd shaped voids that appear to intrude into the depth and connect with other open regions. Although some are round and, none appear to have a well-defined length as a capillary. Therefore, the measured permeability is compared to that of a capillary type model of characteristic capillary diameter δ and tortuosity τ. One such model is shown by Scheidegger (1974):

$$k = \frac{1}{96}\frac{\phi\delta^2}{\tau^2} \tag{1.31}$$

Since this assumes a capillary pore structure, the reported values of δ could be in terms of this model and its assumptions, and could be compared against SEM images to observe any correlation between visual cell size and δ. In this context, δ is defined as

$$\delta \equiv \sqrt{\frac{96k{\tau_e}^2}{\phi}} \tag{1.32}$$

This information does not set an upper bound on the pore diameter, a useful number for filtration applications. However, a clever experiment can be performed to establish it. First, a collection of uniform density spherical nanoparticles is obtained whose diameter δ_p distribution is P. This is a probability density function, i.e.

$$\int_0^\infty P(\delta_p)d\delta_p = 1 \tag{1.33}$$

This function can be approximately determined by performing image analysis on SEM images of these nanoparticles. Then, a low concentration gas/nanoparticle suspension is passed through the nanofoam. The particles with $\delta_p < \delta_{max}$ can pass, while those with $\delta_p > \delta_{max}$ cannot. One method to determine δ_{max} is to measure the mass fraction m_f of the nanoparticles that were blocked by nanofoam. Then δ_{max} is such that

$$\frac{\int_{\delta_{max}}^\infty \delta_p^3 P(\delta_p)d\delta_p}{\int_0^\infty \delta_p^3 P(\delta_p)d\delta_p} = m_f \tag{1.34}$$

Where m_f can be found by carefully weighing the nanofoam sample before and after the experiment to determine the mass of nanoparticles entrenched therein. Another method, perhaps more accurate, would be to collect the nanoparticles that pass through the nanofoam on a sticky substrate and perform SEM image analysis to find the maximum particle that was able to pass through. To prevent clogging of the nanopores by the larger nanoparticles, those with smallest diameter should be passed through first. Since the cross-sectional area to mass ratio is proportional to $1/\delta_p$, the Venturi effect with progressively higher flow rates can be used to selectively transport the nanoparticles in order by size.

1.3.4 Effect of reversing the pressures

The question arises as to what happens to the flow rate if the sample is flipped in the permeability testing apparatus. Mathematically, the boundary conditions get reversed. Let ∂S denote the boundary of a specimen. Let T and B denote the sets corresponding to the top and bottom

surfaces (boundaries) of the test specimen, respectively. Let T_1 and B_1 denote the sets of areas where the skin is removed from the top and bottom surfaces, respectively.

$$T_1 \subset T, B_1 \subset B \tag{1.35}$$

The boundary conditions are $p = p_T$ everywhere on T_1 and $p = p_B$ everywhere on B_1.

$$p = p_T \forall x \in T_1 \tag{1.36}$$

$$p = p_B \forall x \in B_1 \tag{1.37}$$

$$p_T, p_B \in \mathbb{R} \tag{1.38}$$

A no-flux condition ($\nabla p \cdot \mathbf{n} = 0$) is applied everywhere else, i.e. on $\partial S - (T_1 \cup B_1)$.

$$\nabla p(x) \cdot \mathbf{n} = 0 \forall x \in \partial S - (T_1 \cup B_1) \tag{1.39}$$

Let p be a solution to $\nabla^2 p = 0$ that satisfies the boundary conditions above. The function

$$f = p_B + p_T - p \tag{1.40}$$

is also a solution to $\nabla^2 f = 0$, however the boundary values of this new function on T_1 and B_1 are reversed, i.e. $f = p_T$ on B_1 and $f = p_B$ on T_1. The flow rate in this case is $\nabla f = -\nabla p$, therefore the magnitude is unchanged and the direction reversed. However, since the specimen itself is flipped, there is no difference in flow in the lab frame. This important result simplifies experiments by making orientation indifferent. It is also important to note that the skin removed areas need not be the same – no assumption was made in the equality of areas of sets T_1 and B_1.

Chapter 2

Experimental Techniques

2.1 Experimental

The work herein involved a significant amount of experimentation. I designed and built most of the equipment needed to perform these experiments. Commercial machines were costly - for example, a permeability measuring apparatus was quoted at $45,000; I built my own for less than $250. Another reason for making custom equipment is that I could optimize the design to work with the dimensions of my PEI nanofoam samples, streamlining the experimental process.

2.2 Blistering and Curling

There are many subtleties when working with the solid-state foaming process, one of which is the difficulty in producing flat, uniform samples. Most previous students in our lab have used a silicon oil bath as the temperature reservoir for foaming; however, this method produces curved samples. An analogy was drawn to potato chips which are also cooked in hot oil and result in a curled, wavy shape. For SEM analysis, this is not troublesome since only a small portion of the sample is needed for imaging. Since I had to perform fluid flow experiments, I needed flat sample to mount and seal in my test apparatus.

The second problem was the blistering that occurs on most samples during foaming. As the polymer expands, pockets of gas form large blisters which often pop and destroy the sample. These blistered samples are unsuitable for flow testing. Miller reported in his thesis that about 3/4 of the PEI samples suffered from severe blistering [9]. A yield of 25% was unacceptable, so I had to seek a better solution.

To solve the curling problem, a hot press was experimented with by previous students. The department has a hydraulic hot press designed for curing composites under pressure. Due to this function, the minimum clamping force of its parallel steel plates is 0.1 tons. This solved the curling problem but the high clamping force collapsed the pore structure at higher foaming temperatures, inhibited cell growth, and produced inconsistent densities. For this reason, I have built my own parallel-plate foaming apparatus and is described in a later section.

The blistering problem persisted in all foaming methods. However, following the suggestion of then Ph.D student Huimin Guo in the lab, I switched from using 1.0 mm thick PEI sheets to 0.5 mm thick PEI sheets for my samples. These thinner sheets simply did not have the blistering problem.

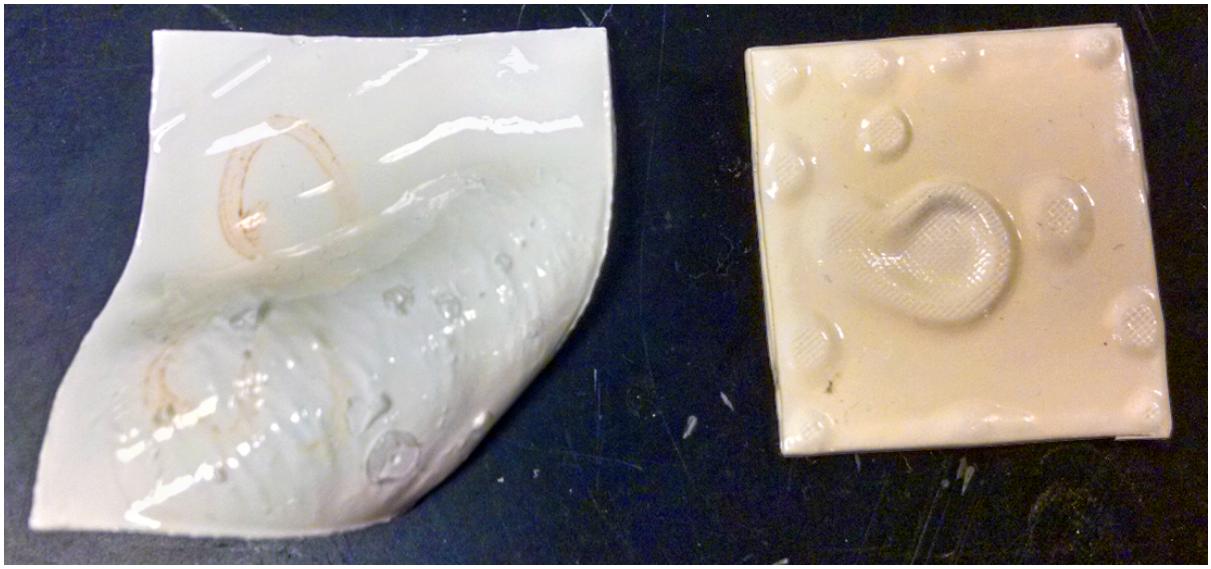

Figure 2.1: Left: PEI sample curled due to foaming in an oil bath. Right: Blisters produced during foaming.

2.3 Stopping Desorption

In using 0.5 mm thick sheets, the desorption time became increasingly critical. With 1.0 mm sheets, it was acceptable to have desorption times as long as 20 minutes without significant skin growth. However, for 0.5 mm sheets, this desorption time was reduced by a factor of 4 down to 5 minutes. While this is enough time to vent the pressure vessel, remove the sample, and place it in the foaming apparatus, I could only foam 3 or 4 samples in this 5 minute time frame. Since I needed a large number of samples in order to reduce statistical variation of the experimental results, I needed a method to stop gas desorption for samples in the queue while waiting to be foamed.

Knowing that diffusion rates increase with temperature, I tried placing the samples in liquid nitrogen immediately after removal from the pressure vessel. This idea was highly successful; the results in table (2.1) show that even 3 hours of desorption under liquid nitrogen had no significant effect on the density of the produced foam.

Table 2.1: Data showing the effectiveness of liquid nitrogen in stopping desorption.

Desorption Time	Liquid Nitrogen	Foaming Temperature	Relative Density
2 min	No	200 °C	0.462
3 min	No	200 °C	0.549
2 hr 38 min	Yes	200 °C	0.547
3 hr 6 min	Yes	200 °C	0.541

This fact alone allowed me to continue researching PEI nanofoams. Had I been unsuccessful in stopping desorption, I would have abandoned this project due to the excessively time

consuming process of producing samples. However, I was able to make as many as 50 samples in one batch using this liquid nitrogen technique.

2.4 Parallel Plate Heater

The constructed foaming apparatus is best explained in the figures. It consists of two parallel aluminum plates constrained by linear bearings to move freely only in the normal direction. Behind each plate is a thin 3" x 3" electric heating pad (nichrome between two kapton sheets) 10 W/in^2 at 110 V.

The clamping force is controlled by placing a mass atop the platform - the required force to produce flat PEI samples is much less than the 0.1 tons that the hydraulic press would exert. Through experimentation, I found that for 0.5 mm PEI disk samples of 1 inch diameter, the optimal weight was 2 kg. Placing 4 kg increased friction which inhibited the polymer's in-plane growth, while 1 kg was insufficient in smoothing out the curl.

Controlling the foaming apparatus was done via serial communication (either USB or Bluetooth). Any terminal software could be used. On startup, the microcontroller prompts the user to enter the set temperature for each plate (they need not be equal). The controller then heats the plates to these temperatures using a feedback system with measurement by two embedded thermocouples (touching each aluminum plate), and displays the temperatures on the terminal window every two seconds for the user to view. Below is a truncated sample serial output of a test run:

```
Connected
Enter Temperature 1.
100
Got it. Now enter Temperature 2.
110
Got it. Control started.

Temp 1: 26
Temp 2: 26

Temp 1: 28
Temp 2: 26

Temp 1: 29
Temp 2: 27

Temp 1: 31
Temp 2: 28

Temp 1: 32
Temp 2: 28

Temp 1: 34
Temp 2: 29
```

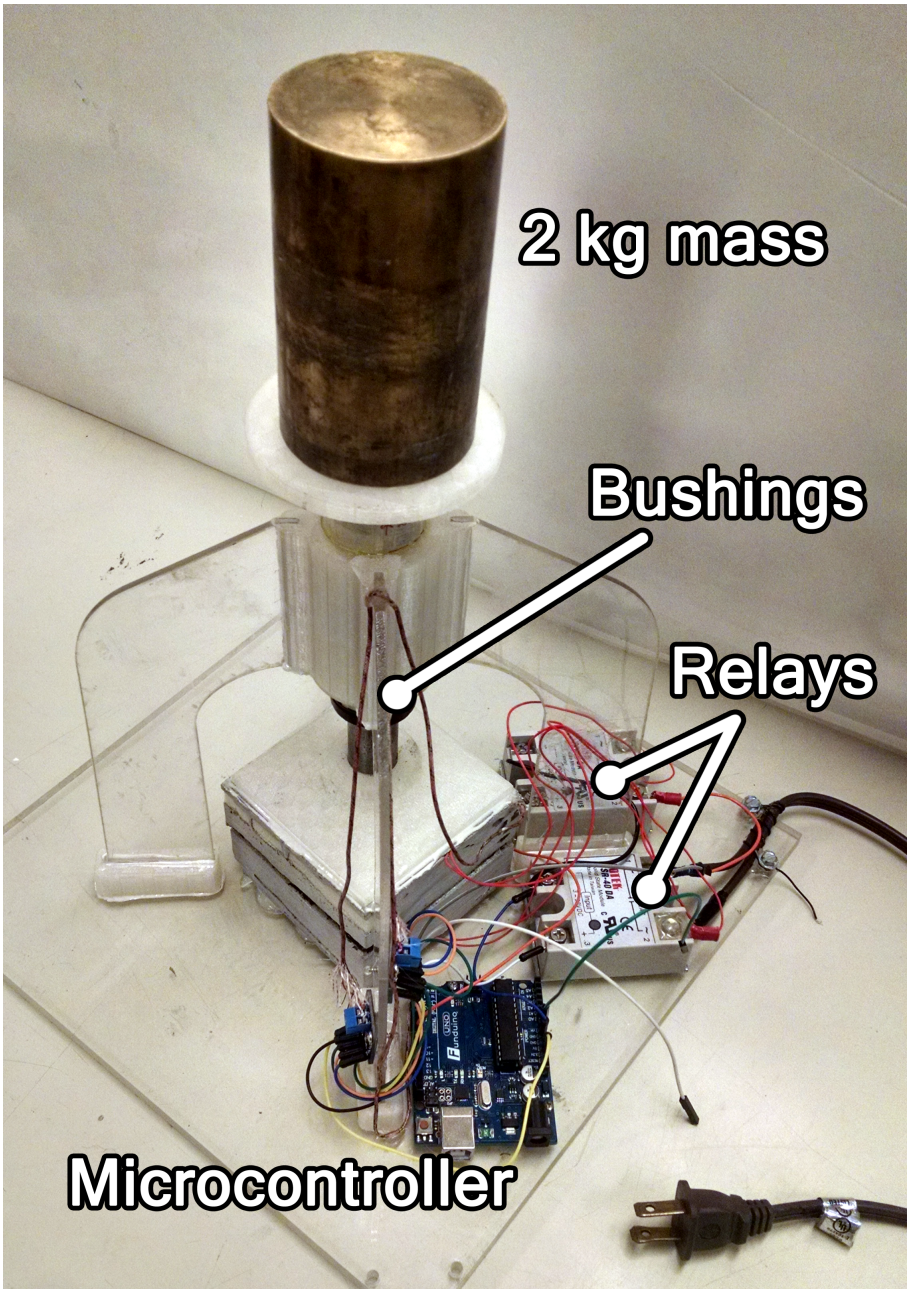

Figure 2.2: Diagram of the foaming apparatus I built and used to produce all my samples for fluid flow analysis.

Even with this apparatus, there was difficulty in getting uniform samples due to the uneven heating. Firstly, the PEI sheets were rolled up and placed in tubes for shipping, thereby acquiring a slight curvature. Attempts to straighten the sheet were only moderately successful. Secondly, the samples swell slightly when saturated with CO_2 gas, which accentuates imperfections. Although the foaming apparatus' aluminum hot plates were flat and parallel (to within some tolerance), the samples weren't; therefore, as the plates closed they would not contact the entire sample simultaneously. The first region to contact the hot plate will foam first, expanding, and creating a gap that further removes the remaining unfoamed region from the heat source.

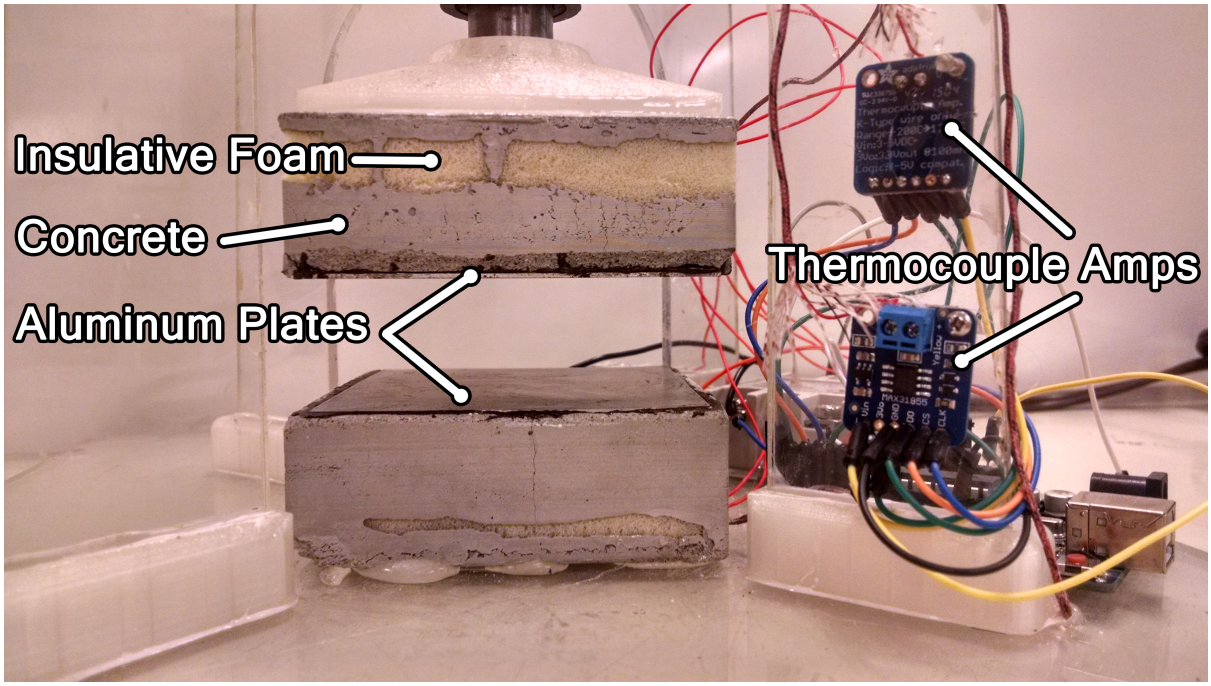

Figure 2.3: Close-up of the clamping plates in the open position.

This unstable system leads to non-uniform density of the sample. To remedy this, I tried many techniques to diffuse the heat evenly across the surface of the sample. One attempt was to use thermally conductive paste to eliminate air gaps and reduce contact resistance; this was ineffective and messy. Another idea was to place PTFE sheets between the sample and aluminum plates, an idea borrowed from Huimin Guo. This was also ineffective. However, placing regular bathroom paper towel was successful. Still, it was found that the optimal number of paper towel sheets was 2 per side. Placing more or less produced uneven samples. However, placing 2 sheets of paper towel reduced the initial heat transfer rate, allowing the sample to foam uniformly. With this discovery, the foaming technique was complete.

2.5 Skin Removal Techniques

Once the foaming method was established, I could reliably produce flat/uniform PEI samples starting with 0.5 mm thick sheets. However, the next step was to determine how to remove the skin. I have done a lot of work experimenting with machining techniques, both end milling and drilling. The results, while interesting, will not be given here as they were presented at the SPE FOAMS 2016 conference and are available as a paper in the conference proceedings [23].

The takeaway from the machining study is that during the cutting process, the fragile nanocellular structure gets collapsed under shearing forces from the cutting tool. This causes a "smudged" appearance which closes the structure and inhibits fluid flow, making the procedure counterproductive since the goal is to remove the skin so that fluid can freely flow. Figure (2.4) shows this smudging effect for three different skin removal methods.

Machining the skin is very involved - since the skin is roughly 150 μm the sample must be highly parallel to the CNC mill's x-y plane. This becomes a problem if the surface isn't planar

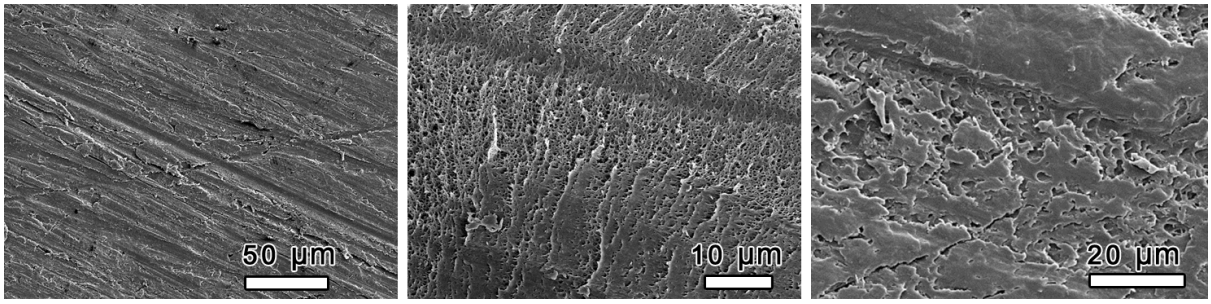

Figure 2.4: The cellular structure gets ”smudged” during the skin removal process. Left: Sanding with 1000 grit paper. Middle: Cutting with a steel razor blade. Right: End milling.

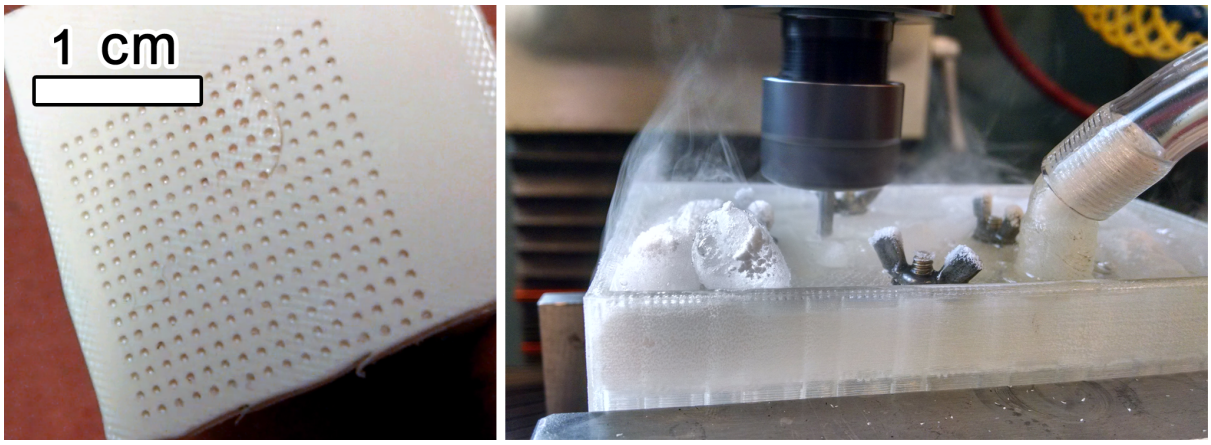

Figure 2.5: Left: PEI sample with skin removed by drilling a square pattern of holes. Right: End milling in a dry ice / alcohol bath to reduce temperature.

to within a few microns tolerance. Additionally, the skin must be removed on both sides, so flipping the sample and realigning the local and machine's coordinate systems is necessary. When drilling holes (which leaves skin for structural integrity) the tool size must be kept small so that the tool tip radius effects are minimized. This requires the drilling of a large number of holes, on the order of 500 per side or 1000 per sample. Therefore, machining the skin in this fashion is very time consuming, not to mention mill and tool setup times.

For the above reasons I abandoned the machining method and developed a new way of removing the skin. Building on what I learned from the machining experiments, I decided to not “remove” the skin but rather “move it out of the way”. This is done by simply piercing the skin with a sharp object such as a needle or razor blade. As the material pushed apart by the sharp object, the cells tear and expose the porous structure. Although there will be some smudging and cell collapse, it does not prevent fluid from flowing through these holes.

The challenge is piercing a large number of holes to a controlled depth on both sides of a PEI sample. Obviously, piercing each hole individually is impractical. Therefore, I needed an array of needles that could be used to simultaneously pierce a large number of holes. Since I could not find such an array for purchasing, I built one myself. Its construction is the most tedious task that I have done at the university. Dressmaker pins were purchased at a local crafts store; these have a flat head (like a nail) and about 1/3 of them were not straight. In order to have a close-packed array of holes, the heads were cut off with hand pliers and every pin was

inspected for curvature and discarded accordingly. This was done for over 700 needles. Once the needles were hand selected and cut, they were placed into a 3D printed part which is a block with a half inch hole in the center. They were carefully placed in a vertical fashion and each pin tip was pressed against a flat piece of glass to ensure that every tip was coincident on the same plane. Superglue was used to temporarily hold the needles together while casting epoxy was injected (using a syringe/needle) between the needles. To ensure that there were no air gaps between the needles (which could cause some needles to slide during use), the part was placed into a vacuum chamber periodically as the epoxy was being injected. It could not be done in one step since the air being removed would cause the uncured epoxy to be pushed out and overflow. Once cured, an extra support structure was made on the needle ends and filled with epoxy to further ensure that all needles were fixed in the holder. Two of these parts were produced so that PEI foam sample could be pierced from both sides simultaneously. Figure (2.6) shows the final product.

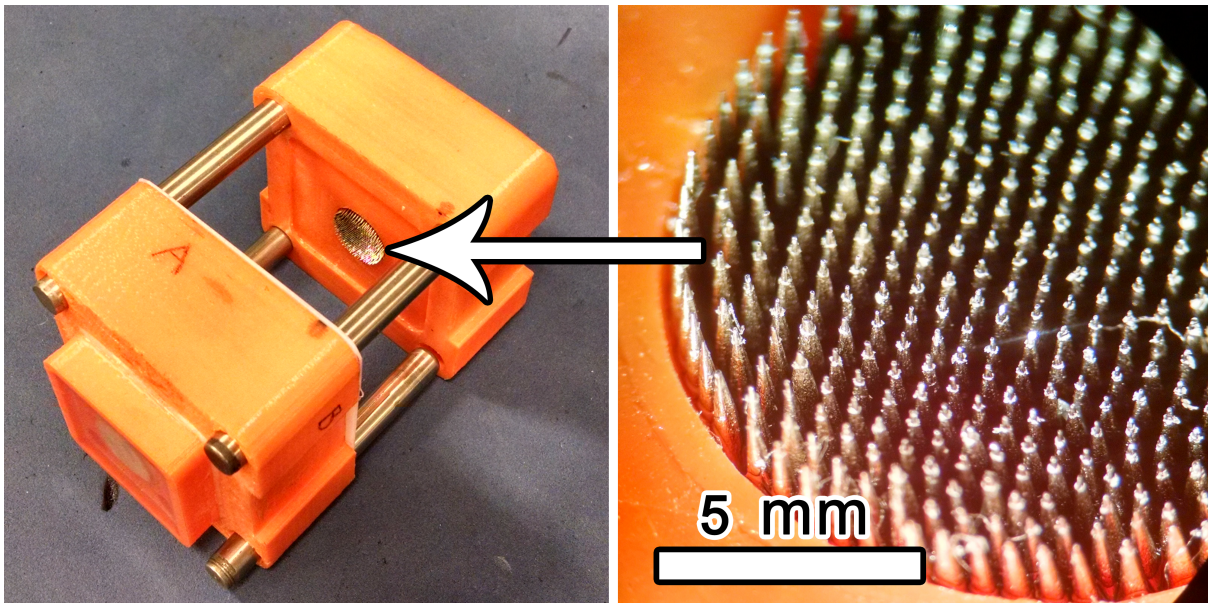

Figure 2.6: Left: Skin piercing apparatus consisting of 2 opposing arrays of needles hexagonally packed into a 0.5" circle. Right: Microscope close-up image of the needle tips.

To use the device, a PEI sample is placed atop the needles and a hand press used to exert the necessary clamping force for piercing the holes. Using one needle and a digital scale, I found that piercing the skin of my PEI foam samples (produced as above) requires about 2 N of force (scale registered 200 g). Using image processing techniques (more on that later) I found that there were about 370 needles in the array, therefore the total force required is roughly 740 N, or 166 lbf. For this task I employed an arbor press in the UW ME machine shop.

Since the arbor press is purely mechanical and has no force readout, I had to design my own way to make consistent samples that were pierced the proper amount. Too little clamping force would not pierce the skin adequately, which would invalidate results. Too much clamping force elastically deformed the needle holders and caused through-holes in the samples. In order to make sure that the samples's skin was pierced but 0.5 mm of nanoporous core remained intact, I placed a 0.5 mm spacer sheet in between the needle holders as shown in figure (2.7). This stopped the two halves to within 0.5 mm in between them. However, applying too much force

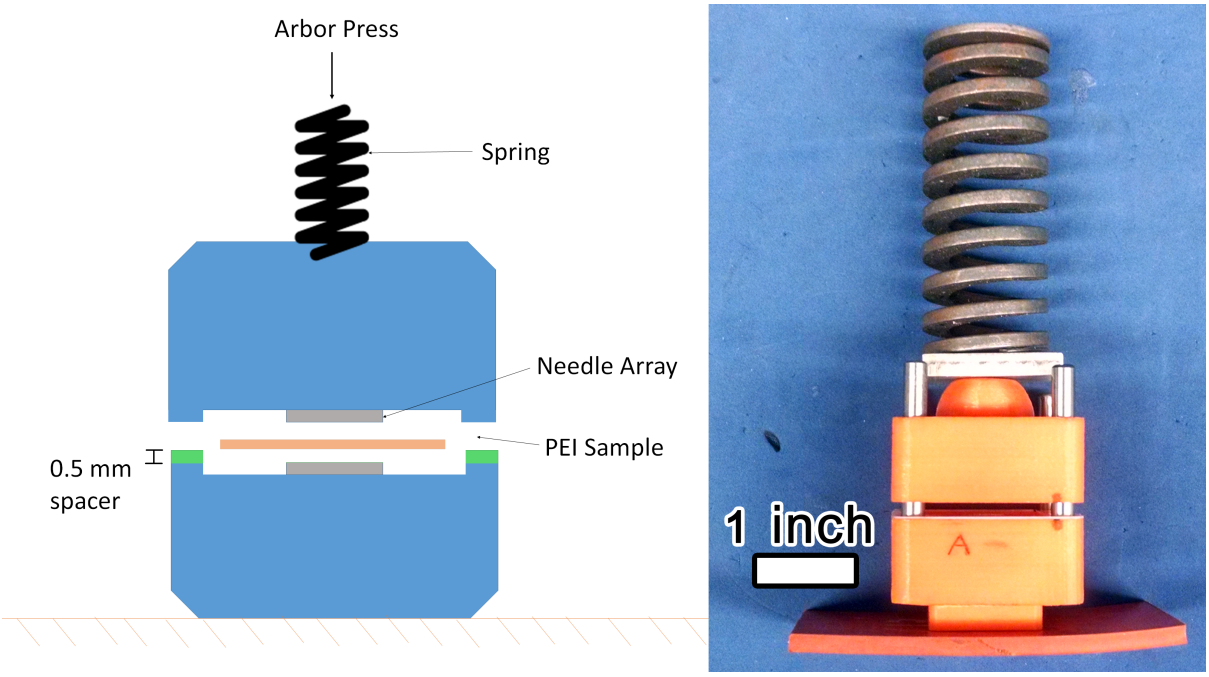

Figure 2.7: Diagram and photograph of the needle piercing apparatus and spring system.

via the arbor press could easily deform the needle holders and pierce further.

To be consistent in the applied force, I built a spring system that makes the applied force insensitive to the arbor press' lever position. Since for a spring $F = -kx$ it follows that $\frac{dF}{dx} = -k$ which is an intuitive result - decreasing the stiffness results in less sensitivity. Without the spring, the stiffness of the system was a function of the elasticity of the holder and the elastic modulus of the rubber pad placed underneath it. This is much higher than the stiffness of the spring. To find the optimum compression of the spring (to result in the optimum clamping force of the needles onto the samples) several samples were pierced with different spring compression displacements and fluid flow analysis was performed on each. Table (2.2) shows the results.

Table 2.2: Data showing the gas flow rate for samples pierced with different spring compressions (clamping forces). These particular samples were PEI foamed at 190 C.

Compressed spring length	Gas ΔP	Time to flow 4 mL.
7.3 cm	100 psi	34.34 s
6.8 cm	100 psi	6.68 s
6.3 cm	100 psi	5.49 s
5.8 cm	100 psi	6.46 s

The data shows that once the skin is pierced, larger forces are unnecessary - this occurs when the spring is compressed to about 6.8 cm. The lever position at this compression was marked, and every sample was therefore produced with the same clamping force.

This apparatus again saved a tremendous amount of time and produced quality, consistent samples. Machining the PEI on an end mill would require about 20 minutes of cutting time,

excluding the tool and mill setup time. With this apparatus, the setup time is less than one minute and the time to pierce one sample is about 30 seconds.

2.6 Fluid Flow Apparatus

To push fluids through the nanofoam, a test apparatus was designed based on ASTM D6539-13. Its function is to apply a controllable constant pressure on one side of a nanofoam sample, atmospheric pressure on the other, and measure the volume flow rate of gas passing through the sample at this applied pressure difference.

This was done by sealing the sample using rubber O-rings into aluminum blocks which had been drilled to allow the flow of gas to/from the faces of the sample. A 3/4" steel porous disk provides support for the sample to prevent warping for pressure but still allow the gas to flow and be collected into the measurement device (a pipette labeled in 0.1 mL increments). Figure (2.8) shows the diagram and device.

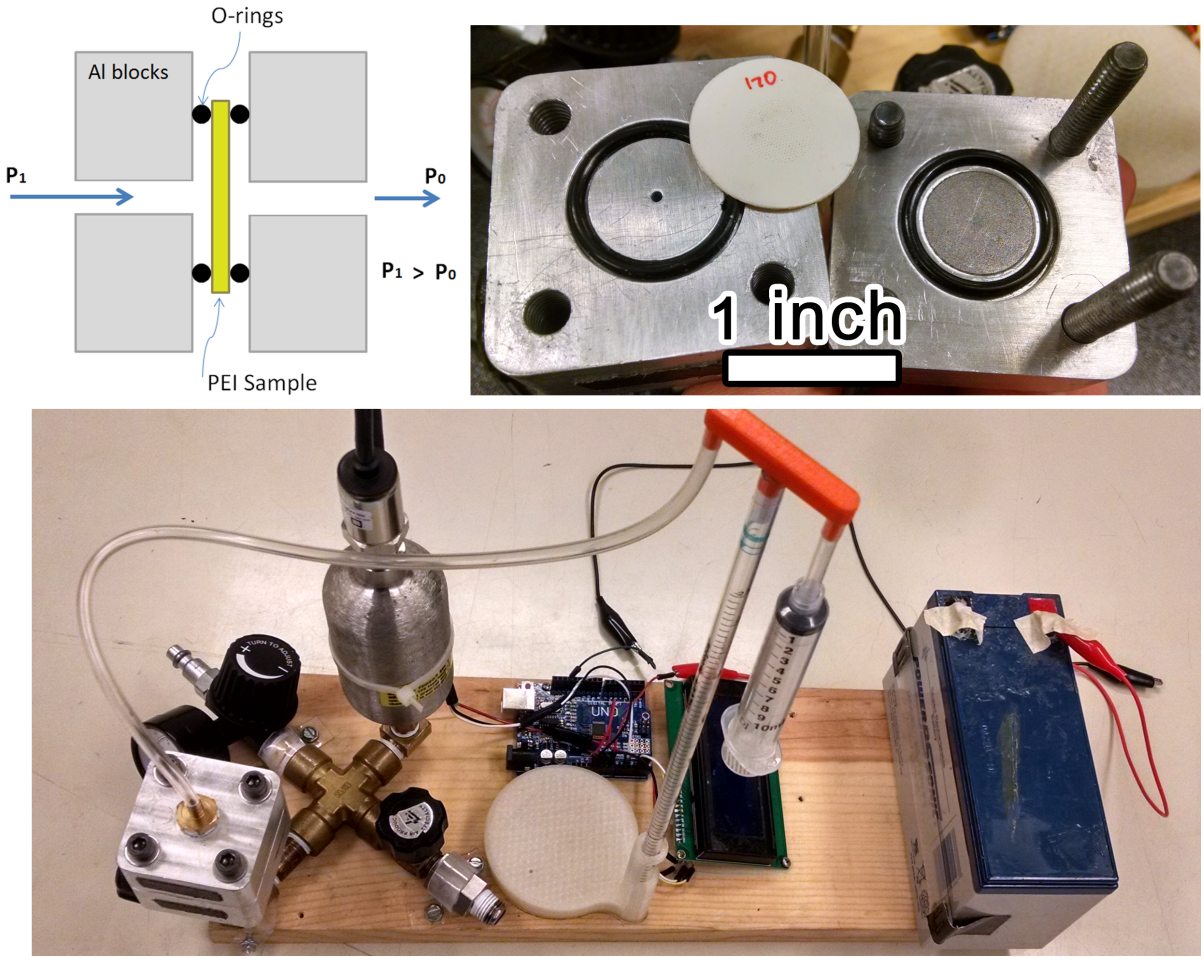

Figure 2.8: Diagram and photographs of the fluid flow apparatus.

To use the apparatus, a PEI sample is placed between the O-rings and the 4 bolts tightened with low torque as to compress the O-ring material and not the PEI sample itself. A gas source is needed, and I used shop air coming at 100 psi. A pressure regulator was used to drop the

pressure to a desired set point, and the analog sensor measures this pressure and sends the signal to the microcontroller which in turn displays it on the LCD screen. When the valve is open, the pressure is applied to the first face of the sample and, if permeable, flow will occur. The gas that passes through is transported via a flexible tube and collected in the pipette which is initially filled with water (this water is drawn up the pipette using the syringe). The fluid pressure at the bottom of the pipette is atmospheric, so the pressure at the water meniscus is 1 atm minus the pressure head of the column of water. Since this is equal to ρgh and $h \approx 0.1m$ this is approximately 1000 Pa. Since the ΔP across the sample is usually several atmospheres, this pressure head is less than 1% and therefore neglected. The pressure due to surface tension (capillary effect) was also neglected. The flow rate is found by simply timing the meniscus to drop 4 mL by watching the pipette and using a hand stopwatch. Three runs were performed for each sample to ensure that human reaction time and error were not a factor. Indeed, all three readings usually agreed to within 1%.

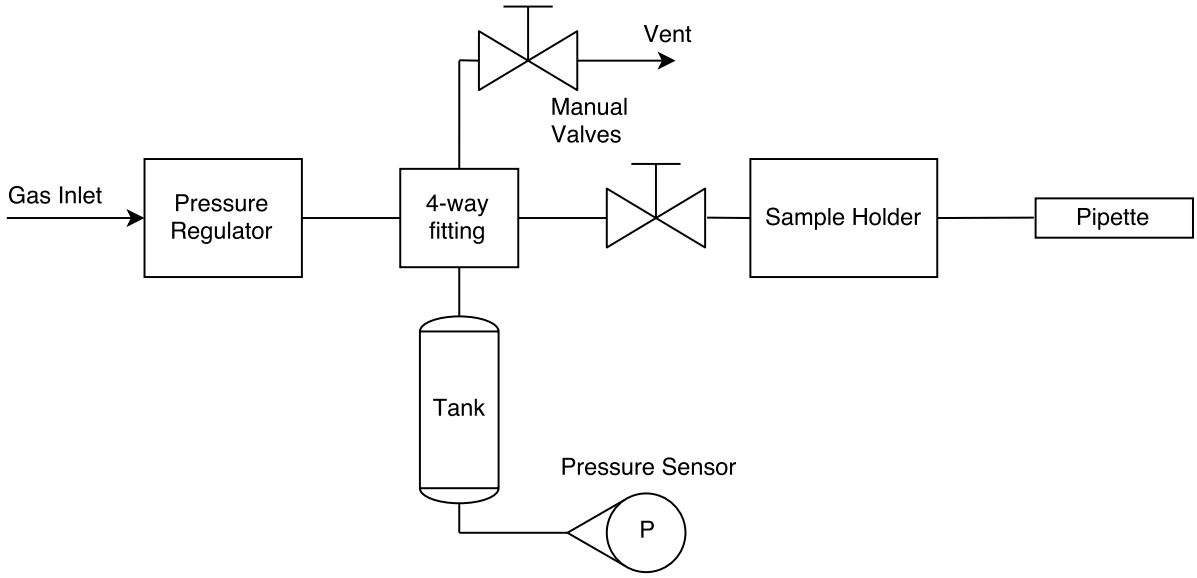

Figure 2.9: Plumbing diagram of the flow apparatus.

2.7 Diffusion Measurements

Diffusion moves mass not by a pressure gradient but rather a concentration gradient of different gas species. The substance to be diffused was placed into a machined aluminum canister atop which a pierced PEI nanofoam sample was placed to seal off the enclosure from the ambient environment. This was done by the use of a rubber O-ring and a screw-on lid to apply the sealing force. Figure (2.10) shows a diagram illustrating the principle of the experiment as well as the physical setup of the aluminum canister and accompanying components.

To find the mass flux through the sample, the entire canister was placed on a digital balance and the weight was periodically recorded. Tests were performed using an unfoamed PEI sample and confirmed that there were no leaks around the O-ring and polymer or canister interfaces. Table (2.3) shows these results for a canister containing isopropyl alcohol.

Figure 2.10: Vapor diffusivity test apparatus diagram and photograph of disassembled/assembled canister system.

Table 2.3: Leak test using an unfoamed PEI sample and isopropyl alcohol in the canister.

Hours Elapsed	Total Mass	Mass of liquid	Percent change
0	17.22986	0.77929	0.00%
2	17.23033	0.77976	0.06%
20	17.23352	0.78295	0.47%
23	17.23331	0.78274	0.44%
52	17.23566	0.78509	0.74%
72	17.23847	0.78790	1.10%
122	17.23369	0.78312	0.49%
191	17.23997	0.78940	1.30%
217	17.24001	0.78944	1.30%

This data shows that the sealing method is good. In normal experiments, 100% of the isopropyl would evaporate in less than 100 hours for a nanoporous sample with skin pierced. Therefore, this device proved to be an accurate research tool for diffusion experiments.

2.8 Intrusion Pressure

It was discovered that water would not intrude into PEI's nanoporous structure since PEI is hydrophobic and the pore structure has a large specific surface area. It was therefore of interest to find the intrusion pressure of water into various samples of the nanofoams. The first attempt was to modify the flow test apparatus to accommodate liquid water. The idea was to slowly ramp the pressure until leaking through the sample occurred. The presence of water was sensed by two wires placed in porous paper towel beneath the sample. When water was present, the resistance between the wires was reduced to a few mega-ohm and it could be sensed by

an operational amplifier circuit and outputted as a digital signal. This signal was read by a microprocessor which immediately recorded the pressure as given by the pressure transducer. To build up the required pressure, dry ice was placed into a pressure vessel and connected to the sample holder and the pressure slowly increased as the dry ice sublimated to gas CO_2.

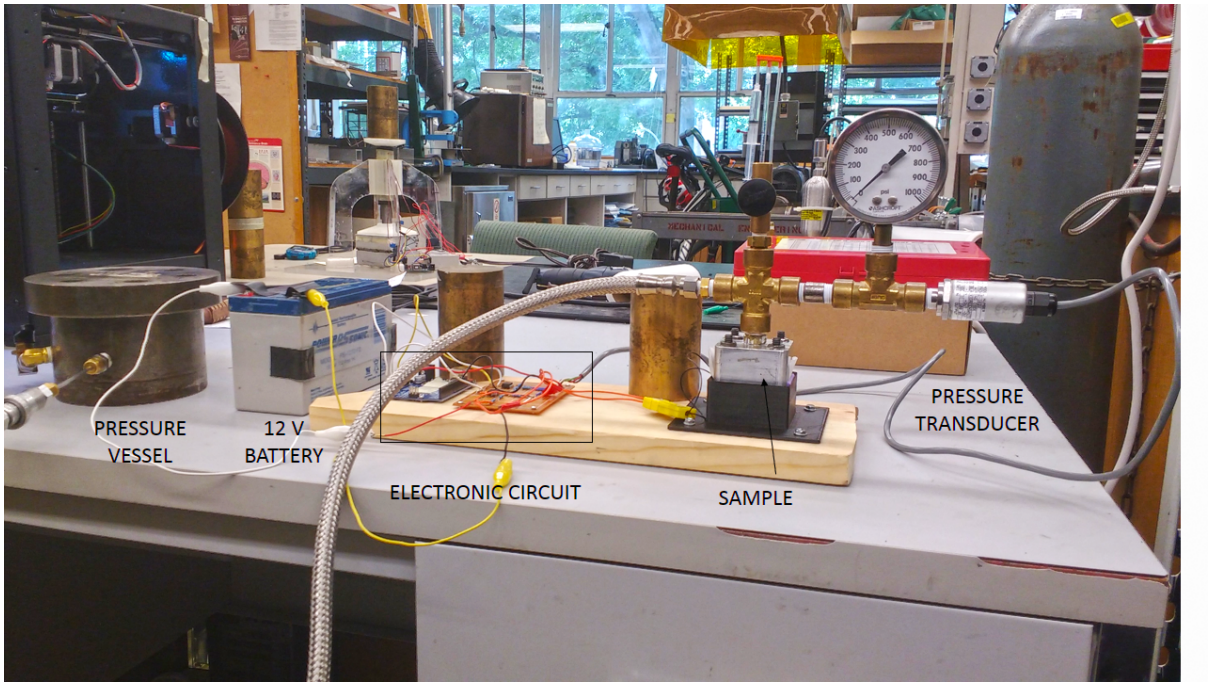

Figure 2.11: Photograph of the first apparatus built to find the intrusion pressure of water into nanoporous PEI.

This setup was about to reach 600 psi, at which point the O-rings ruptured. However, no water passed through the PEI sample. To test at higher pressure, a new method was used: the sample had one face sealed by a transparent polycarbonate sheet while the other side was open - see figure(2.12). Since the skin was pierced, there were small gaps of air trapped between the sample and the polycarbonate sheet - this air was at 1 atmosphere of pressure. This was placed underwater inside a pressure vessel. Dry ice again was used to elevate the pressure inside the vessel (and therefore of the liquid water). In this manner, the sample had 1 atm air on one face and water on the other, with water pressure equaling the pressure inside the vessel. After about 30 minutes at 800 psi, the samples were removed and inspected for water penetration. Weight measurements before/after the experiment showed that there was no water inside the nanostructure. Since the limit of our laboratory equipment was reached without successful water intrusion, I began looking elsewhere on campus to test at higher pressures. Finally, I was able to find a pressure vessel capable of 8000 psi water pressure. This experiment is detailed in a later chapter.

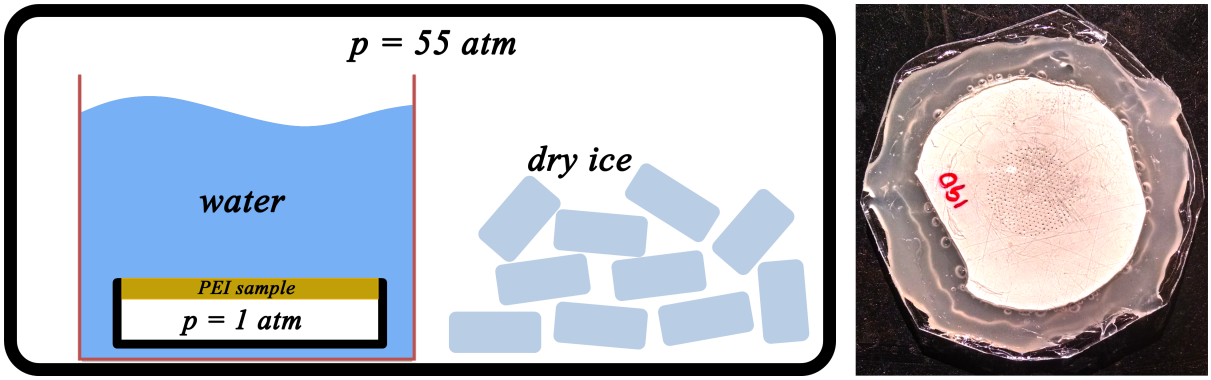

Figure 2.12: Left: Diagram of the second water intrusion test method. Right: Photograph of a sample used. Notice the pierced skin is visible through the polycarbonate sheet.

2.9 Image Processing

Determining the total area pierced by the holes as well as the hole size distribution was done by acquiring an image processing code I wrote in MATLAB. It uses the Image Processing Toolbox and it works by turning the photograph into a binary image and searching for connected regions. The full code is given in the appendix, so only the front end is described below.

First, the photograph is converted to grayscale and then turned into a binary image by specifying a cutoff brightness value (between 0 and 255). The connected regions of this binary matrix are found by the algorithm and the number of pixels (bits) in each region is stored in an array. The size of this array (after filtering small values) is the number of holes found in the image, which corresponds to the number of needles — approximately 367 on each side.

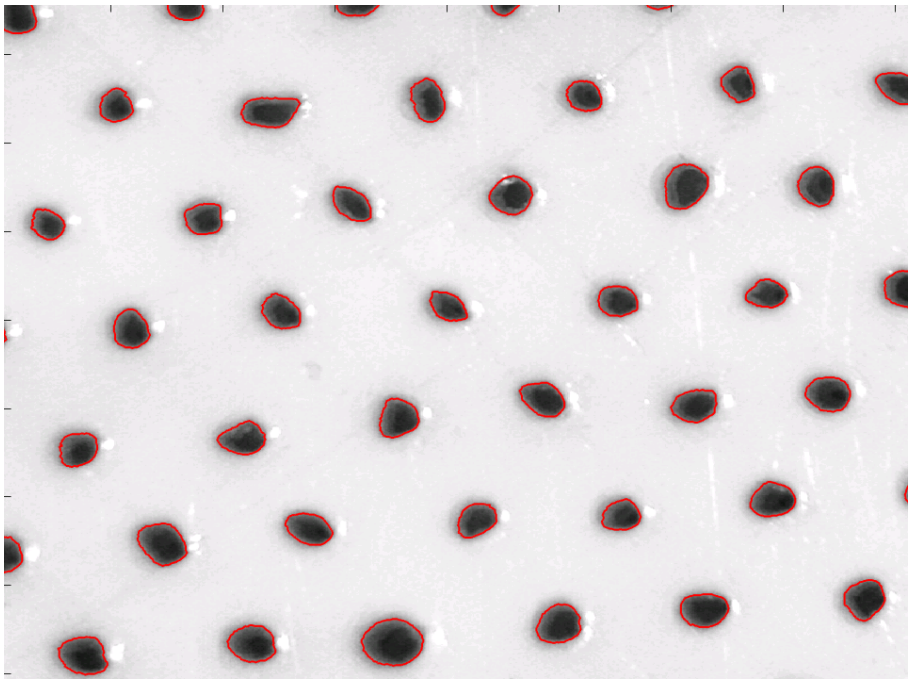

Figure 2.13: Needle hole boundaries found by the image processing algorithm and plotted in red.

An image calibration is performed by measuring the number of pixels per millimeter, thus giving a constant that converts pixels to mm. Since the holes are not perfect circles, the diameter of each hole is defined as the diameter of a circle which would have the same total area A, namely

$$d = \sqrt{\frac{4A}{\pi}} \tag{2.1}$$

A histogram of needle hole sizes can therefore be found. The average hole size in my experiments is about 160 microns. This parameter is very important for calculating the permeability of nanofoams as it will later be shown.

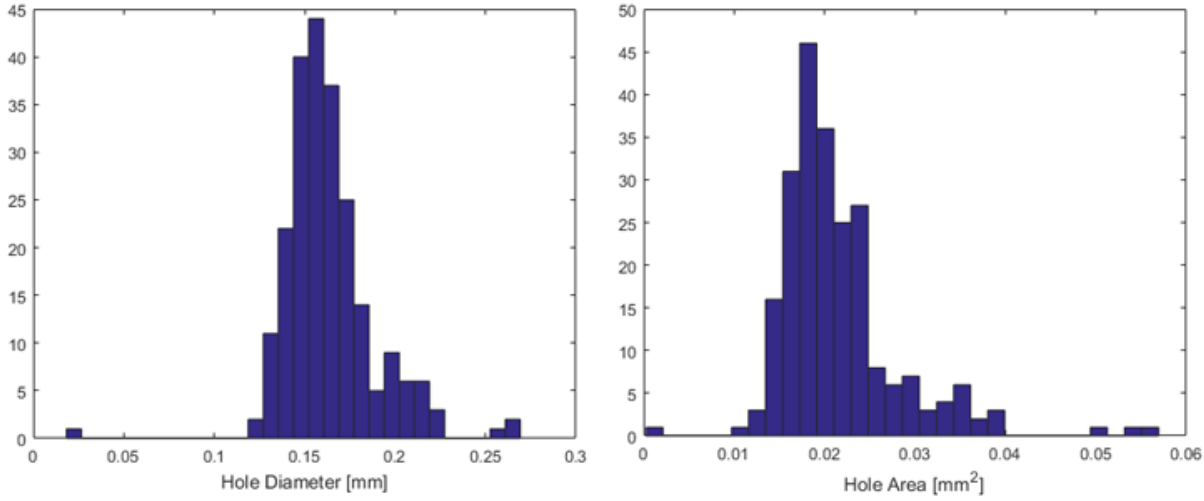

Figure 2.14: Histogram of needle hole diameters and areas for a representative PEI sample.

Image analysis was also used to test the standard deviation of needle hole depth. For this task, a pierced sample was cut with a razor blade through the holes and a photograph taken of its cross section, shown in figure (2.15). With this image, the standard deviation of needle depth can be calculated. The data is shown in table (2.4)

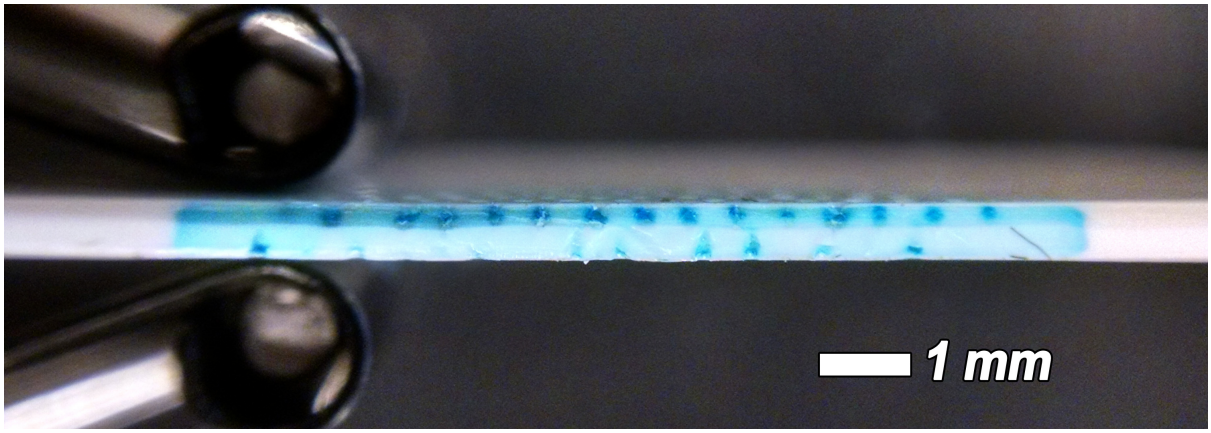

Figure 2.15: Cross section of PEI sample showing the depth of needle holes.

.

Table 2.4: Data showing the needle depth on the PEI sample shown in figure (2.15).

Depth [pixles]	Depth [mm]
82	0.222
81	0.219
83	0.224
76	0.205
75	0.203
82	0.222
78	0.211
77	0.208
81	0.219
77	0.208
84	0.227
84	0.227
76	0.205
79	0.214
mean:	0.215
standard deviation:	0.00852
st.dev / mean:	3.96%

Chapter 3

Fluid Flow Through a Cut Skin

3.1 Introduction

The dynamics of fluids in porous media is a mature subject, mostly used in earth science disciplines in the study of soils, sandbeds, and porous rocks [20]. Manufactured porous materials such as porous metal are also tested with the same principles of measuring flow rates across a pressure drop [24,25]. However, the porous samples presented in these cited works are not covered by impermeable skins which are selectively removed; thus, 1D flow is correctly assumed. However, in machining or piercing the skin of nanoporous PEI samples, the boundary conditions are mixed no-flux and constant pressure. Thus, the experimental data obtained by the standard methods cannot be translated into permeability values with the simple 1D equation. The scope of the work in this chapter is to create the mathematical model that correctly models the flow of a Newtonian fluid through a porous structure covered by selectively removed skin.

3.2 Measurement of Fluid Permeability

As seen in SEMs from refs [10–12, 15] the pore sizes of open-cell foams are always less than $10^{-7}m$, making the Reynolds number of any practical flow through it very small. The Navier-Stokes equations modeling flow through a porous medium at such low Reynolds numbers reduce to the general relation [20].

$$u_i = \frac{1}{\mu} K_{ij}(p_{,j} - \rho f_i) \tag{3.1}$$

Where K_{ij} are components of a symmetric 2^{nd} order tensor known as "permeability" which is an intrinsic property of the material, and f_i are components of a body force. If the fluid is incompressible, body forces are negligible, and the nanoporous structure is isotropic (as suggested by SEM), the equation can be simplified to

$$\mathbf{u} = -\frac{k}{\mu}\nabla p \tag{3.2}$$

In section (1.3) I showed that the pressure field of a Newtonian fluid inside the nanoporous structure satisfies Laplace's equation

$$\nabla^2 p = 0 \tag{3.3}$$

This will be the model used for all experiments in this study. Recall that in one dimension, $|\boldsymbol{u}|$ is a constant and p linear, and the following relation holds

$$|\boldsymbol{u}| = \frac{k}{\mu} \Delta p \tag{3.4}$$

A method of measuring permeability is given by ASTM D6539-13 and involves a simple apparatus which creates a fixed pressure boundary condition on a sample while measuring the volume flow rate of a gas through the sample. In this setup, the effective permeability k for a cylindrical sample of cross sectional area A and height h can be computed from ASTM D6539-13 equation (3). In SI units, the equation is

$$k = \frac{\mu Q h}{A \Delta p} \tag{3.5}$$

Removing the impermeable skin layer can be performed by milling, drilling, cutting, or piercing only a select region, keeping the remaining skin for structural strength. The flow field with such boundary condition is no longer one dimensional, so it can be expected that equation 3.5 will give erroneous values of permeability. This discrepancy must be quantified.

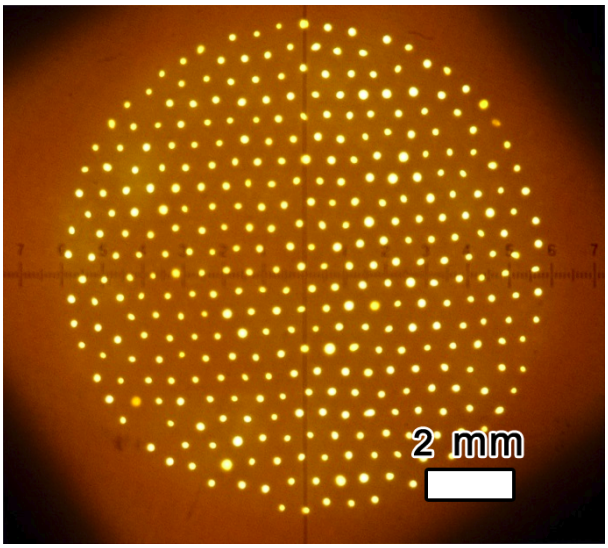

Figure 3.1: Selectively pierced PEI foam sample.

3.3 Effect of selectively removing the skin

If the skin is removed only in certain areas rather than the whole face, the solution to the pressure field is no longer a linear function. The deviation from the 1D solution must be known in order to accurately compute the permeability if such skin removing technique is used to make experimental samples.

This was done by mathematically modeling the problem and solving the continuity PDE for the pressure field. The model was restricted to 2D planar flow which approximates an

experiment where the skin cuts are long and narrow (such as when the skin is milled in straight lines). Let $p\,(x, y)$ be the pressure inside a nanofoam's cross section where the skins lie in the $y = 1$ and $y = -1$ planes. A cut of width $2a$ is made into the skin at $x = 0$, rendering the boundary permeable in that region. For mathematical purposes, let $k = \mu = 1$. Following this model, two problems were posed. The first problem has the following domain and boundary conditions

$$\nabla^2 p\,(x, y) = 0, \quad x \in [-\pi, \pi]\ ,\ y \in [-1, 1] \tag{3.6}$$

$$p\,(|x| < a,\ 1) = 1\ ,\ p\,(|x| < a,\ -1) = -1,\ \ a \in \mathbb{R} > 0 \tag{3.7}$$

$$\frac{\partial p}{\partial y} = 0\ \text{ for }\ |x| > a,\ y = 1,\ y = -1 \tag{3.8}$$

$$\frac{\partial p}{\partial x} = 0\ \text{ for }\ |x| = \pi \tag{3.9}$$

A method to obtain a solution is by eigenfunction expansion with $p = X(x)Y(y)$ giving the ODEs

$$Y'' = \zeta^2 Y,\ \ X'' = -\zeta^2 X \tag{3.10}$$

where the sign of ζ^2 chosen to yield solutions that can satisfy the boundary conditions and anti-symmetry about $y = 0$. For the $X(x)$ ODE, the solution is

$$X = c_1 \sin \zeta x + c_2 \cos \zeta x \tag{3.11}$$

By applying the boundary condition of $p = XY$ we have

$$X'(0) = 0 = c_1 \zeta\, \cos(\zeta \cdot 0) - c_2 \zeta\, \sin(\zeta \cdot 0) \to c_1 = 0 \tag{3.12}$$

$$X'(\pi) = 0 = -c_2 \zeta\ \sin \zeta\pi \tag{3.13}$$

which means that c_2 is arbitrary, and

$$\zeta \in \mathbb{Z} > 0\ \ \therefore X(x) = \tilde{c}_\zeta \cos \zeta x + \tilde{c}_0 \tag{3.14}$$

For the $Y(y)$ ODE the solution is

$$Y = c_3 e^{\zeta y} + c_4 e^{-\zeta y} + c_5 y + c_6 \tag{3.15}$$

However, $c_6 = 0$ to satisfy the boundary conditions. Now multiplying the two solutions for X and Y and absorbing constants, the general solution is

$$p = b_0 y + \sum_{\zeta=1}^{\infty} b_\zeta\, \cos \zeta x \sinh \zeta y \tag{3.16}$$

For a numerical approximation to $p(x, y)$, the series is truncated at $\zeta = N$ and a set of M linear equations can be written by enforcing the boundary conditions for M unique points at $y = 1$ then arranging into a matrix A of size $M \times N$. If $M = N$, a solution can be found by matrix inversion. If $M > N$, a least squares solution can be found.

As N gets large, the condition number of A increases rapidly due to the $\sinh \zeta$ terms, and the linear system becomes unsolvable by computer. To reduce the condition number of A, a parameter c_ζ was defined by

$$b_\zeta = \frac{c_\zeta}{\zeta \ \sinh \zeta y} \tag{3.17}$$

A well-conditioned linear system in c_ζ is first solved, then b_ζ computed from the definition.

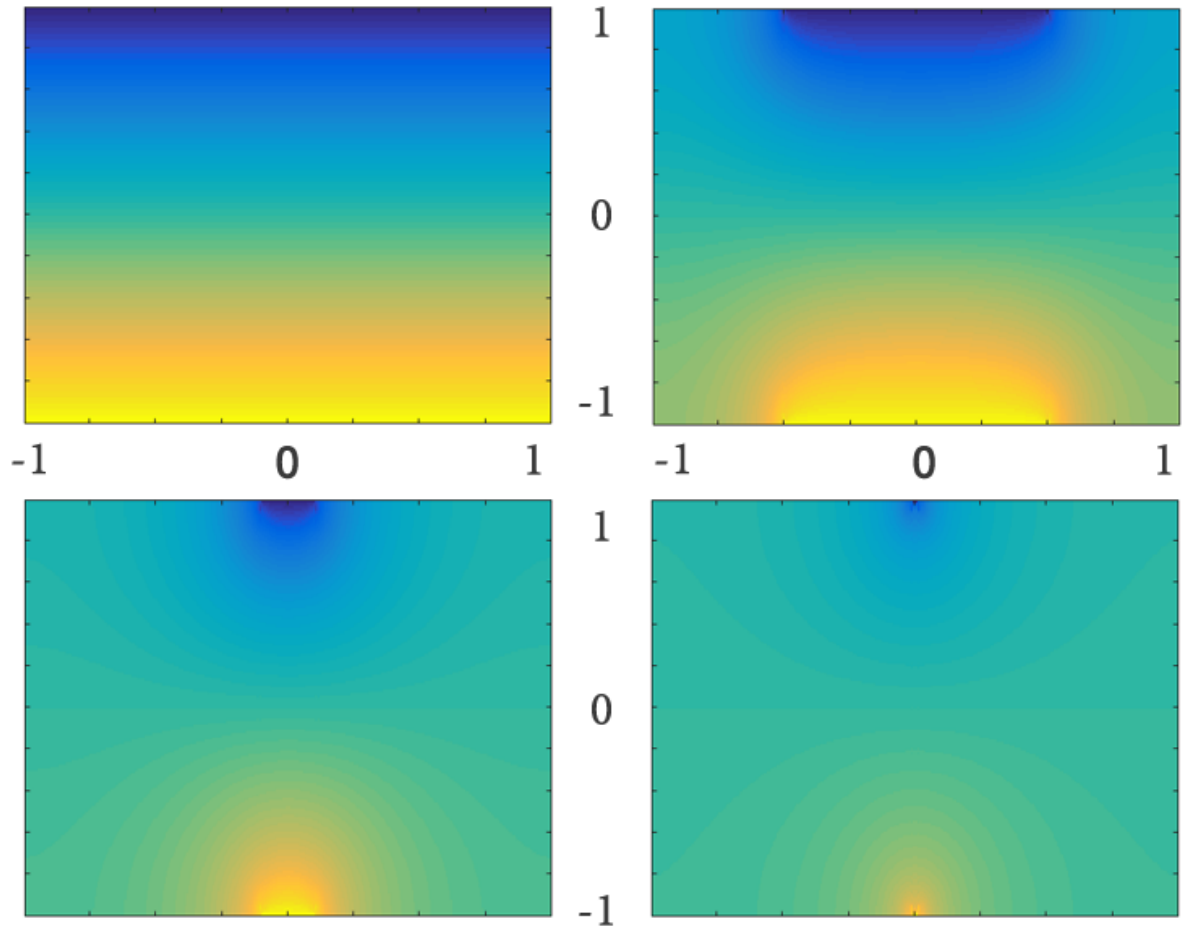

Figure 3.2: Color plot of pressure given by the separation of variables method. The value of a is 1, 0.5, 0.1, 0.01. Note that as a decreases, the solution becomes increasingly nonlinear. The $a = 1$ case appears to approximate the 1D case.

By truncating the series for p, the Gibbs phenomenon occurs due to the discontinuity of flux at point $(a, 1)$. To ensure that the solution was not inaccurate due to the truncation of terms, the PDE was also solved by a different method and compared. The domain for the second problem includes the entire real x-axis, and the boundary condition is a prescribed flux of 1 for $x \in [-a, a]$ rather than a prescribed pressure.

$$\nabla^2 p\,(x, y) = 0, \quad x \in [-\infty, \infty] \ , \ y \in [-1, 1] \tag{3.18}$$

$$\frac{\partial p}{\partial y} = f\left(x\right) = \begin{cases} 1, & -a \leq x \leq a \\ 0, & \text{otherwise} \end{cases} \quad \text{at } \; y = 1, y = -1 \tag{3.19}$$

The Fourier transform of the PDE is taken with respect to x to yield an ODE in variable y:

$$\mathcal{F}\left\{\frac{\partial^2 p}{\partial x^2} + \frac{\partial^2 p}{\partial y^2} = 0\right\} = -\omega^2 P\left(\omega,\ y\right) + \frac{\partial}{\partial y} P(\omega, y) = 0 \tag{3.20}$$

This has general solution

$$P(\omega, y) = b\ \cosh \omega y + c\ \sinh \omega y \tag{3.21}$$

Due to the anti-symmetry about $y = 0, \;\; b = 0$. To find c, the boundary condition is first transformed into the frequency domain and then imposed at $y = 1$.

$$\left.\frac{dP}{dy}\right|_{y=1} = c\omega\ \cosh\ \omega\ = \mathcal{F}\left\{f(x)\right\} = \frac{\sin a\omega}{\pi\omega} \tag{3.22}$$

Therefore the unique solution is

$$p\left(x, y\right) = \mathcal{F}^{-1}\left\{P\right\} = \int_{-\infty}^{\infty} \frac{\sin\ a\omega\ \ \sinh\ \omega y}{\pi w^2\ \cosh\ \omega}\ e^{-i\omega x} d\omega \tag{3.23}$$

However, due to symmetry about $x = 0$

$$\int_{-\infty}^{0} P\left(\omega, y\right)\ e^{-i\omega x} d\omega = \text{conjugate}\left(\int_{0}^{\infty} P\left(\omega, y\right)\ e^{-i\omega x} d\omega\right) \tag{3.24}$$

This fact can be used to cut computation time in half.

The result is that the pressure at any point in the domain can be found by computing one improper integral. Several computer algorithms exist that can approximate definite integrals with infinity limits.

Floating-point overflow occurs as ω gets large, since the $\sinh \omega y$ and $\cosh \omega$ terms get computed individually by the computer. To prevent this, $P(\omega, y)$ is rewritten as

$$P(\omega, y) = \frac{e^{\omega(y-1)} - e^{\omega(1-y)}}{1 + e^{-2\omega}} \frac{\sin\ a\omega}{\pi w^2} \tag{3.25}$$

which is computationally stable for $\omega \gg 1$. After applying the simplifications, the final result is

$$p(x, y) = 2\ \text{real}\left(\int_{0}^{\infty} \frac{e^{\omega(y-1)} - e^{\omega(1-y)}}{1 + e^{-2\omega}} \frac{\sin\ a\omega}{\pi w^2} e^{-i\omega x} d\omega\right) \tag{3.26}$$

3.4 Apparent Permeability

In the 1D flow model such as that assumed in ASTM D6539-13, the flow rate is directly proportional to the boundary opening. In higher dimensions this is not necessarily the case. However, we can define an *apparent permeability* in analogy with the 1D case. The relation between the apparent permeability k_{app} and total flow rate $Q = \iint \nabla p \cdot \boldsymbol{n}\ dS$ is therefore

$$k_{app} = \frac{Q(y_2 - y_1)}{2a\,(p_2 - p_1)} \tag{3.27}$$

Following the parameters in the first problem, this is simply $k_{app} = Q/2a$. Further,

$$Q = \int_{-\pi}^{\pi} \frac{\partial p}{\partial y} dx = \int_{-\pi}^{\pi} \left(b_0 + \sum_{\zeta=1}^{\infty} b_\zeta \cos \zeta x \cosh \zeta y \right) dx = 2\pi b_0 \tag{3.28}$$

This simple expression is the result of the convenient domain choice $x \in [-\pi, \pi]$ in the first problem. The effective permeability is therefore

$$k_1 = \pi b_0 / a \tag{3.29}$$

In the second problem, the pressure at the boundaries is not constant, but the flux is prescribed to be 1 on $x \in [-a, a]$. Therefore, the apparent permeability is defined in this manner

$$k_2 = \frac{2}{\Delta \overline{p}} \tag{3.30}$$

where $\Delta \overline{p}$ is the difference between the average pressure at the boundaries. By the antisymmetric nature of the solution

$$\Delta \overline{p} = \frac{2}{a} \int_0^a p\,(x, 1)\, dx = 2 \text{ average } (p\,([-a, a]\,, 1)) \tag{3.31}$$

As $a \to 0,\ k_{app} \to \infty$ as shown in the figure 1.2. The small difference between k_1 and k_2 is due to the finite domain of problem 1, therefore as $a \to \pi,\ k_1 \to k = 1$ whereas in the problem 2 as $a \to \infty,\ k_2 \to k = 1$. However, the deviation of k_{app} from k in either case becomes pronounced for $a < 1$. In an experimental sense, one way to think of a is the ratio of slit width to specimen's thickness (since the slit width is $2a$ and $-1 \leq y \leq 1$). If this slit is narrower than the specimen's thickness, applying equation (3.5) for experimentally calculating the permeability will introduce significant error; therefore, a correction must be applied. This important correction is discussed next.

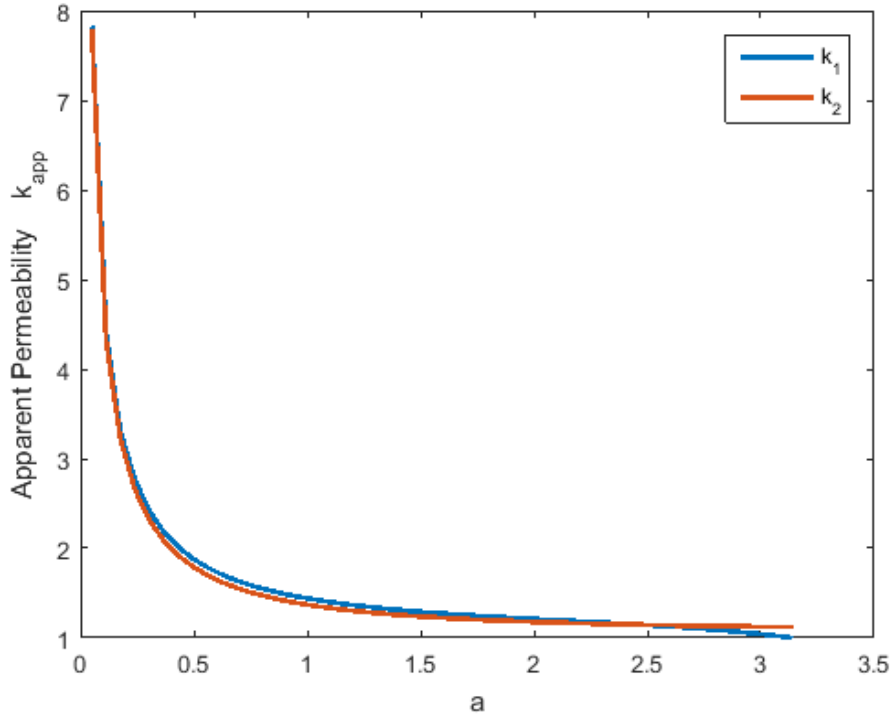

Figure 3.3: The error in using equation (3.5) to calculate permeability grows rapidly as the slit size approaches zero.

3.5 Effective Slit Width

Since the flow through a slit of width $2a$ in the infinite domain is larger than in a domain of width $2a$ (1D solution), it is convenient to define an *effective slit width* a_{eff} which is one that has equal flow in the 1D solution as the 2D solution with slit width a. Specifically,

$$Q = A\nabla p = 2a_{eff}\Delta p/h = a_{eff}\Delta p \tag{3.32}$$

In the context of problem 1, this defines

$$a_{eff} = \pi b_0 \tag{3.33}$$

and for problem 2

$$a_{eff} = \frac{2a}{\Delta \overline{p}} \tag{3.34}$$

For $a \gg 0.1$, the difference between a_{eff} and a is approaches 0.357. This result is significant – it states that equation (3.5) can be used to calculate the permeability if the slit width used in the calculation is corrected to the effective slit width by the simple relation

$$a_{eff} \approx a + 0.357\ , \quad (a > 0.1) \tag{3.35}$$

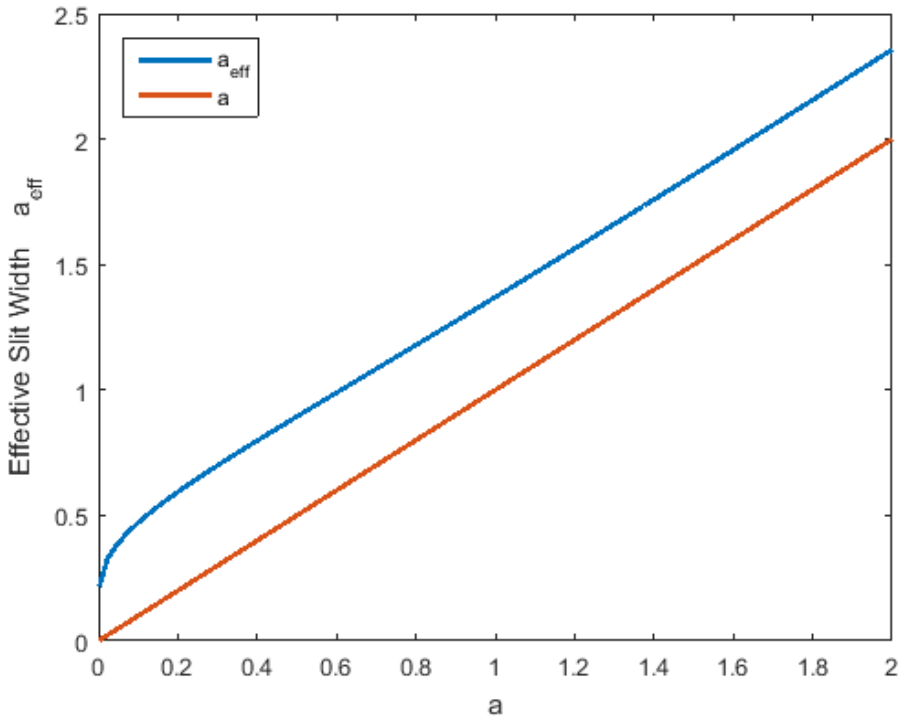

Figure 3.4: The effective slit width for $a > 0.2$ is approximately equal to a constant plus a.

3.6 Multiple slits

In a real experiment there may be a plurality of evenly spaced slits. In that case, the solution can be found by using the linearity of Laplace's equation to superpose appropriately shifted solutions of a single slit. Let $p(x, y)$ be the solution with one cut of width $2a$ centered at $x = 0$ (as before) and let $p_M(x, y)$ be the solution with an additional M number of cuts on either side, all separated by a distance s between the adjacent cuts for a total number of $2M + 1$ slits (and symmetric about $x = 0$). Then

$$p_M(x, y) = p(x, y) + \sum_{m=1}^{M} p(x + m(2a + s), y) + p(x - m(2a + s), y) \tag{3.36}$$

For this case, a_{eff} can be similarly computed, however it will be dependent on $M,\ s$. The case of interest is when M is very large. However, since the solution decays quickly with $x,\ M$ can be truncated when the farthest neighbor ends at $x_1 \approx 4$. The reasoning is that the apparent permeability was a weak function of a at $a = 4$ as seen in figure 3.4. Thus,

$$M(2a + s) + a \geq x_1 \tag{3.37}$$

should be satisfied. Since $M \in \mathbb{Z}^+ > 1$, a good value of M is

$$M = \lceil x_1/(2a + s) \rceil \tag{3.38}$$

Scaling M in this fashion also reduces computation time by keeping M small when appropriate.

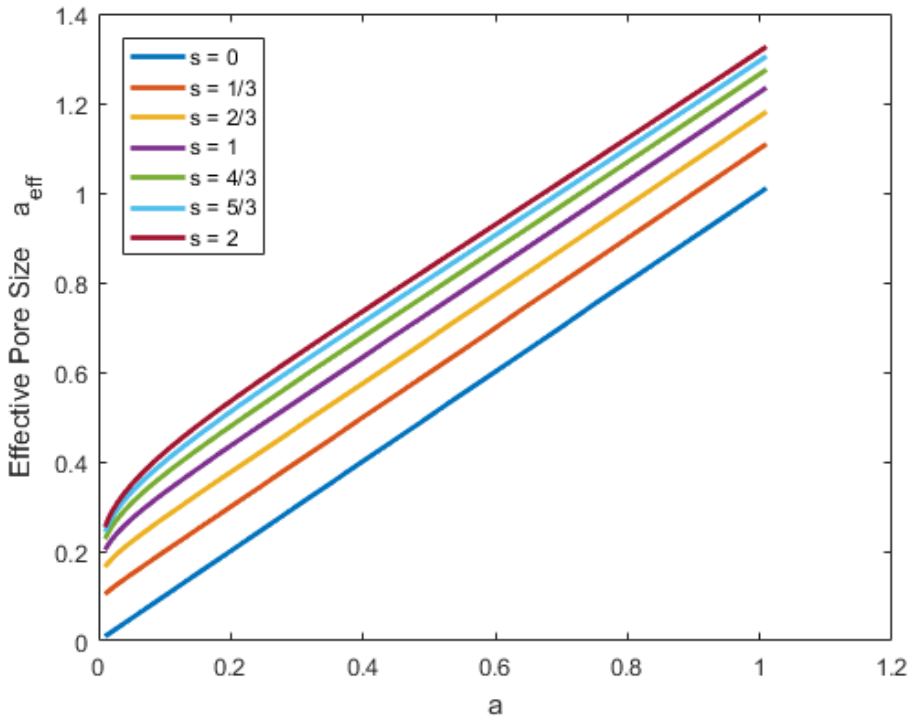

Figure 3.5: Effective slit width for various values of s starting with $s = 0$ (blue) up to s = 2 (purple).

The $s = 0$ case is equivalent to one slit with $a = 4$, which, as seen in figure 3.3, is a great approximation to the 1D solution. For small spacing, a_{eff} is a linear function of a down to smaller values of a than in the single slit case. To better see the relationship between a_{eff} and a for different values of s, the plot in figure 3.5 was generated for several values of s. As s gets larger, the curve approaches the one slit case in figure 3.4.

The plot in figure 3.6 shows the number that must be added to a in order to compute a_{eff} as a function of s. When experimentally calculating k, equation 3.5 can be used as long as a is replaced by the appropriate value of a_{eff} as given in this plot.

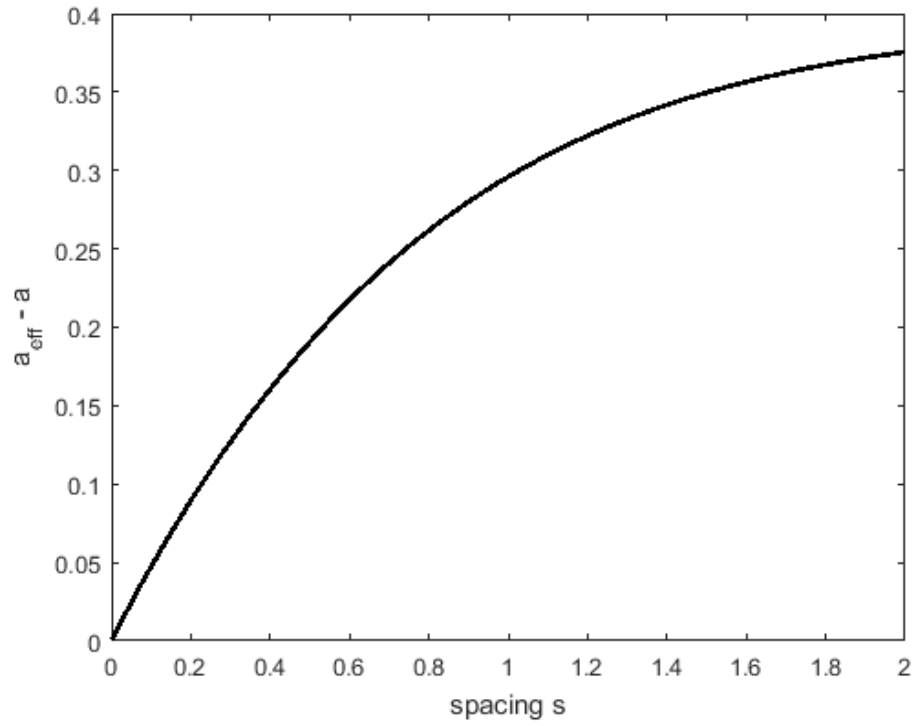

Figure 3.6: Plot of $\boldsymbol{a_{eff} - a}$ as a function of *s*. When *s* = 0, no correction must be done as this is equivalent to the 1D case (removing the entire skin from a sample). When *s* is large, it is the equivalent of an isolated slit as in figure 3.4.

3.7 Circular holes

In some experiments, the specimen's skin may be drilled instead of cut in a slit geometry. These holes require the model to be solved in cylindrical coordinates (with axial symmetry). Proceeding in a similar fashion as previously, the PDE and boundary conditions are posed as

$$\frac{\partial^2 p}{\partial r^2} + \frac{1}{r}\frac{\partial p}{\partial r} + \frac{\partial^2 p}{\partial z^2} = 0 \ , \ \ r \geq 0, \ \ -1 \leq z \leq 1 \tag{3.39}$$

$$\frac{\partial p}{\partial y} = g\left(r\right) = \begin{cases} 1, & |r| \leq a \\ 0, & \text{otherwise} \end{cases} \qquad \text{at } \ z = 1, \ \ z = -1 \tag{3.40}$$

The Hankel transform of order 0 (Fourier-Bessel transform) is applied to the PDE. It has the transform pair defined as

$$\mathcal{H}_0\{f\left(r\right)\} = F\left(q\right) = \int_0^\infty f\left(r\right) J_0\left(qr\right) r dr \tag{3.41}$$

$$f\left(r\right) = \mathcal{H}_0^{-1}\{F\left(q\right)\} = \int_0^\infty F\left(q\right) J_0\left(qr\right) q dq \tag{3.42}$$

where $J_0(r)$ is the zeroth order Bessel function of the first kind. This transform has the property of transforming the Bessel operator to multiplication by $-q^2$ [26]. The PDE is thus transformed

to the following ODE in z

$$\mathcal{H}_0\left\{p\frac{\partial^2 p}{\partial r^2}+\frac{1}{r}\frac{\partial p}{\partial r}+\frac{\partial^2 p}{\partial z^2}=0\right\}=-q^2P+\frac{\partial^2}{\partial z^2}P=0 \tag{3.43}$$

which is the same ODE as obtained in the second problem after applying the Fourier transform. The solution, with the antisymmetric condition applied, is

$$P=c\sinh(qz) \tag{3.44}$$

Taking the Hankel transform of the boundary condition

$$\mathcal{H}_0\left\{g\left(r\right)\right\}=G\left(q\right)=\int_0^\infty g\left(r\right)J_0\left(qr\right)rdr=aJ_1(qa)/q \tag{3.45}$$

Applying the boundary condition in Hankel space

$$\frac{\partial P}{\partial z}=qc_1\cosh\, qz=G(q)=aJ_1(aq)/q \tag{3.46}$$

where J_1 is the order 1 Bessel function of the first kind. Therefore,

$$c_1=\frac{aJ_1(aq)}{q^2\cosh q} \tag{3.47}$$

and the transformed solution is

$$P(q,z)=aJ_1(aq)\mathrm{sech}(q)\sinh(qz)/q^2 \tag{3.48}$$

If we are only interested in the solution at $z=\pm1$ and not on the entire domain, this can be simplified computationally as

$$P(q,1)=aJ_1(qa)\ \tanh(q)/q^2 \tag{3.49}$$

Finally, taking the inverse Hankel transform gives the pressure

$$p\left(r,1\right)=\mathcal{H}_0^{-1}\left\{P\left(q,1\right)\right\}=\int_0^\infty\frac{a}{q}J_1\left(qa\right)J_0\left(qr\right)\tanh\left(q\right)\,dq \tag{3.50}$$

The integral converges for all $r\geq0$ since

$$\lim_{q\to\infty}J_1\left(qa\right)J_0\left(qr\right)=0,\ \ a>0,\ r\geq0 \tag{3.51}$$

Analysis on the circular holes can now be performed.

3.8 Apparent permeability for circular holes

For the circular holes problem, the apparent permeability is defined in the same manner:

$$k_{app}=\frac{2}{\Delta\overline{p}} \tag{3.52}$$

with

$$\Delta\overline{p} = \frac{2}{A}\iint_S p\, dS = \frac{2}{\pi a^2}\int_0^{2\pi}\int_0^a p\,(x,1)\, r dr \approx \frac{4}{a}\,\text{average}\,(p\,([0,a]\,,1)\,r\,) \tag{3.53}$$

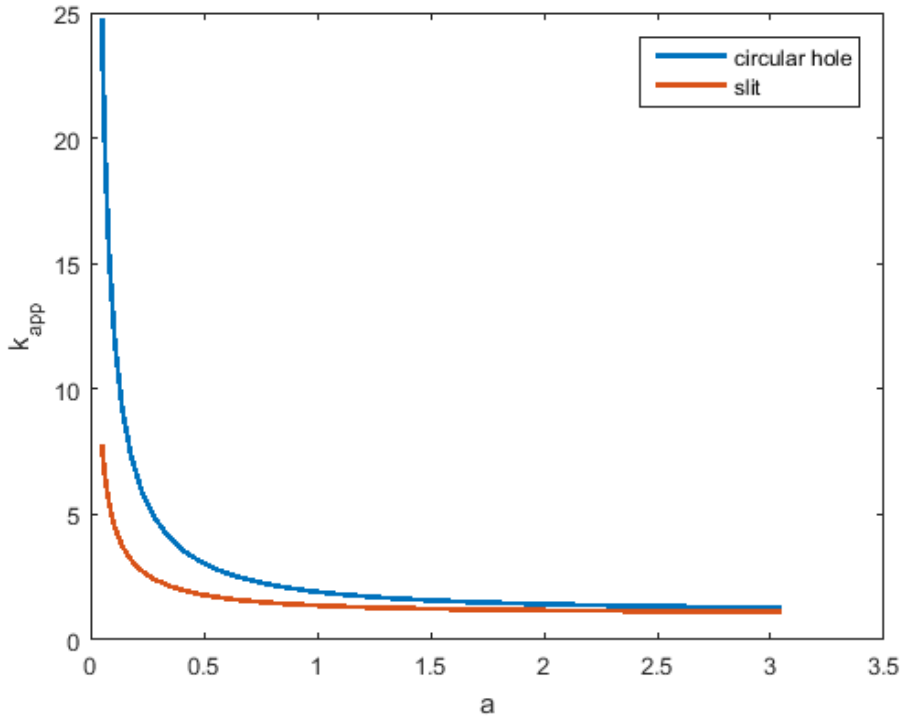

Figure 3.7: Apparent permeability for the circular hole case (blue) and slit case from figure 3.3 (in red).

The error in applying equation (3.5) for calculating k is much larger for the case of circular holes than for long slits, and also decays slower with increasing a.

3.9 Effective Hole Size

The effective hole size is also defined in the same manner as before

$$Q = \pi a_{eff}^2 \nabla p = \ \pi a_{eff}^2 \Delta\overline{p}/2 \ = \pi a^2 \tag{3.54}$$

Therefore, the effective hole size for the circular hole case is

$$a_{eff} = \sqrt{2a^2/\Delta\overline{p}} \tag{3.55}$$

As in the slit case, the effective hole size can be approximately related to a by an additive constant. The error becomes small when $a > 0.5$.

$$a_{eff} \approx a + 0.387 \tag{3.56}$$

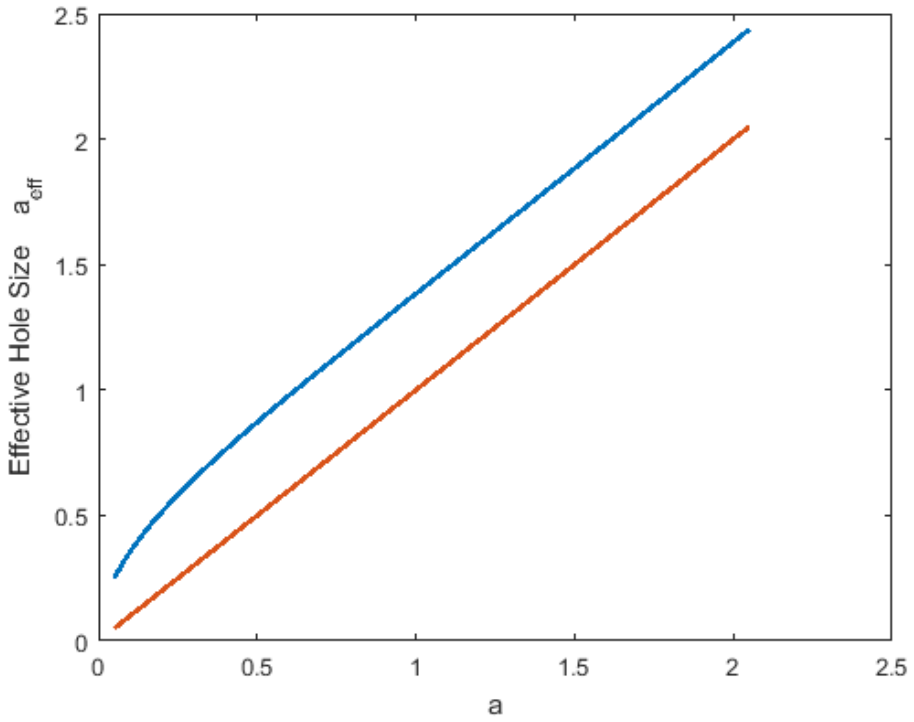

Figure 3.8: For circular holes, the effective hole size is also related to a by an additive constant for large a.

3.10 Multiple holes

Placing holes on a plane can be done in various patterns. If an array of needles is constructed, the holes will resemble a hexagonal pattern since this is the densest packing. Therefore, the following analysis is performed for this geometry. Let b denote the distance between circle centers, and s denote the smallest distance between adjacent circles. It follows that $b = s + 2a$. To find the solution to a domain with multiple holes, the coordinates of each hole center must be known so that the solution for one hole can be appropriately shifted and superposed. To do this, the pattern is broken up into two rows, even and odd rows, as shown in figure 3.9. These two rows form a basis unit that can be stacked by vertically shifting an arbitrary number of times to create an array of hexagonal packed circles.

The coordinates of the circle centers in terms of $m, n \in \mathbb{Z}$ are given by

$$x = nb, \quad y = mb\sqrt{3} \quad \text{(even row)} \tag{3.57}$$

$$x = \left(n + \frac{1}{2}\right) b\,, \quad y = \left(m + \frac{1}{2}\right) b\sqrt{3} \quad \text{(odd row)} \tag{3.58}$$

These equations can be used to generate an arbitrary number of points representing the centers of neighboring holes. As in the slit problem, the effect of neighbors far away becomes negligible, so the neighbors will be generated up to a distance of $r_o \approx 15$ from the center hole. Thus, the maximum values of m and n (M and N respectively) are computed by

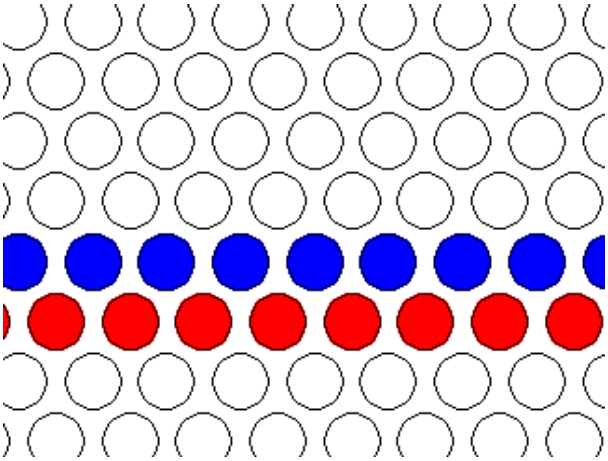

Figure 3.9: Hexagonal circular packing. The two repeating rows are colored.

$$N = \lceil r_o/b \rceil \quad , \quad M = \left\lceil r_o/2b\sqrt{3} \right\rceil \tag{3.59}$$

This creates a rectangular pattern. To create a circular one, a constraint is imposed on the points that belong to the set of neighbors:

$$\sqrt{x^2 + y^2} \leq r_o \tag{3.60}$$

Since the solution is only dependent on r and z, neighbors at the same distance have the same contribution to total flux through the central hole (centered at the origin), even though they have different spatial coordinates. This can be used to reduce computation time by combining holes at similar distances to the origin, specifically, holes whose centers are within tol of each other. The value of tol is arbitrarily set to $a/10$.

The apparent permeability is calculated as before – by finding the average pressure across the center hole and using equation (3.55). Referring to figure (3.10), the contribution to the pressure from each hole can be computed by integrating the isobaric arcs along the circle's diameter.

$$r^2 + (r+\alpha)^2 - 2r\,(r+\alpha)\cos\theta - a^2 = 0 \tag{3.61}$$

$$c = 2r\theta = 2r\cos^{-1}\left(\frac{2r^2 + 2\alpha r + \alpha^2 - a^2}{2r\,(r+\alpha)}\right) \tag{3.62}$$

The average pressure due to one distant hole can be computed by

$$\overline{p} = \frac{1}{\pi a^2}\int_{-a}^{a} c\,p\,(r+\alpha, 1)\,d\alpha \;\approx\; \frac{2}{\pi a}\text{average}\left(2r\cos^{-1}\left(\frac{2r^2 + 2\alpha_j r + \alpha^2 - a^2}{2r\,(r+\alpha_j)}\right)\; p\,(r+\alpha_j)\right) \tag{3.63}$$

where $a_j \in [-a, a]$ is a set of equidistant points. This is applied for $r > a$, i.e. not for the central hole.

The nonlinearity of $a_{eff} - a$ versus a for $a < 0.5$ is can be seen in figure (3.11). However, for larger a, the result is similar to a slit case. That is, a single curve is sufficient for determining

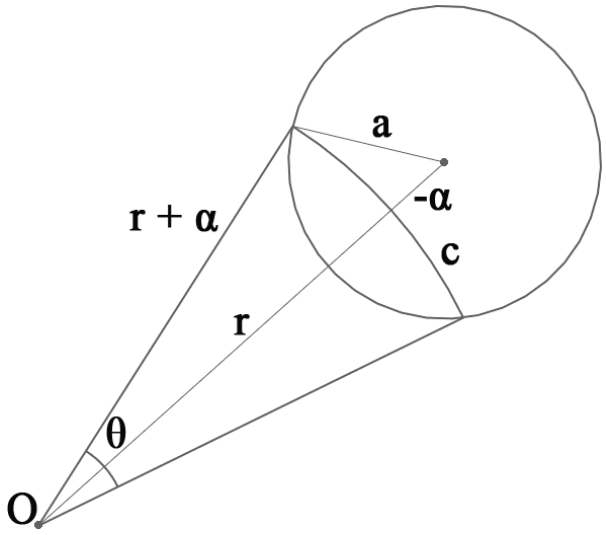

Figure 3.10: A hole at distance r from the origin.

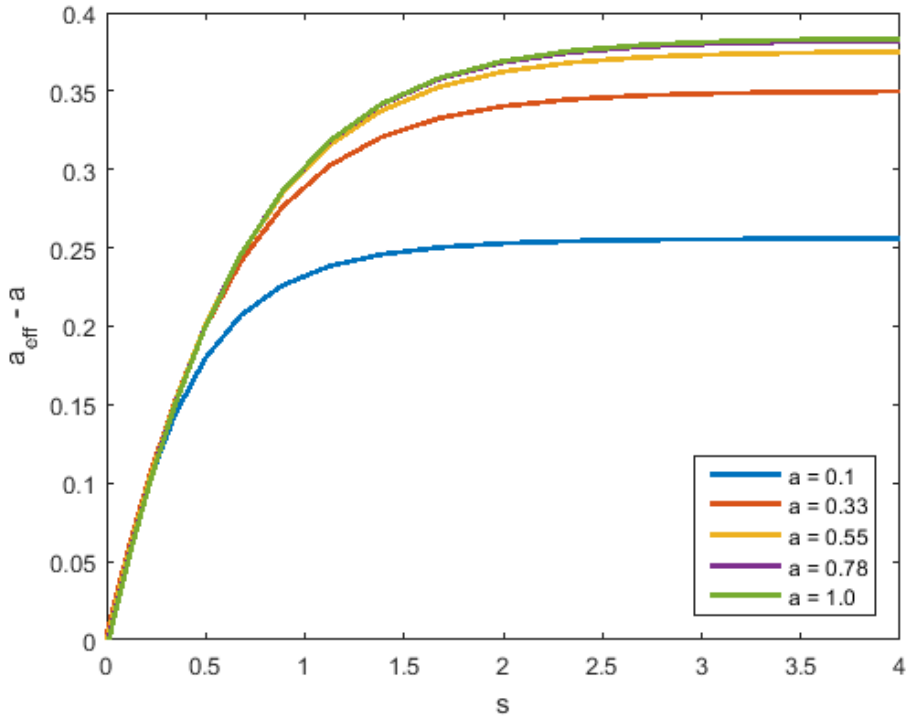

Figure 3.11: Plots of $a_{eff} - a$ for various values of s and a.

the apparent hole size for a given spacing, and thus correct calculations of permeability given by equation (3.5).

3.11 Experimental Verification of the Model

To test the validity of the model in practice, PEI nanofoam samples processed under identical conditions had skins removed in different patterns - drilling and needle piercing, corresponding to different values of a and s. The total exposed area, mean diameter, and thickness was computed from photographs of the samples by image processing using MATLAB. Following the model, the values of a and s are normalized by the nanofoam's thickness h, and thus are dimensionless. Therefore, the value of a is computed by dividing the hole's diameter by h, and s computed by dividing the hole spacing by h. This was programmed into a MATLAB script which only requires as input the image, the conversion factor from pixels to millimeters, and the spacing between holes measured in pixels. It then outputs the total area, total effective area, and various statistics about the hole size distribution such as mean and standard deviation. For example, in the case of holes drilled with a 0.5 mm twist drill, the computed mean hole diameter was 0.5113 mm and standard deviation 0.0112 mm. For the needle pierced sample in figure (3.1), the average hole diameter was 0.182 mm and standard deviation 0.0277 mm. All samples had a thickness of 0.39 mm. For these two cases, the effective hole size was computed, along with the effective hole area. The samples were tested for flow rate by applying a constant pressure differential and measuring volume flow per unit time. The permeability of both samples was thus computed with and without the correction of the present model for comparison. The results are shown in table (3.1).

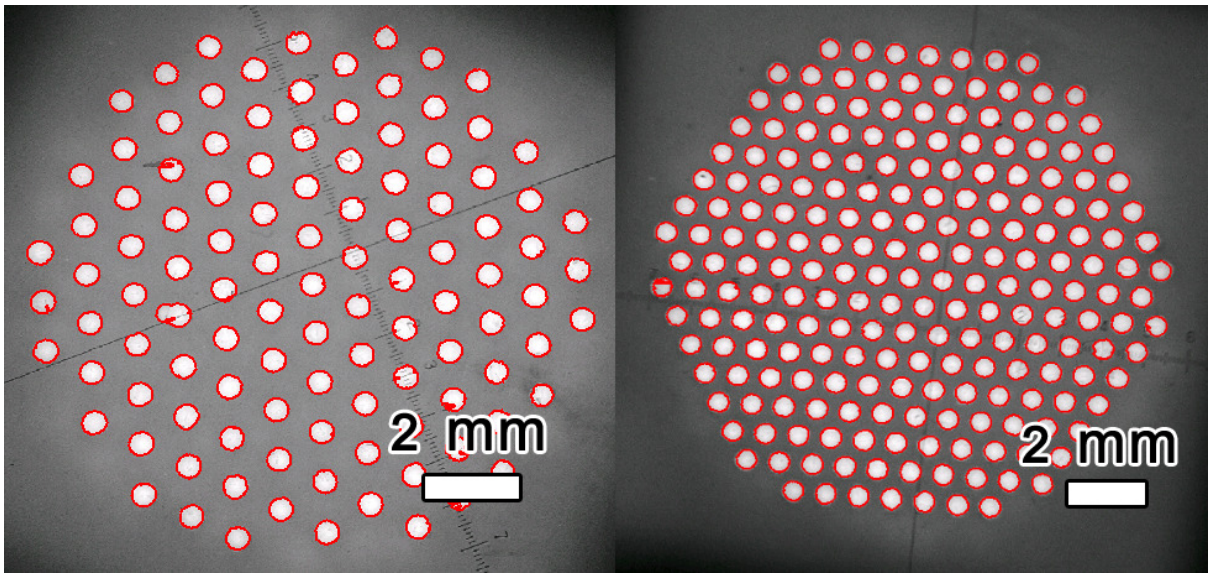

Figure 3.12: Two drill patterns with same hole diameter but different spacing. A MATLAB image processing script computes *a,s,* and a_{eff} as well as the total area and total effective area.

Experimentally, the samples were placed in a setup where an air pressure differential Δp was formed on the two faces of the sample. The resulting gas flow was collected and measured with time, thus the flow rate was obtained. Although the experimental details are numerous and omitted here for brevity, the results are tabulated in table (3.1) for three samples: drilled with close spacing, drilled with large spacing, and pierced with needles.

The results in table (3.1) shows permeability values calculated by equation (3.5), where k

Table 3.1: Experimental results testing the validity of the proposed model.

Sample	**Q [m^3/s]**	**Δp [Pa]**	**Area [m^2]**	**Effective area [m^2]**	**$k[m^2]$**	**Uncorrected $k[m^2]$**	**Error if not corrected**
Close spacing drilled	6.30E-7	3.10E+05	4.34E-05	6.31E-05	3.127E-16	4.55E-16	45.5%
Large spacing drilled	3.78E-7	3.10E+05	2.20E-05	3.77E-05	3.144E-16	5.37E-16	70.80%
Needle pierced	7.38E-7	6.89E+5	8.90E-06	2.57E-05	3.185E-16	9.54E-16	199.53%

is computed using the "effective area" for A in equation (3.5) as computed by the MATLAB code, while the "uncorrected k" is computed using the actual exposed area (removed by drilling or needles). Since the samples were prepared in the same batch, using the same processing conditions, the permeability should be approximately constant for all three samples - indeed, this is the case for the corrected case. However, in the uncorrected case, the permeability values differ significantly, with errors listed in the last column of table (3.1). Thus, the validity of the model is established by these experiments, and the importance of correcting the area A used in equation (3.5) is emphasized by the results.

3.12 Conclusion

When performing flow experiments on porous samples whose skins has been selectively machined either by cutting parallel rows or drilling/piercing hexagonal packed holes, the effective hole size is larger and therefore must be corrected using the proposed model. The discrepancy comes from the assumption in equation (3.5) that the flow is one dimensional, which is true for some boundary conditions but not for those representing a sample with selectively machined skins. However, if the total exposed area is computed using a_{eff} rather than a, equation (3.5) can be used to accurately compute the permeability of the material. The model also suggests that the correction becomes increasingly important for small hole sizes. This work established a methodology for applying this correction and validated it by experimental results with great success. This technique can now be used to conduct research on various open cell foam sheets produced with the solid-state microcellular foaming process.

Chapter 4

Permeability and Diffusivity Data

So far the mathematical formulas, algorithms, and equipment used to characterize the nanofoams have been established. In this chapter, the actual experimental results are presented and put in perspective by comparing them to other natural porous materials such as rocks. Both permeability and diffusivity were analyzed for a large number of PEI samples.

4.1 Experimental

The material used in this study came as 0.50 mm extruded sheets made from Ultem^{TM} 1000 PEI resin. Samples were die cut into 1" disks and saturated with 99.9% purity CO_2 for 48 hours. Immediately after removal from the CO_2 environment, the samples were heated between flat aluminum plates with 20 N clamping force for 30 seconds at temperatures of 170°C, 180°C, 190°C, and 200°C. This process was previously explored in detail by Miller et. al. [10] and used as a guide for sample preparation. For foaming temperatures below 170°C, the samples showed very little expansion so these temperatures were not investigated. Samples foamed above 200°C were excessively warped and/or blistered, making them unsuitable for investigation using the established experimental process. This phenomenon was observed and described by Miller [9] and Aher [17]. The solid skin was pierced on both faces simultaneously by arrays of hexagonally packed needles as previously described.

To correctly calculate the total area pierced by the needles, their average diameter, and spacing, an image processing code was written to compute the information from photographs of the samples. The samples were photographed using a lens containing a scale bar, with back illumination by a lightbox.

The information given by the image processing code was entered directly into the flow simulation code (shown in the appendix) mentioned previously, and an "effective area" was computed and correctly used in equation (3.5). The thickness of the core was digitally measured by fracturing the sample and photographing the cross-section.

The flow experiments were performed in a custom built apparatus based on the ASTM D6539-13 sample setup. The principle of operation is simple. On one side of the sample, a pressure regulator keeps the pressure at a specified constant value. The other side of the sample is exposed to the atmosphere. The gas which flows through the nonporous structure is captured using a tube and its volume is measured using a pipette (which is initially occupied by water). The rate is calculated by measuring the time required for 4 mL of gas to flow through the sample. This method is robust in that any leaks on the high pressure side do not compromise

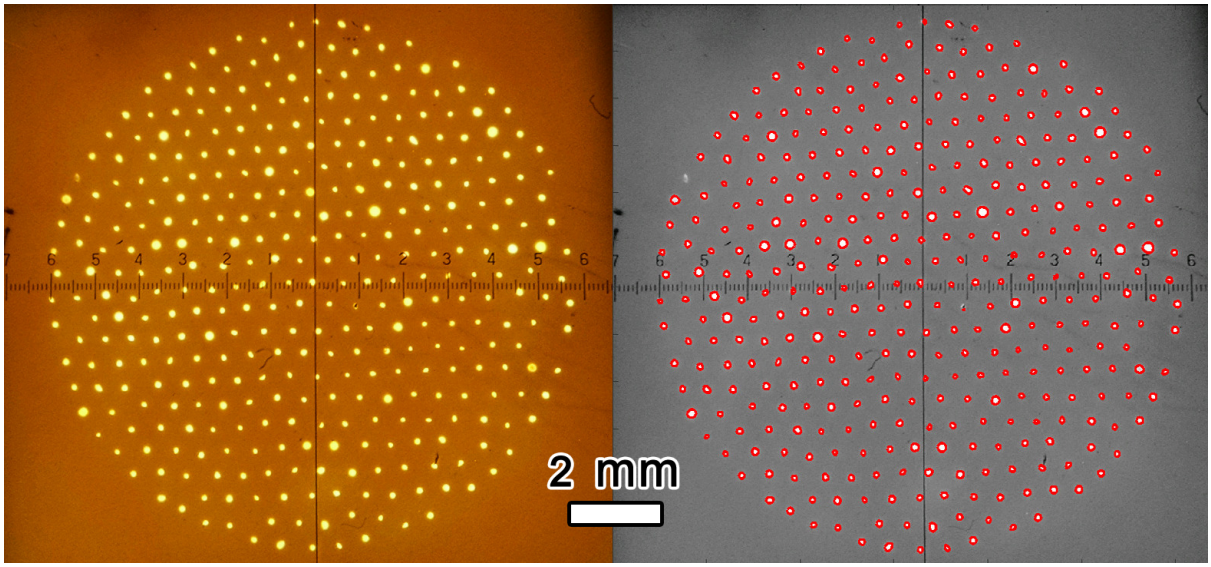

Figure 4.1: Left: Photograph of a pierced PEI sample. Right: Same sample with holes found by the image processing code.

accuracy, since that gas is not measured. The only important seal is on the low pressure side, which is easily sealed by a rubber O-ring.

4.2 Experimental Results

The results are summarized in table (4.1) and plotted in figure (4.3). Included is also the relative density Φ which is defined as the foam density divided by the density of PEI.

Table 4.1: Permeability results from experiments.

Foaming temp.	**Number of samples**	**Relative density**	**Average permeability [m^2]**	**Permeability standard deviation**	**St.dev / mean**
170°C	12	0.531	1.79 E-16	3.13 E-17	0.18
180°C	20	0.499	3.25 E-16	7.73 E-17	0.24
190°C	14	0.450	7.38 E-16	8.13 E-17	0.11
200°C	7	0.414	1.66 E-15	3.88 E-16	0.23

The equation for the curve fit gives permeability k and a function of temperature T.

$$k\,(T) = 10^{-14}(1.94\ \cdot 10^{-4}\,T^2 - 0.0669\,T + 5.78) \tag{4.1}$$

With units of meters squared and Celsius for k and T respectively. The residual sum of squares is $R^2 = 0.0998$ indicating very good agreement between data and model. To understand why this relationship is quadratic, a plot of relative density versus foaming temperature is shown in figure (4.4).

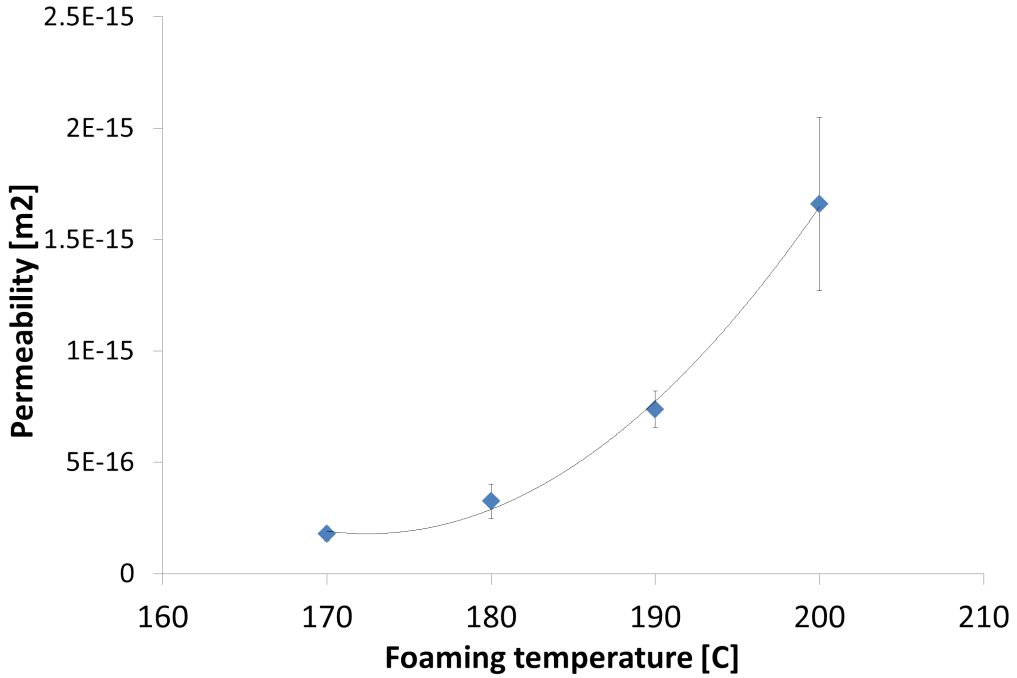

Figure 4.2: Plot of permeability vs. foaming temperature. The error bars are +/- 1 standard deviation. The curve fit is a polynomial of degree 2.

Consider a cross section of the nanoporous foam – the relative density is simply the fraction of void over solid polymer area. In a simple model used for porous media, the pores are treated as circular capillaries of diameter d [27]. The relative density is therefore proportional to the diameter squared, i.e.

$$\Phi \sim d^2 \tag{4.2}$$

Laminar fluid flow in a capillary is modeled by the Hagen-Poiseuille equation [18]

$$\Delta P = \frac{128 \mu L Q}{\pi d^4} \tag{4.3}$$

where L is the length. Therefore, for a fixed pressure drop, the flow rate is proportional to the fourth power of capillary diameter:

$$Q \sim d^4 \tag{4.4}$$

It then follows that

$$Q \sim \Phi^2 \tag{4.5}$$

and by equation (3.5)

$$k \sim Q \tag{4.6}$$

Therefore the final result is

$$k \sim \Phi^2 \tag{4.7}$$

which is precisely what the experimental data shows.

In asserting that $\Phi \sim d^2$, the number of capillaries was assumed constant. If more cells are nucleated with higher foaming temperature, the relative density can also decrease solely by an

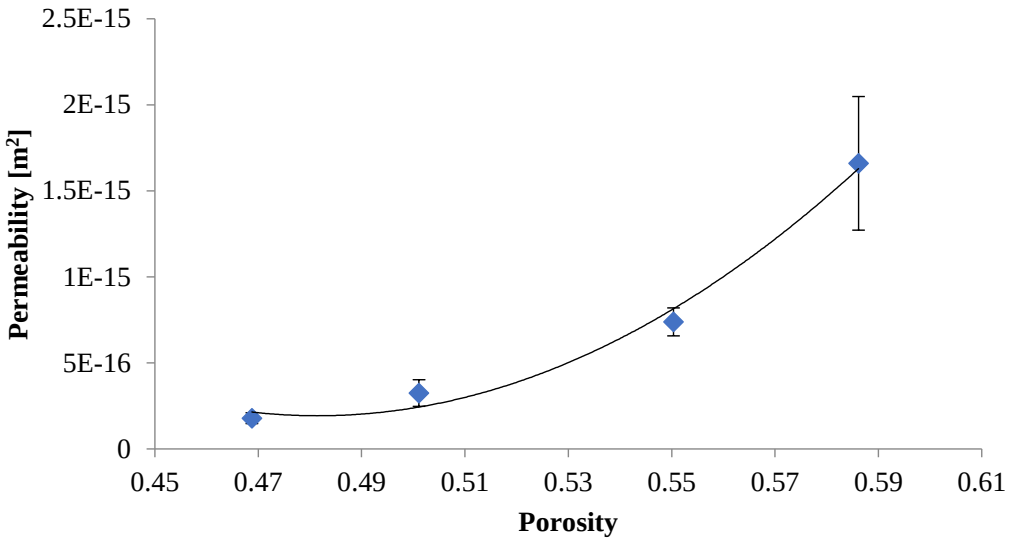

Figure 4.3: Plot of permeability vs. porosity (void fraction).

increase in the number density of voids. However, it has been previously shown that, in the solid-state foaming process, the cell nucleation density is not a strong function of temperature but of CO_2 concentration [1, 6, 11, 12]. Since all samples were saturated at 5.0 MPa and room temperature, the assumption holds.

4.2.1 Vapor Diffusion through the Nanostructure

A different kind of flow test was conducted, wherein there was no bulk fluid pressure gradient but rather a concentration gradient of different gas species. Three pure substances, water, isopropyl alcohol, and acetone, were placed into an aluminum canister atop which a PEI nanofoam sample was placed to seal off the enclosure from the ambient environment. This was done by the use of a rubber O-ring and a screw-on lid to apply the sealing force. Tests were performed using an unfoamed PEI sample and confirmed that there were no leaks. Figure (4.4) shows a diagram illustrating the principle of the experiment as well as the physical setup.

Since the liquid rests at the bottom of the canister, only the vapor can interact with the nanofoam. The gas present is a mixture of air and the substance in the vapor state. Since the total pressure is 1 atmosphere which is well above the vapor pressure of the substances at the temperature at which the experiments were performed (room temperature), the system can be considered as a mixture of ideal gasses [28, 29]. Therefore, the partial pressure of the substance in the gas mix is simply its saturation temperature at 1 atmosphere.

4.2.2 Diffusion model

Since there is no pressure difference across the sample in the bulk fluid, the substance exits the canister through the nanofoam at the molecular level by diffusion. The thermodynamic driving force is rather a concentration gradient. Fick's law of diffusion models the diffusion flux $\boldsymbol{J}$ as proportional to the concentration gradient [30]

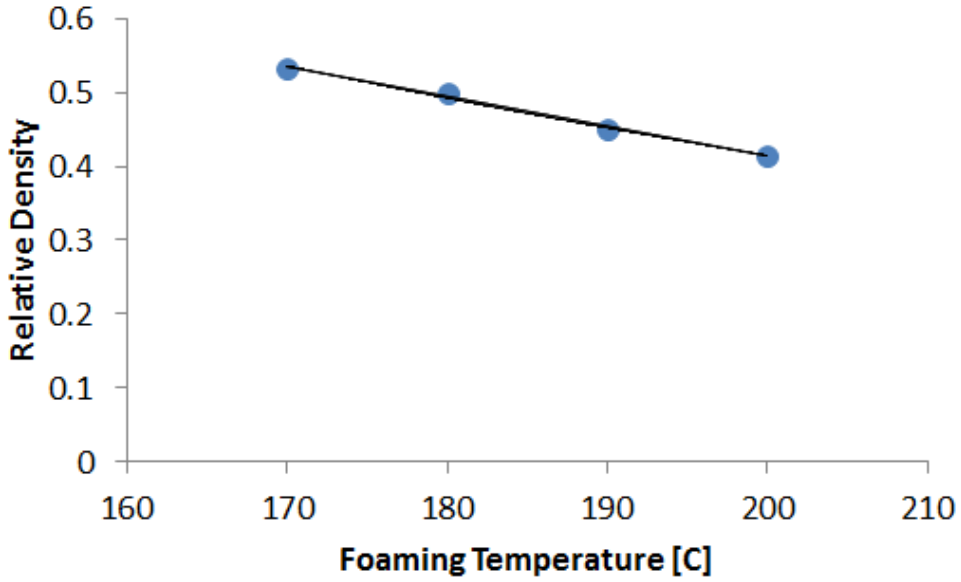

Figure 4.4: Plot of relative density versus foaming temperature showing a linear relationship.

$$\boldsymbol{J} = -D_p \nabla C, \;\; |\boldsymbol{J}| = -D_p \frac{\partial C}{\partial x} \tag{4.8}$$

where D_p is the proportionality factor referred to as the diffusion coefficient in the porous medium. Since the macroscopic molecular transport is normal to the sample face, the flux is a scalar satisfying

$$J = |\boldsymbol{J}| = -D_p \frac{\partial C}{\partial x} \tag{4.9}$$

Here, J is the molar flux with units $[mol \;/\; m^2 \cdot s]$ and C is the molar concentration in units $[mol \;/\; m^3]$. Indeed the diffusion inside the sample is not 1D, however this will be corrected using the model and computer code in the same fashion as was done for permeability. Since the diffusion equation (4.8) and fluid equation (3.1) are partial differential equations of the same mathematical form, it may be expected that the diffusion coefficient in the porous medium D_p will be a quadratic function of T as is $k(T)$. Indeed this will be shown to be the case.

To compute J, the canisters were weighted periodically using a digital lab scale with accuracy of 10 μg. Figure (4.5) shows a plot of mass escaped versus time (hours) showing a very slow but constant release rate. Since the vapor pressure is a function of temperature only and the sole driving force, the escape rate was constant regardless of the amount of liquid present in the canister.

This evaporation rate f [g/sec] was divided by the *effective area* A_{eff} of exposed core, the same value as computed for the permeability calculations. The concentration is assumed zero on the face exposed to ambient air. Therefore, the diffusion coefficient is given by

$$D_p = \frac{fhRT}{MA_{eff}p_v} \tag{4.10}$$

where R is the universal gas constant, M the molar mass of the substance, and p_v its vapor pressure at ambient temperature. Table 2 shows the diffusion coefficient computed for each

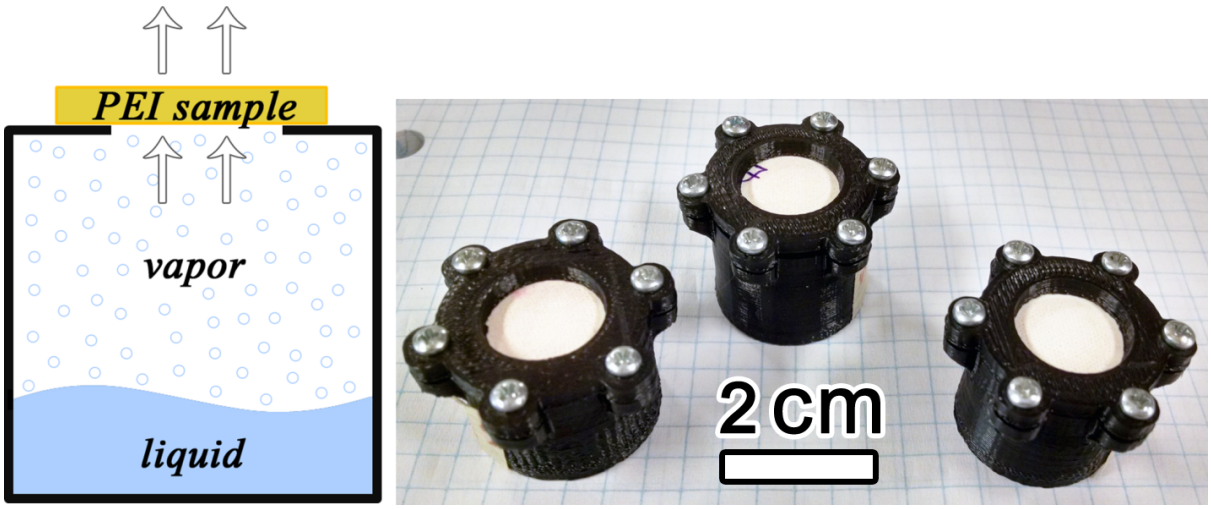

Figure 4.5: Vapor diffusivity test apparatus diagram and photograph.

substance with samples at each foaming temperature. The same sample was used for all 3 substances to eliminate compounded uncertainty.

Table 4.2: Diffusion coefficient computed from evaporation tests. The units are in mm^2/s.

Foaming temp:	**Water**	**Isopropyl**	**Acetone**
170 C	0.285	0.234	0.320
180 C	0.303	0.270	0.349
190 C	0.391	0.399	0.362
200 C	0.582	0.520	0.643

A plot of the results in table (4.2) shows that the diffusion coefficient is similar to all three substances moving through the nanostructure, and has a quadratic relationship to foaming temperature as does the permeability.

The function that fits the data in figure (4.6) is

$$D_p(T) = 0.0005\,T^2 - 0.1472\,T + 13.047 \tag{4.11}$$

The residual sum of squares for this fit is 0.9954, again indicating strong agreement between data and model.

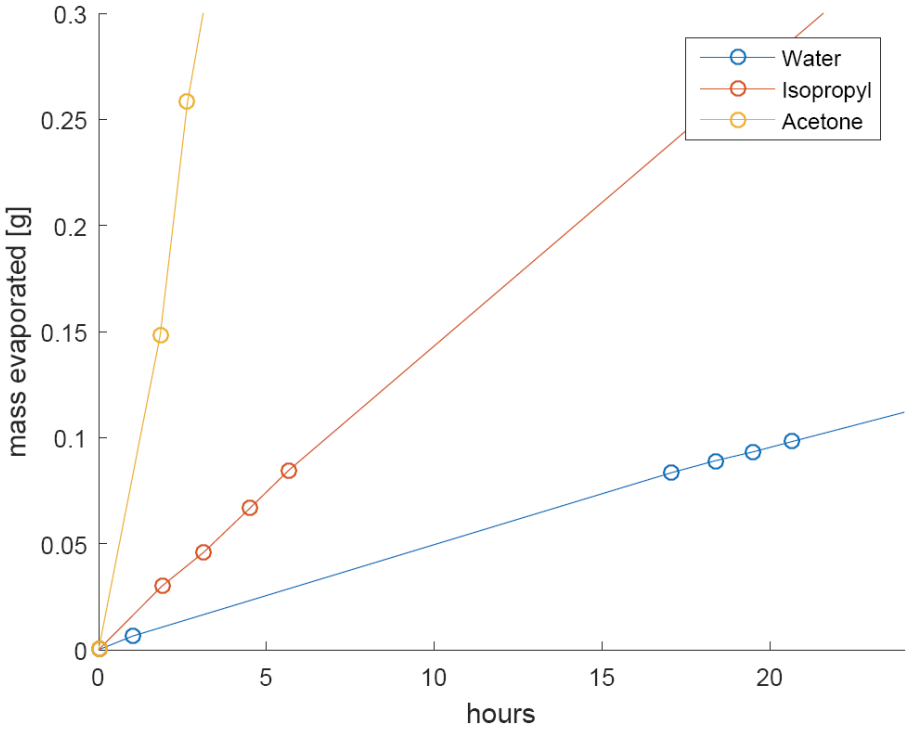

Figure 4.6: Plot of mass escaped versus time for the three liquids through a PEI sample foamed at 180°C. The escape rate of the substance was constant for all time.

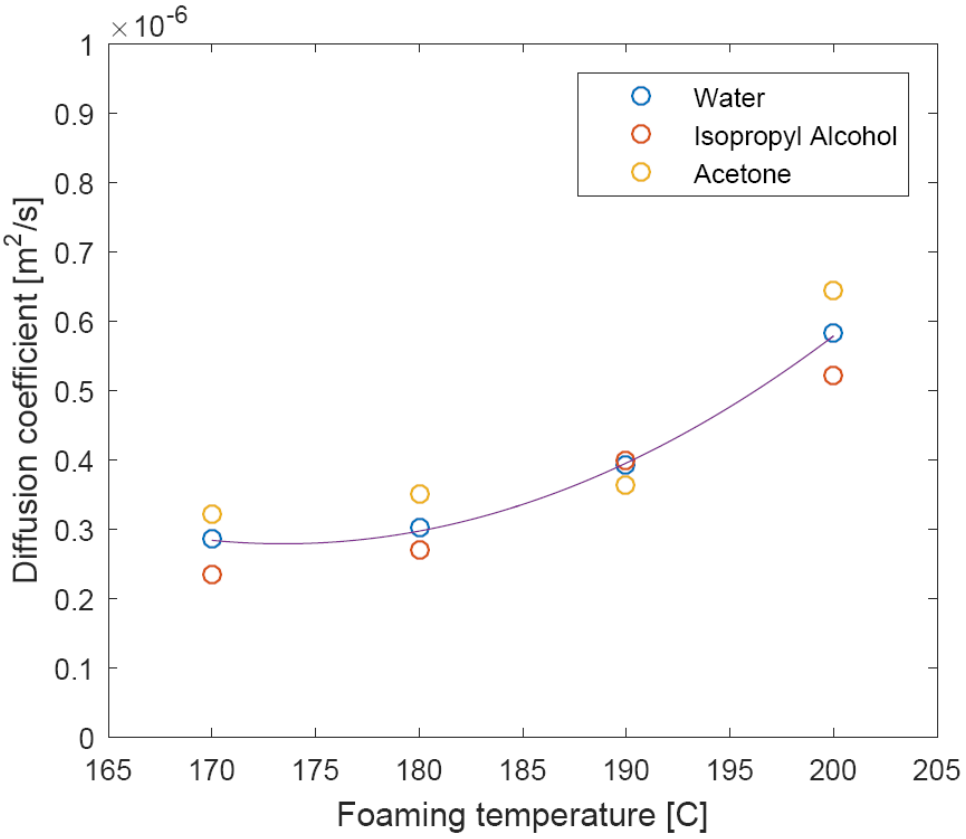

Figure 4.7: Plot of diffusion coefficient for 3 substances for 4 samples at different foaming temperatures. A quadratic curve fits the data.

4.3 Propagation of Uncertainty

It is useful to know the uncertainty in the values calculated above. This is done by assuming the measurements are random variables drawn from a distribution with variance σ^2. By propagating the variance in each measurement, we can compute the variance of permeability. Assuming the fractional standard deviation is small, the variances add in the following way [31]:

$$\sigma_k^2 = k^2 \left(\left(\frac{\sigma_\mu}{\mu}\right)^2 + \left(\frac{\sigma_Q}{Q}\right)^2 + \left(\frac{\sigma_h}{h}\right)^2 + \left(\frac{\sigma_{\Delta p}}{\Delta p}\right)^2 + \left(\frac{\sigma_{A_e}}{A_e}\right)^2 \right) \tag{4.12}$$

The last term, $\sigma_{A_e}^2$ is the variance in the effective area A_e. This can be computed from the function of effective pore size $a_e = f(a, s)$ as follows:

$$\sigma_{A_e}^2 = \frac{n\pi h^2 f}{2} \left(\left(\frac{\partial f}{\partial a}\right)^2 \sigma_a^2 + \left(\frac{\partial f}{\partial s}\right)^2 \sigma_s^2 + 2\frac{\partial f}{\partial a}\frac{\partial f}{\partial s}\sigma_{as} \right) \tag{4.13}$$

The partial derivatives of $f(a, s)$ are estimated numerically for the values of a, s in the needle-pierced specimens to be $\partial f/\partial a = 1.090$ and $\partial f/\partial s = 0.1146$. Table 4.3 shows the numerical data from the algorithm used for this computation:

Table 4.3: Computing the partial derivatives of effective pore size a_{eff} for use in uncertainty propagation.

a	s	a_{eff}
0.326	0.960	0.6116
0.336	0.960	0.6225
0.326	1.008	0.6171

Uncertainties in other measurements can be estimated as follows: for a, the variance in the histogram from the image processing script is used - 0.03 mm. Since h here is 0.49 mm, we have $(\sigma_a/a)^2 = (0.05/0.49)^2 = .0104$. From data in section 2.9, $(\sigma_h/h)^2 = 0.00157$. Uncertainty in Q is due to human error in timing the liquid, and was found to be 300 milliseconds in a 4 second interval - therefore $(\sigma_Q/Q)^2 = 0.00563$. For Δp, the uncertainty comes from the sensor which is 0.25 psi out of 50. Lastly, the uncertainty in effective area is computed using eq.(4.13). The covariance between a and s is small since the uncertainty comes from measurement error which is independent of the other variable. Also, the partial derivative with respect to s is very small, so that term is ignored for simplicity since it is negligible. Therefore, $\sigma_{A_e}/A_e = \sigma_a/a$. Putting everything together,

$$\frac{\sigma_k}{k} = \sqrt{\left(\frac{\sigma_a}{a}\right)^2 + \left(\frac{\sigma_Q}{Q}\right)^2 + \left(\frac{\sigma_h}{h}\right)^2 + \left(\frac{\sigma_{\Delta p}}{\Delta p}\right)^2} \tag{4.14}$$

Using the numbers above, this evaluates to **0.133**, or 13.3%. This is the uncertainty in any single permeability measurement obtained using my experimental setup.

4.4 Discussion

The permeability values obtained in this work can be put into perspective by comparing it to naturally porous solids. Figure (4.8) shows the permeability ranges of various rocks given by Brace [32]. The nanoporous PEI samples investigated here have a permeability value comparable to that of volcanic rock and limestone and two to four orders of magnitude lower than that of sand.

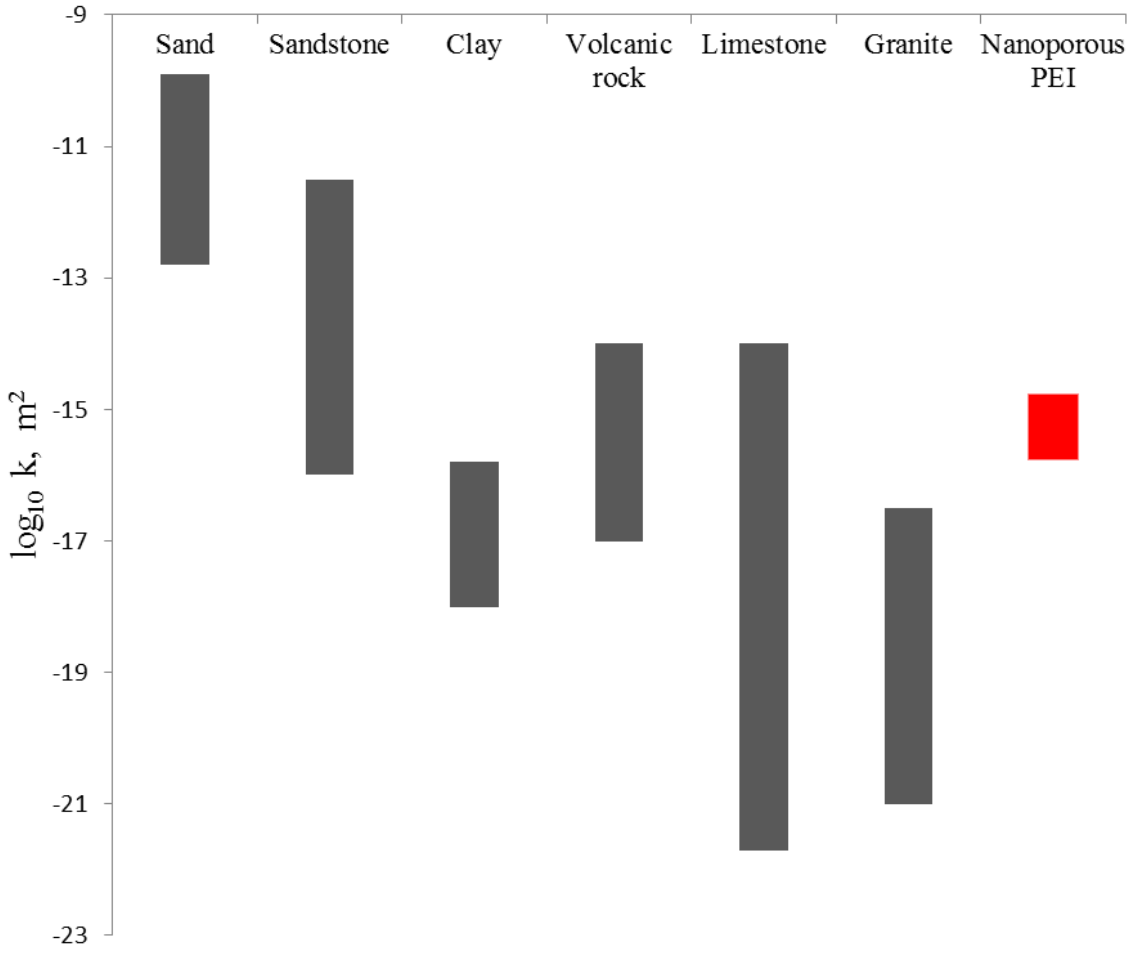

Figure 4.8: Chart showing the permeability ranges of various solids including the nanoporous PEI samples investigated in this work.

The diffusion coefficient through porous D_p media is modeled by [33]

$$D_p = c\phi D_0/\tau \tag{4.15}$$

where $\tau > 1$ is the tortuosity, ϕ is the porosity, D_0 is the diffusion coefficient in the fluid occupying the pores, and c is a constrictivity factor which can be approximated as 1 since the pore sizes are above 1 nm [34]. The diffusion coefficient of a dilute ideal gas diffusing into a second ideal gas is a function of temperature T, the diffusing molecule's transport cross-section σ, the pressure p, and the molecular mass of the solvent gas m_1 by the relation [35]

$$D_0 = T^{\frac{3}{2}}/\sigma p\,\sqrt{m_1} \tag{4.16}$$

Since the diffusion coefficient through air D_0 of water, isopropyl alcohol, and acetone differs but is on the same order of magnitude, the diffusion coefficient through the porous medium D_p also differs, a fact shown by the data.

The ratio D_p/D_0 is called the *diffusivity* and by equation (4.15) is dependent on the porous structure only. This fact can be used to find the *tortuosity* of the material as defined in the above equation. For water vapor through excess air, the diffusion coefficient at 293.15 K is $D_0 = 2.42 \cdot 10^{-5}\ \mathrm{m}^2/\mathrm{s}$ [36, 37]. Table 3 shows the diffusivity and tortuosity of the PEI nanoporous samples in this study.

Table 4.4: Structure properties of PEI nanofoams as computed from diffusion data.

Foaming Temp	$\boldsymbol{D_p/D_0}$	**Porosity** $\boldsymbol{\phi}$	**Tortuosity** $\boldsymbol{\tau}$
170°C	0.0118	0.469	39.8
180°C	0.0125	0.501	40.1
190°C	0.0162	0.550	34.0
200°C	0.0240	0.586	24.4

Figure (4.9) compares the diffusivity of nanoporous PEI with literature values of various materials [33, 38, 39].

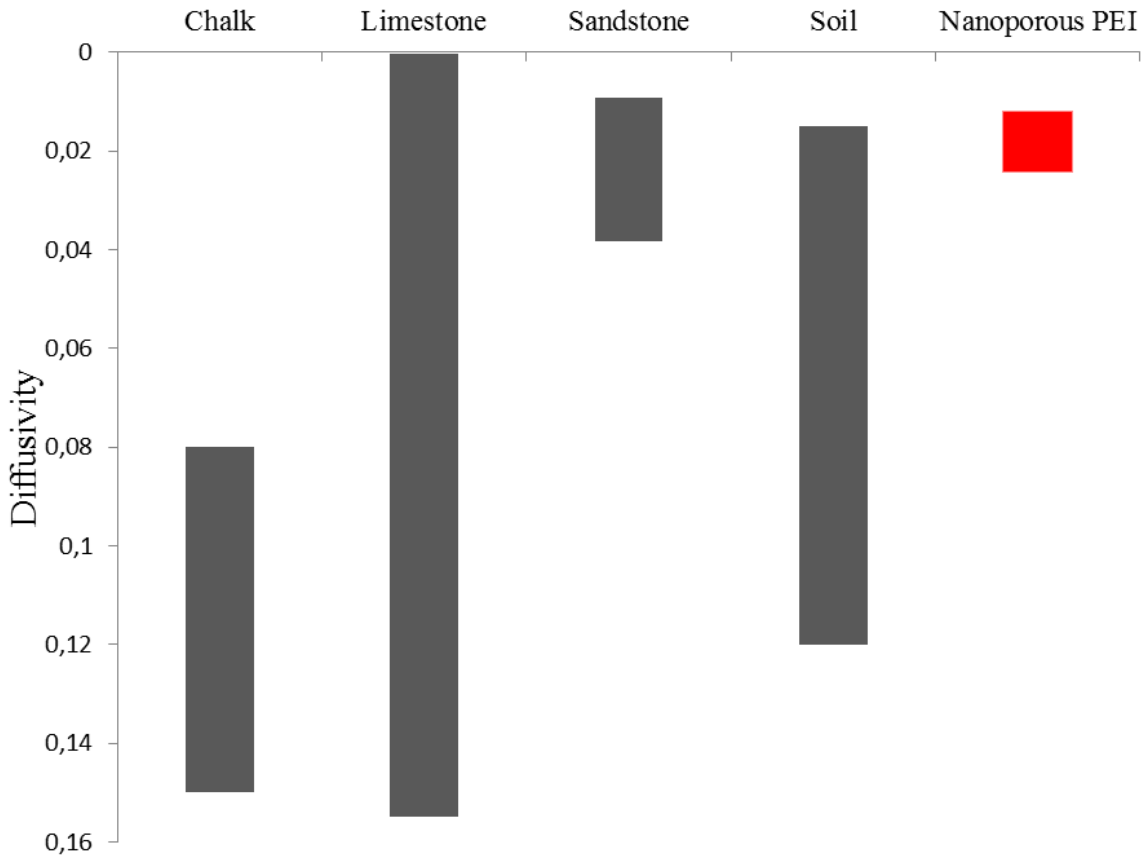

Figure 4.9: Comparison of diffusivities between nanoporous PEI and other natural materials.

4.5 Remarks

The permeability, diffusivity, and tortuosity are properties of porous materials which were unknown for nanoporous polyetherimide created with the solid-state foaming process. In this work, all three of these properties were experimentally quantified for PEI samples saturated with CO_2 at 5.0 MPa and foamed at 170°C, 180°C, 190°C, and 200°C. A quadratic dependence of both permeability and diffusivity on foaming temperature was found and correlated to the linear dependence of porosity on temperature. Both permeability and diffusivity are comparable to that of sandstone, showing that mass transport through the structure is slow due to the small pore sizes of the nanofoam.

Chapter 5

Transparent Foams

5.1 Transparent PEI Foams

In November 2016, I made an interesting accidental discovery: while experimenting with increased CO_2 concentrations in PEI via a low temperature absorption process, I created a foam material that is partially transparent. PEI samples of 0.5 mm thickness (as before) were saturated with CO_2 at 5.0 MPa and -30°C and foamed at temperatures between 130°C and 170°C. This resulted in foams that were of lower density than before, and optically transparent enough to read text through it, as shown in figure (5.1).

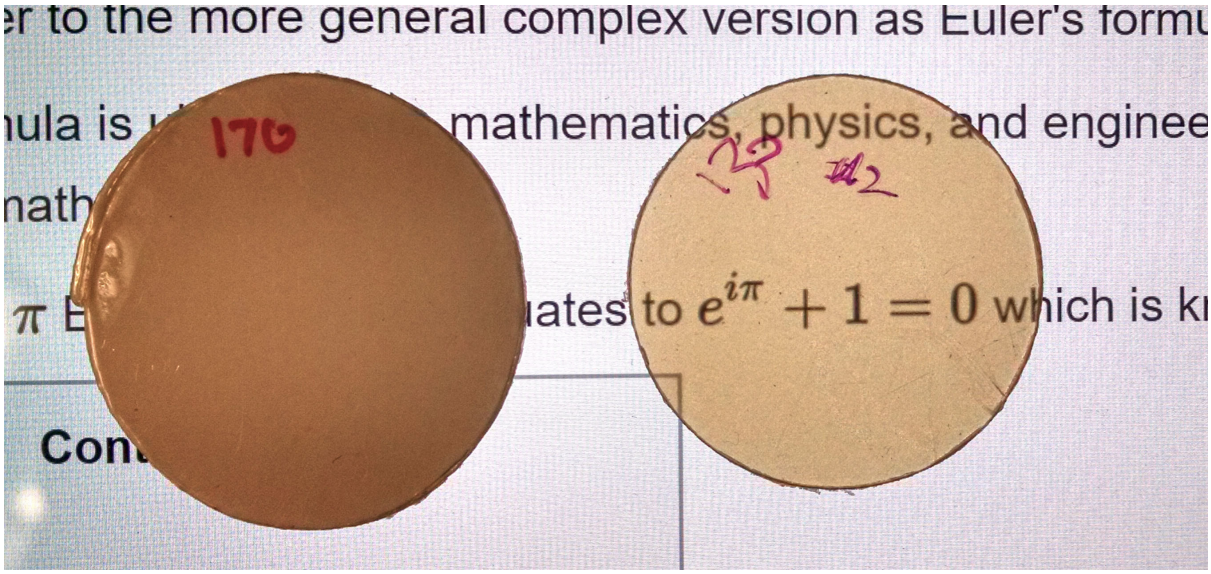

Figure 5.1: Left: Opaque PEI foam sample saturated at room temperature 5.0 MPa and foamed at 170°C. The relative density is approximately 0.55. Right: Semi-transparent PEI foam sample saturated at -30°C and foamed at 130°C. The relative density of this sample is 0.41.

SEM micrographs were very difficult to obtain for two reasons. First, the sputter coating necessary to image these non-conductive samples would block the pores, making them invisible. For this reason, we switched the sputter coating material from the usual Gold/Palladium (grain size 5-10 nm) alloy to pure Platinum (grain size 1 nm). The sputter time was also reduced from the usual 60 seconds to 10 seconds, in order to have fewer metal particles on the

surface. Charging was reduced by reducing the SEM spot size (from 3 to 2) and accelerating voltage from 5kV to 3kV. Second, the pores were so small (less than 10 nm) that the required imaging magnification pushed the SEM equipment to its limits, and focusing became difficult. The high magnification combined with some charging often produced blurry images where no pores would be visible. Nonetheless, with help from the Molecular Analysis Facility staff, I was able to image the pores - figure (5.2).

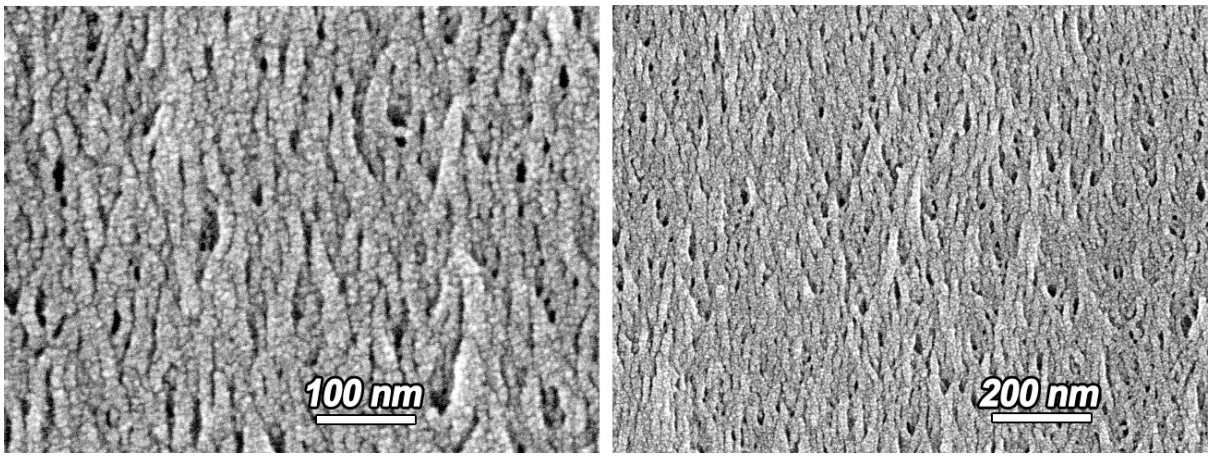

Figure 5.2: SEM images of a transparent PEI sample saturated at 5.0 MPa, -30C and foamed at 160C.

The images do not reveal details of the nanostructure, especially void size distribution. Most notably, the porosity of these foams were measured to be over 0.5, however the SEM images do not reflect this. Other imaging techniques need to be explored, such as Scanning Helium Ion Microscopy, Transmission Electron Microscopy, or Atomic Force Microscopy.

5.2 Permeability

The next question was whether the nanostructure in these samples is open-porous. This was confirmed with the acetone/dye penetration test as before, so the same permeability and diffusivity tests were conducted on this transparent PEI foam in the exact manner as the opaque samples.

The line that best fits the relative density data in figure (5.4) is

$$\frac{\rho_{foam}}{\rho_{PEI}} = 0.896 - 0.0034T \tag{5.1}$$

where T is the foaming temperature in degrees Celsius. The residual for this fit is $R^2 = 0.9985$, showing strong linearity.

Table 5.1: Permeability and density values of the transparent PEI material saturated at -30°C and 5.0 MPa.

Foaming temp:	**Relative Density** ρ_{foam}/ρ_{PEI}	**Porosity** ϕ	**Permeability** $[m^2]$	**Permeability St.dev** $[m^2]$
130 C	0.45	0.55	0	N/A
140 C	0.41	0.59	1.33E-15	2.94E-16
150 C	0.38	0.62	1.03E-16	1.69E-17
160 C	0.35	0.65	2.41E-16	7.47E-17
170 C	0.36	0.64	2.16E-16	7.70E-17

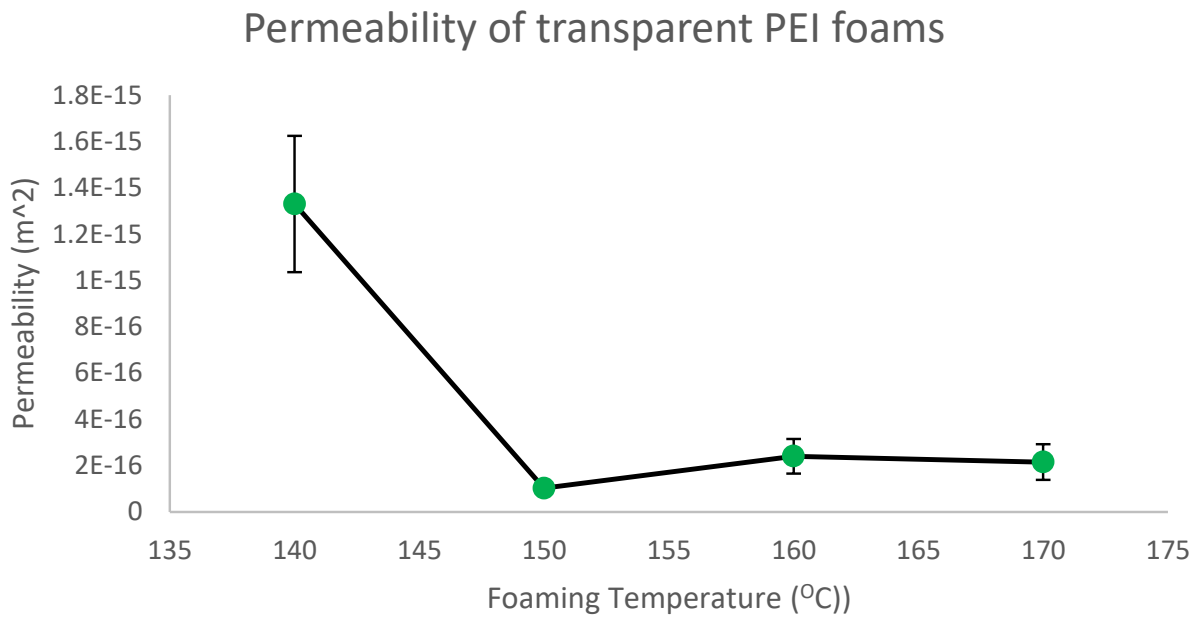

Figure 5.3: Plot of permeability versus foaming temperature for transparent PEI samples. Note that the permeability values are approximately an order of magnitude lower than the regular, opaque samples.

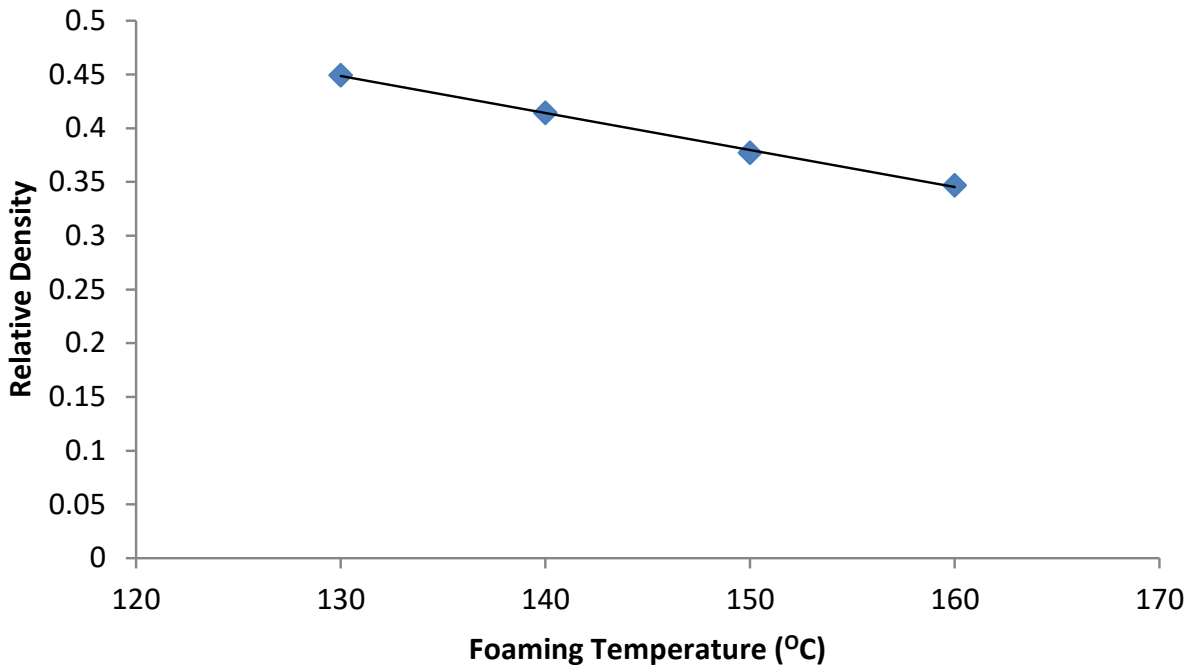

Figure 5.4: Plot of relative density ρ_{foam}/ρ_{PEI} versus foaming temperature for transparent PEI samples. The relation is linear in the 130 - 160°C range.

5.3 Diffusivity

The diffusivity of the transparent PEI nanofoams was computed in the exact same manner as previously. Only water was used, since the diffusion coefficient of water through excess air is known (again, it is 2.45E-5 m^2/s) and thus the diffusivity and tortuosity could be computed. Table (5.2) summarizes the data.

Table 5.2: Diffusion coefficient of water through the nanofoam; diffusivity, and tortuosity of transparent PEI nanofoams.

Foaming temp:	**Diffusion Coefficient** mm^2/s	**Diffusivity**	**Porosity** ϕ	**Tortuosity**
130 C	0.043	1.77E-03	0.55	311
140 C	0.100	4.13E-03	0.59	142
150 C	0.214	8.84E-03	0.62	70
160 C	0.300	1.24E-02	0.65	53
170 C	0.385	1.59E-02	0.64	40

Plots of the data are shown in figures (5.5) and (5.6).

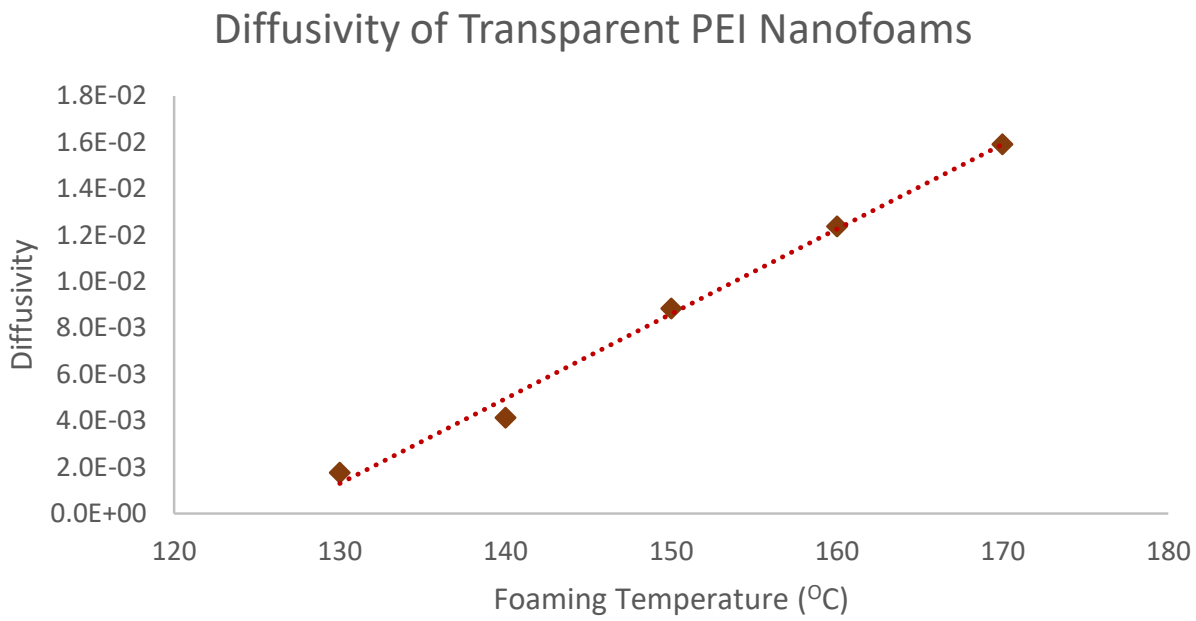

Figure 5.5: Diffusivity (dimensionless) of transparent PEI nanofoams. The equation of the line fit is $y = 0.0004x - 0.0462$ and $R^2 = 0.9928$.

The relation between diffusivity and foaming temperature is linear, whereas in the opaque PEI samples it was quadratic. This means that the structure is fundamentally different, since the quadratic relation was expected due to both structures' linear function mapping porosity to

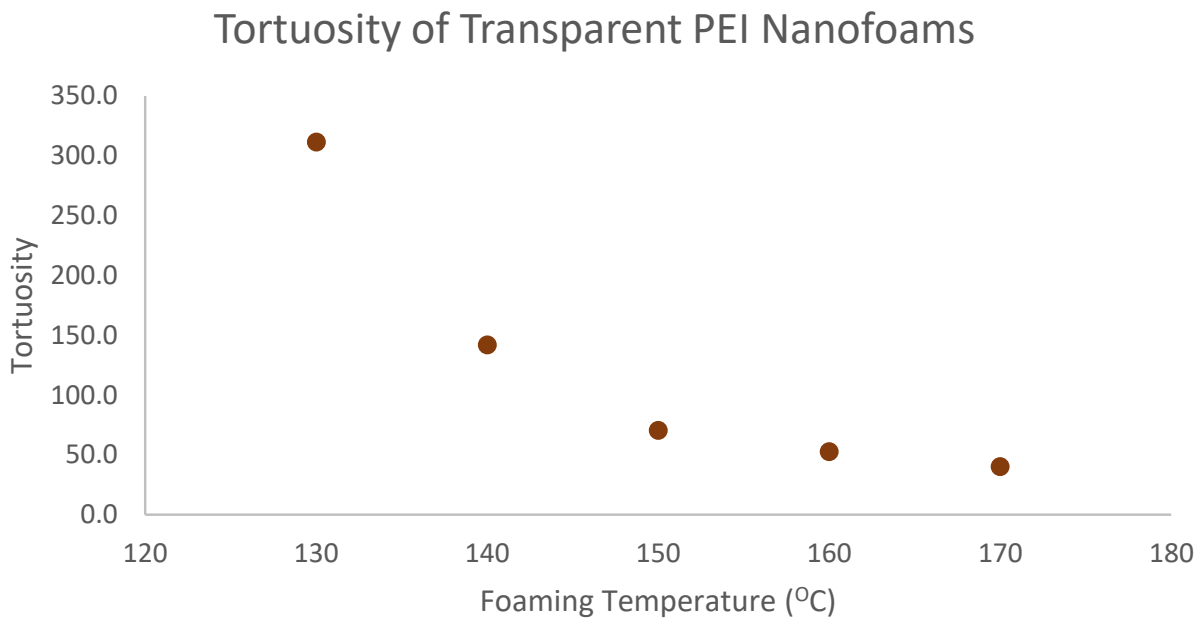

Figure 5.6: Tortuosity (dimensionless) of transparent PEI nanofoams.

foaming temperature. However, the relation between tortuosity and foaming temperature is of negative derivative $\forall T \in [130, 170]$ which is not the case for the opaque samples. We shall see that such discrepancies again appear in thermal conductivity measurements. This suggests that transport through these nanostructures is fundamentally different, perhaps due to the cell size approaching the molecular scale.

5.4 Light Transmittance

Light transmittance through these samples was quantified using a Varian Cary 5000 UV-Vis-NIR apparatus. Light of wavelengths from 400 to 1200 nm were passed through the sample, and the transmission intensity was measured by the device. Samples of 0.50 mm and 0.15 mm were used. Of course, the thinner samples are much more optically clear. The transmission isn't 100% even for the raw material, since PEI has a yellow tint. However, comparing the transmission of foams to that of raw PEI makes the natural tint immaterial.

The following plots show the UV-Vis-NIR results presented in two ways: 1) as a transmission coefficient T and 2) relative to raw PEI (T_{foam}/T_{PEI}). Note that in this section, T refers to a transmission coefficient (dimensionless) and not temperature.

The data in figures (5.7, 5.9) have a jump when the wavelength is 800 nm. This is due to the apparatus changing detectors to an infrared detector, and the calibration being slightly different than for the UV-Visible detector. However, normalizing the data with respect to the raw PEI sample eliminated this effect.

From figures (5.8, 5.10) it is noticeable that the transmission through the foam samples is, for some wavelengths, higher than through raw PEI (for foaming temperatures other than 170°C). To see whether this is true on average for the entire tested spectrum, the average trans-

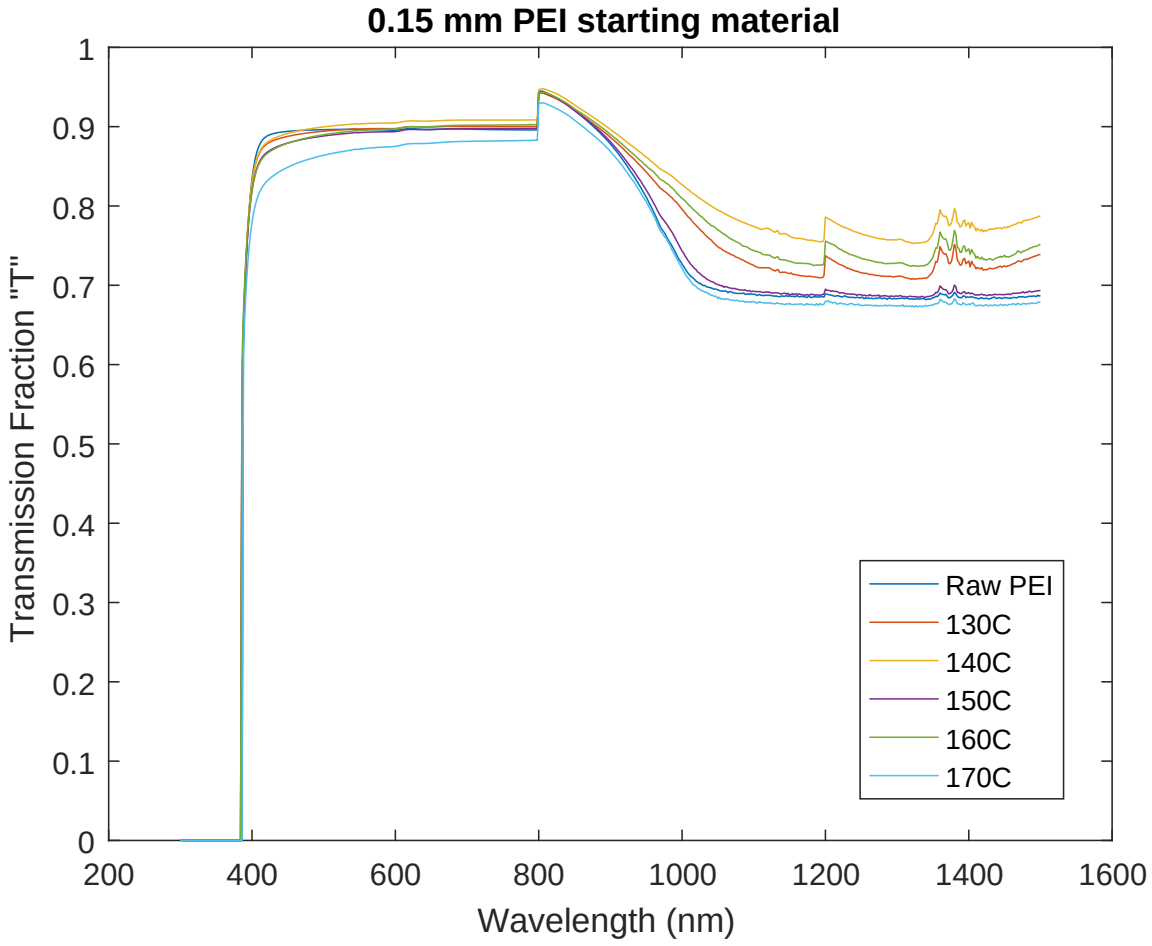

Figure 5.7: Transmission coefficient as function of wavelength for transparent PEI samples of starting thickness 0.15 mm saturated with CO_2 at 5.0 MPa, -30°C and foamed at various temperatures.

mission ratio was computed as

$$\overline{\left(\frac{T}{T_{raw}}\right)} = \frac{\bar{T}}{\overline{T_{raw}}} \tag{5.2}$$

The results are tabulated in table (5.3).

Table (5.3) shows that the light transmission, on average, is indeed higher through the foam samples than through the raw PEI material for all foaming temperatures except 170°C. This is unexpected for two reasons: 1) the cells cause a refractive-index discontinuity and would normally scatter light, causing internal reflection and thus increased absorbance and 2) the foamed samples are thicker after foaming. However, the material expands in all directions during the foaming process, meaning that the yellow pigment present in the raw polymer is less concentrated after foaming, possibly explaining the reduction in absorbance.

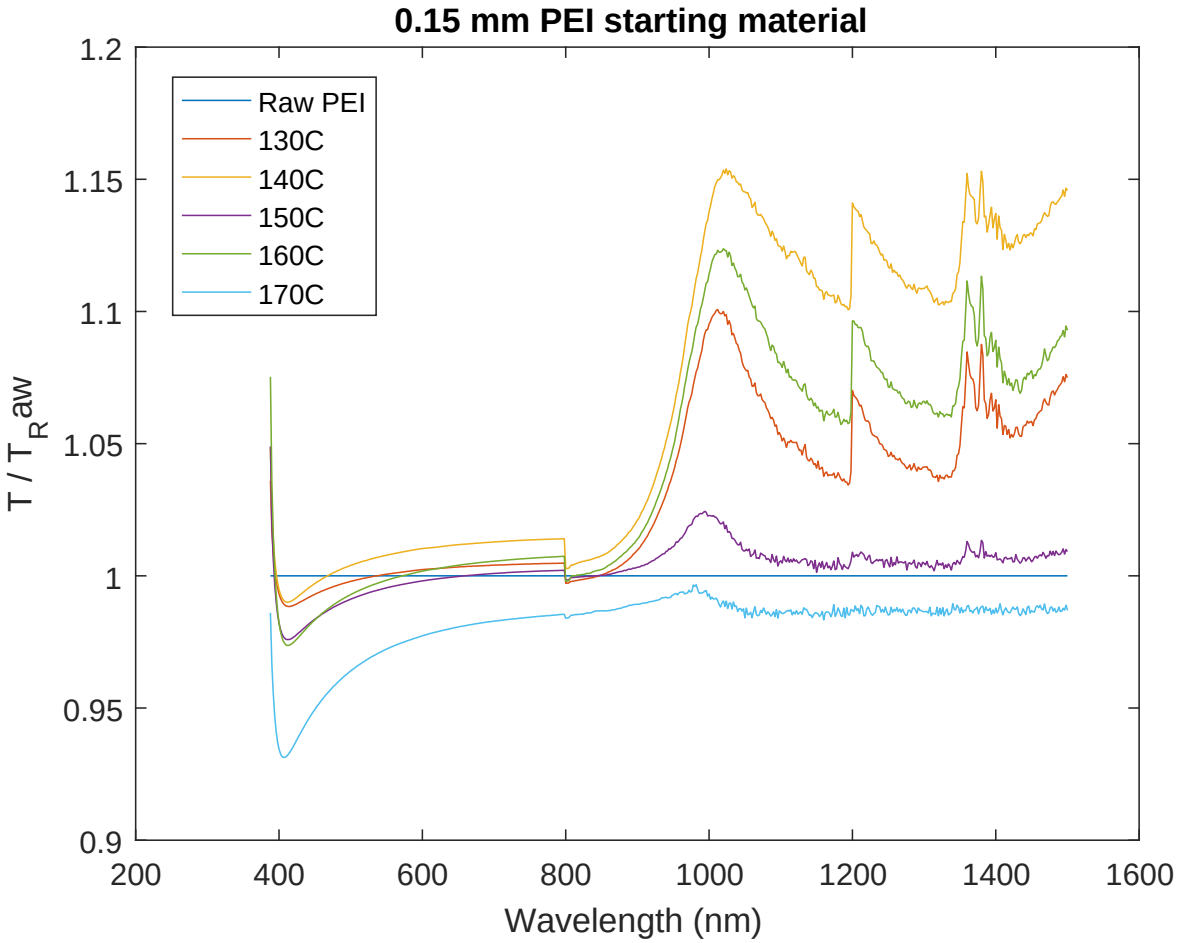

Figure 5.8: Same data as figure (5.7) but the transmission value at each point was divided by that of raw PEI at the same wavelength.

Table 5.3: Normalized light transmission averaged for wavelengths between 400 and 1200 nm.

Foaming Temp $(^{\circ}C)$**:**	T/T_{raw} **(0.50 mm):**	T/T_{raw} **(0.15 mm):**
130	1.1057	1.0281
140	1.0107	1.0606
150	1.0024	1.0033
160	1.0134	1.0378
170	0.9806	0.9806

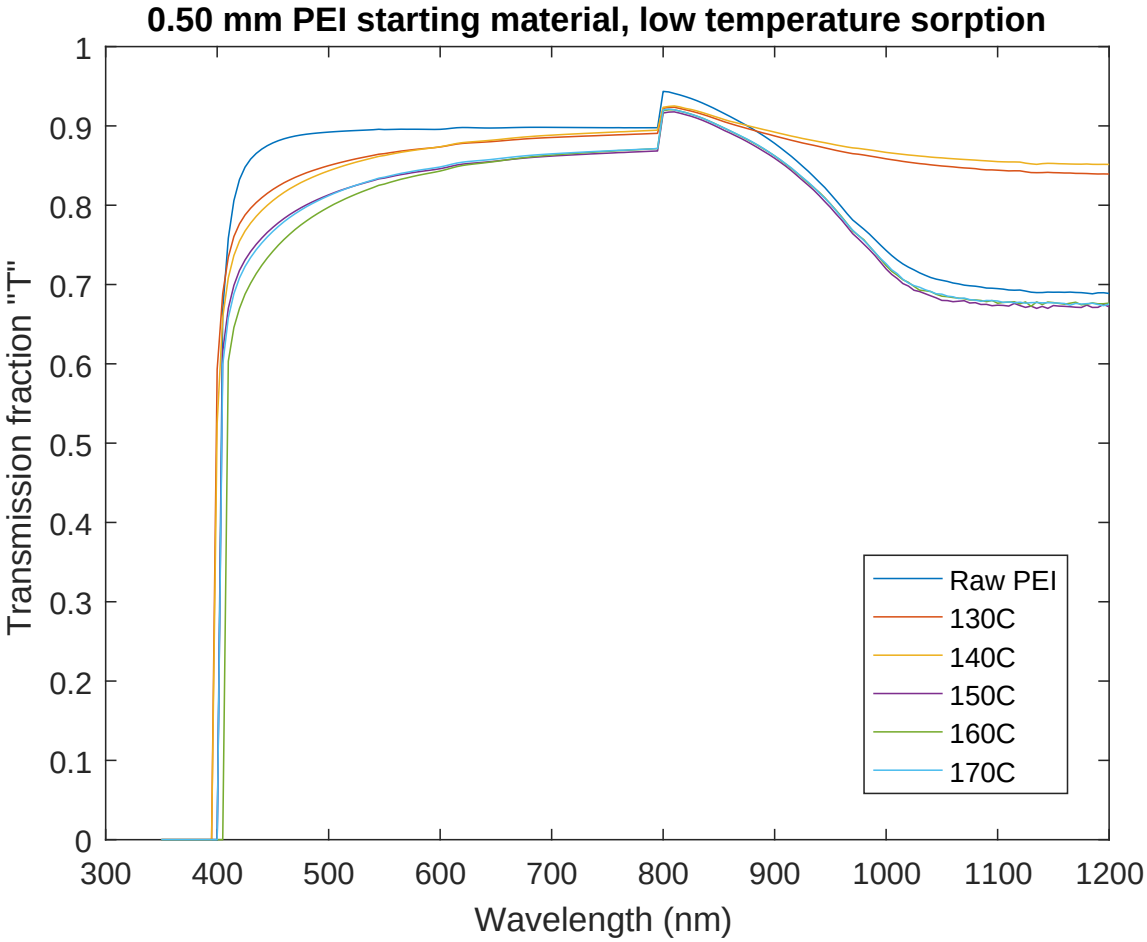

Figure 5.9: Transmission coefficient as function of wavelength for transparent PEI samples of starting thickness 0.15 mm saturated with CO_2 at 5.0 MPa, -30°C and foamed at various temperatures.

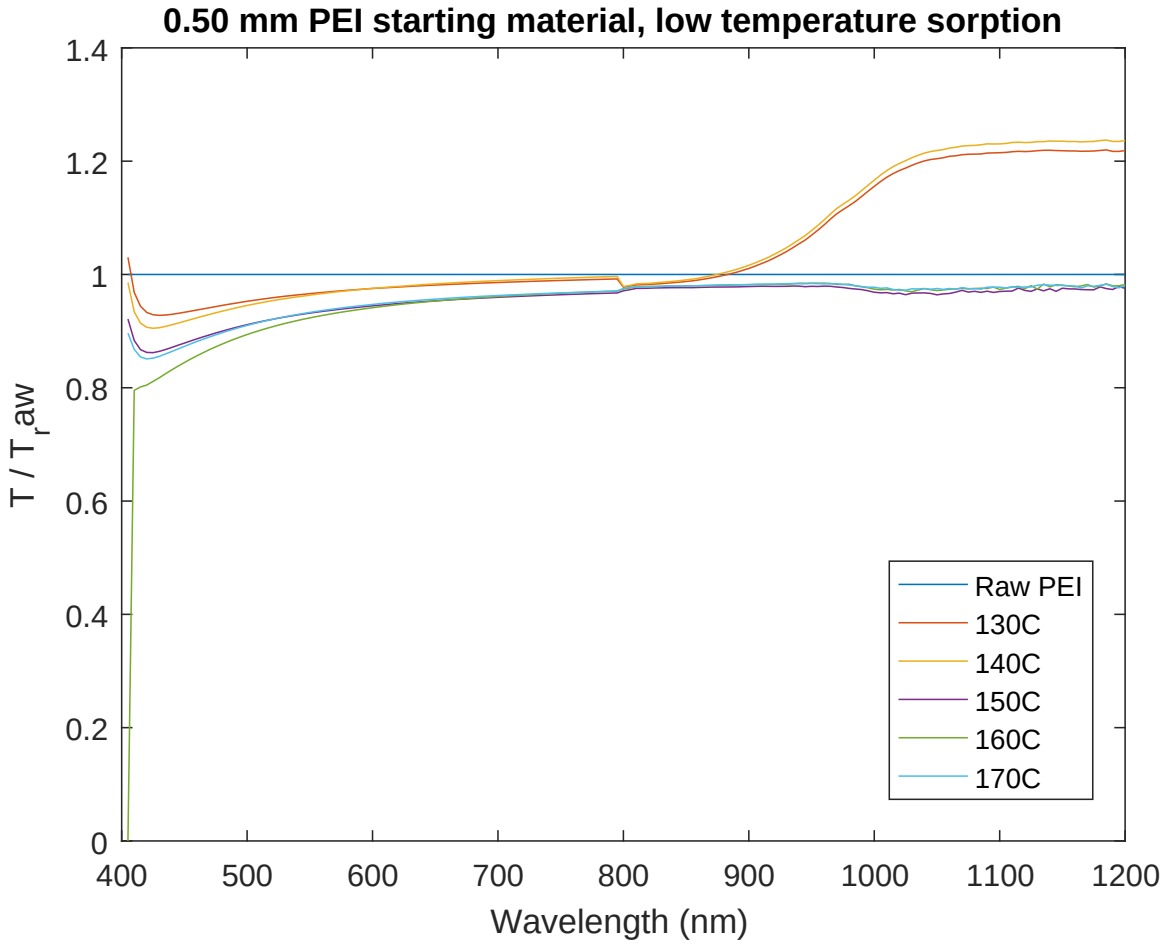

Figure 5.10: Same data as figure (5.9) but the transmission value at each point was divided by that of raw PEI at the same wavelength.

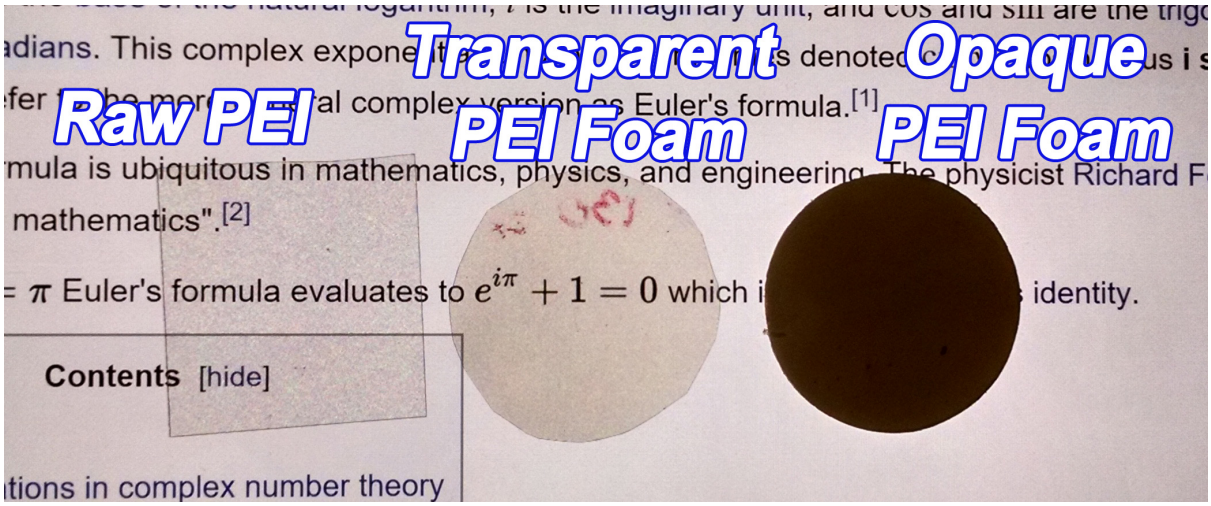

Figure 5.11: PEI sheets of 0.15 mm starting thickness processed in the same way as the samples above. The transparency is much better.

Chapter 6

Thermal Conductivity

6.1 Introduction

At the PhD general examination, the committee suggested that I measure the thermal conductivity of my nanoporous PEI material for two reasons: 1) to have the data available for commercialization efforts and 2) to test if the Knudsen effect is measurable in the pores' gas phase thermal conduction. Commercial products for measuring conductivity were quoted at $30,000 to $60,000. Another option was to send samples out to other laboratories for measurement. One such service required the samples to be at least 2 mm thick (my samples are less than 1 mm) and did not have the ability to draw vacuum while taking the measurement. Therefore, I chose to build my own apparatus designed specifically for my sample geometry and with the ability to operate under vacuum.

Note that in this chapter, k refers to thermal conductivity rather than permeability as in previous chapters. It is an inconvenient coincidence that the scientific community has chosen the letter k for both of these unrelated material properties.

6.2 Guarded Hot Plate Method

The apparatus works by placing the sample between two parallel temperature reservoirs (T_h and T_c) and measuring the power required to hold the system at steady state. The power input is supplied by an electrical resistor. The difficulty in this technique lies in the necessity to direct the energy flux through the sample only.

To solve this problem, 2 main methods were employed 1) the use of vacuum, and 2) the use of a guard heater. Figure (6.1) shows the arrangement of this design.

The power to the guard heater is controlled such that $TC5 = TC1$. When this is achieved, the temperature gradient in the insulator between the guard heater and main heater is zero, therefore the heat flux is only in the $+z$ direction. Even if there is a small temperature difference between $TC5$ and $TC1$ (say 1°C), the low conductivity of the insulator ($\approx 0.1\, W/m{-}K$) keeps the heat flux in the $-z$ direction to a very small value. Since the insulator is approximately 6 mm thick and diameter is 1 inch, the heat flow Q as a function of $\Delta T = TC1 - TC5$ is

$$Q = -k_{ins} A \frac{\Delta T}{\Delta z} = 0.05 \cdot 6.45 \cdot 10^{-4} \cdot \Delta T / 0.006 = 0.0054 \Delta T \tag{6.1}$$

Therefore, controlling to two heaters to be within 2°C ensures that the heat flux through the insulator is less than $0.01\ W$, which is negligible compared to the steady-state flux during experiments which is $1 - 3\ W$.

Heat transfer out of the solid surfaces in the r direction can only happen by radiation. However, the aluminum surface has a very low emissivity value ($\epsilon \approx 0.1$), and the operating temperatures are also close to room temperature ($\approx 60^o C$). Therefore the heat flux via radiation is

$$Q = 0.006 \cdot 2\pi r \epsilon \sigma (T_h^4 - 298^4) = 2.71 \cdot 10^{-12} * (Th - 7.886 \cdot 10^9) \tag{6.2}$$

where σ here is the Stefan-Boltzmann constant. For $T_h = 60$°C the flux is $0.012\ W$, again negligible. From these results, it is acceptable to assume that all the power form the main heater is passing through the sample (whose thermal conductivity is to be measured) to the cold reservoir (the copper heat sink at T_c). If $dT/dr = 0$ the flux is equal to $Q\hat{\mathbf{z}}$ and a scalar equation can be written as

$$Q = kA\frac{dT}{dZ} = kA\frac{T_h - T_c}{h} \quad \therefore \quad k = \frac{Qh}{A(T_h - T_c)} \tag{6.3}$$

where h is the sample thickness, A is its area, and k is its thermal conductivity. To ensure that the temperature does not vary with r, two thermocouples were placed on each metal cylinder: one in the center ($r = 0$) to measure T_0 and one at the edge ($r = r_0$) to measure T_e. To account for any temperature variation in the r direction, the average temperature T_m of each metal cylinder is used (T_m is either T_h or T_c). Since only the center and edge temperatures are known and not the distribution, a linear function of r to T_m is assumed. Therefore, the average

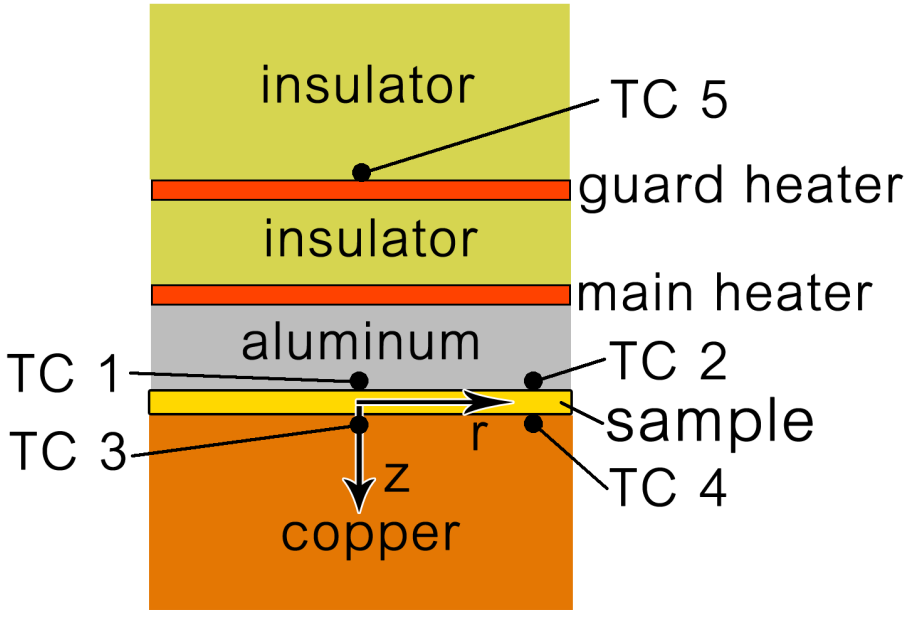

Figure 6.1: Diagram of the thermal conductivity apparatus. The aluminum cylinder is the hot plate at T_h while the copper is the cold reservoir at T_c. There are 5 thermocouples in the system labeled TC1 - TC5.

temperature of the disk surface $\bar{T}_m$ is

$$\bar{T}_m = \frac{1}{A_m}\int T_m dA_m = \frac{2}{r_0^2}\int_0^{r_0} T_m r dr = \frac{2}{r_0^2}\int_0^{r_0}\left(T_0 + \frac{r}{r_0}(T_e - T_o)\right) r dr = \frac{1}{3}(T_0 + 2T_e) \tag{6.4}$$

6.3 Mechanical Design and Construction

The temperature reservoirs are two 1 inch diameter cylinders: one copper (taller, cold reservoir) and one aluminum (shorter, heated). This choice of material and cylinder height was made to ensure that the cold reservoir has a large thermal mass and able to keep a steady temperature during operation, and that the hot cylinder would have low thermal mass and thus faster temperature response to heating. A jar mounts on an O-ring to create the enclosure which is to be evacuated with a rotary vacuum pump. The base piece was 3D printed and sealed with epoxy and is shown in figure (6.2).

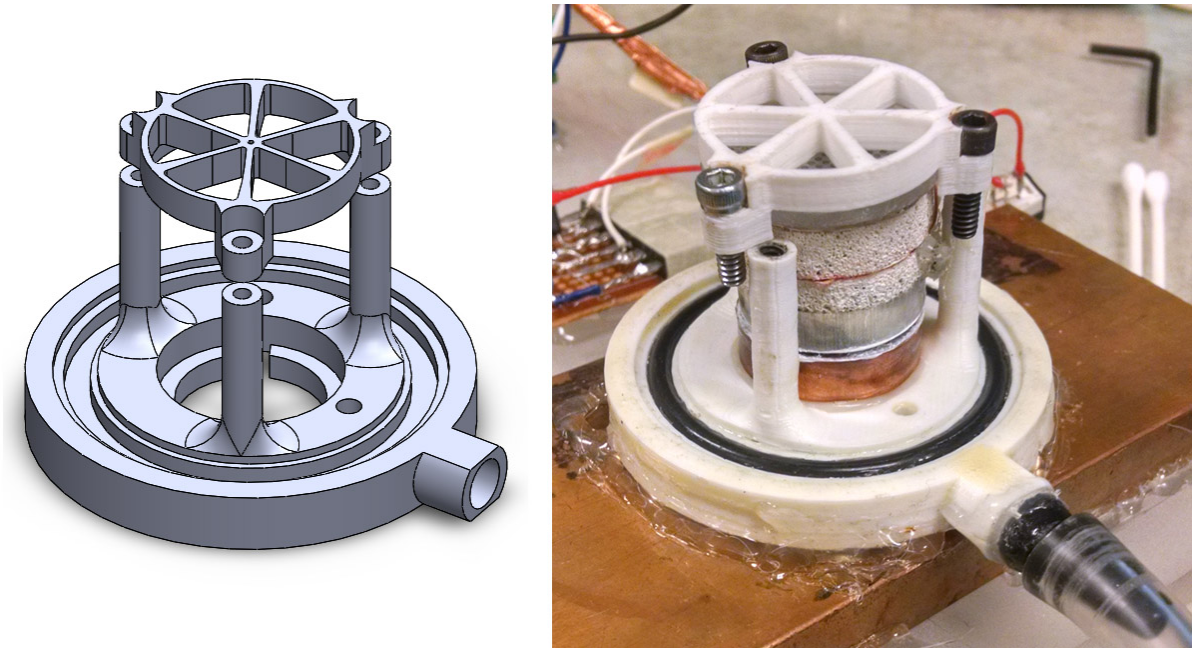

Figure 6.2: Base piece: CAD model and finished part.

Wiring and vacuum holes were built into the base. Three screws are used to provide the clamping force on the sample so that any gaps between the metal temperature reservoirs and the sample are eliminated. To further eliminate contact resistance, a thermal grease was used on both faces during experiments.

The Arduino UNO with ATmega 328 microcontroller was used along with a thermocouple shiled. This uses the MAX31855 K-type thermocouple amplifer (communicates via SPI) and a ADG608 analog multiplexer. This allows up to 8 thermocouples to connect to the same amplifier. The finished apparatus is shown in figure (6.4).

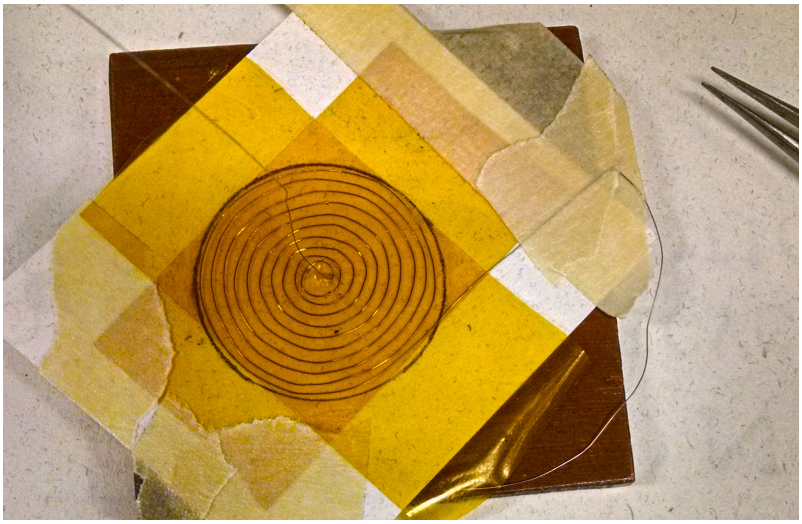

Figure 6.3: Hand wound Nickel-Chromium wire heater. The coil resistance is approximately 30 Ω.

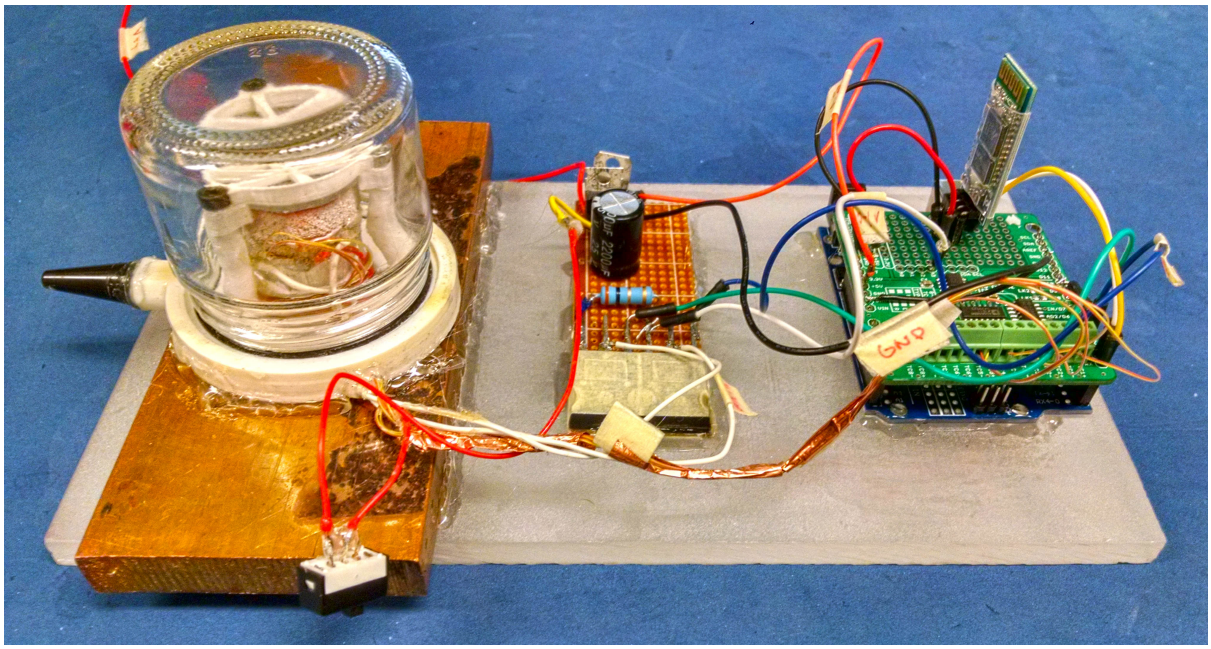

Figure 6.4: Finished thermal conductivity apparatus.

6.4 Governing Equations

It is now convenient to define the conductance $c \equiv k/h$ and temperature difference $T \equiv T_h - T_c$. The usefulness of these definitions will become apparent later. Equation (6.2) can now be written as

$$Q = cAT \tag{6.5}$$

To heat flow Q, is obtained by measuring the electrical power through the main heater. The circuit diagram is shown in figure (6.5). The MOSFET gate gets a pulse-width modulated

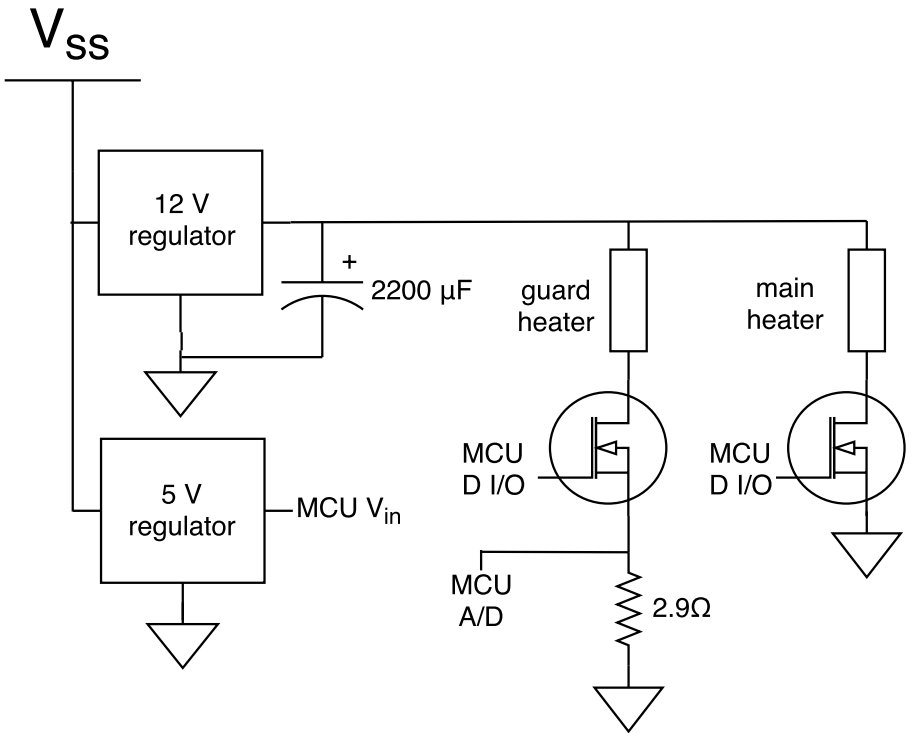

Figure 6.5: The heater circuit of the thermal conductivity apparatus.

signal from the microcontroller whose duty cycle D_c is controlled by passing a byte to the write function with the uniform map [0=0, 255=1]. Although this is a discreet map, it will be treated as continuous in modeling.

The 12V regulator outputs a constant voltage (measured) of $V_R = 11.95\ V$. The MOSFETs used are IRFP250N with an $R_{DS(ON)}$ of 0.075 Ω, therefore the current through the main heater can be computed by measuring the voltage across the 2.9 Ω resistor R_1. With a duty cycle of 1, the power to the heater is approximately 3.7 W. During measurements, the power was more precisely calculated using measurements of the voltage across R_1 (V_Ω) using the microcontroller's analog-to-digital converter.

$$Q = \frac{D_c}{255}\frac{V_R - V_\Omega}{R_1} \tag{6.6}$$

The control system can be displayed in block-diagram form, as shown in figure (6.6). The transfer functions of the first 3 blocks are as follows

$$A(s) = \frac{\mathcal{D}_c(s)}{\mathcal{T}_{ref}(s) - \mathcal{T}(s)} = K_p + K_d s + K_i/s \quad \text{(PID implemented in software)} \tag{6.7}$$

$$B(s) = \frac{\mathcal{Q}(s)}{\mathcal{D}_c(s)} = 3.7/255 \quad \text{(heater power (W) / pwm (bytes))} \tag{6.8}$$

$$L(s) = \frac{\mathcal{Q}(s)}{\mathcal{T}(s)} = cA \quad \text{(heat transfer through sample)} \tag{6.9}$$

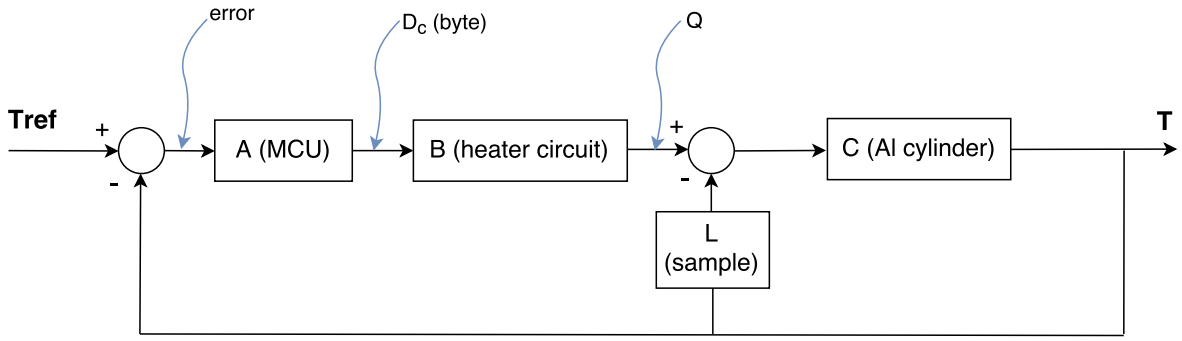

Figure 6.6: Block diagram of the thermal conductivity apparatus.

The transfer function of block C is yet to be determined. A simple model (assuming temperature uniformity) is

$$Q = mc_p \frac{\partial T}{\partial t} \tag{6.10}$$

we have

$$C(s) = \frac{\mathcal{L}(T)}{\mathcal{L}(Q)} = \frac{\mathcal{T}(s)}{\mathcal{Q}(s)} = \frac{1}{mc_p s} \tag{6.11}$$

Therefore we can expect the actual transfer function of C to be similar to this form. However, the temperature in the aluminum cylinder isn't uniform, especially during the transient period. For this reason, I experimentally determined the transfer function. The idea is simple: measure the step response of the system described by block C, fit with a polynomial, and then find the Laplace transform. Specifically,

$$Q(t) = Q_0 u_s(t) = 3.7 u_s(t) \tag{6.12}$$

$$T(0) = 0 \quad \text{(metal is initially at room temperature)} \tag{6.13}$$

The time response plot is shown in figure (6.7). A linear function fits the data very well, therefore the transfer function of C is

$$C(s) = \frac{\mathcal{L}(T(t))}{\mathcal{L}(Q(t))} = \frac{1}{3.7/s} \frac{0.0138s + 0.3522}{s^2} = \frac{0.0138s + 0.3522}{3.7s} \tag{6.14}$$

The combined transfer function for the feedback system is

$$\frac{\mathcal{T}}{\mathcal{T}_{ref}} = \frac{ABC}{1 + ABC + CL} \tag{6.15}$$

This system was simulated using the MATLAB control system toolbox (the script is in the appendix). After some tuning to avoid significant overshoot and keep the settling time low, the best I could do was a control scheme with the step response shown in figure 6.8). The large settling time is due to the fact that the power output was limited at 3.7 W (a duty cycle of 255/255).

Since the time to reach steady state is very long with PID control, I devised a different scheme which converge to steady-state much quicker. The PID controller seeks to set the

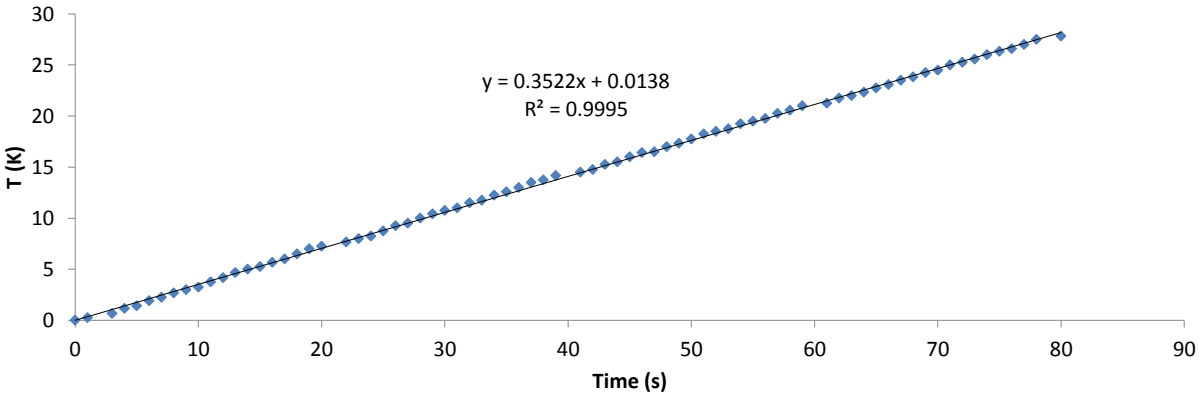

Figure 6.7: Step response of aluminum cylinder to a $3.7u_s(t)$ input.

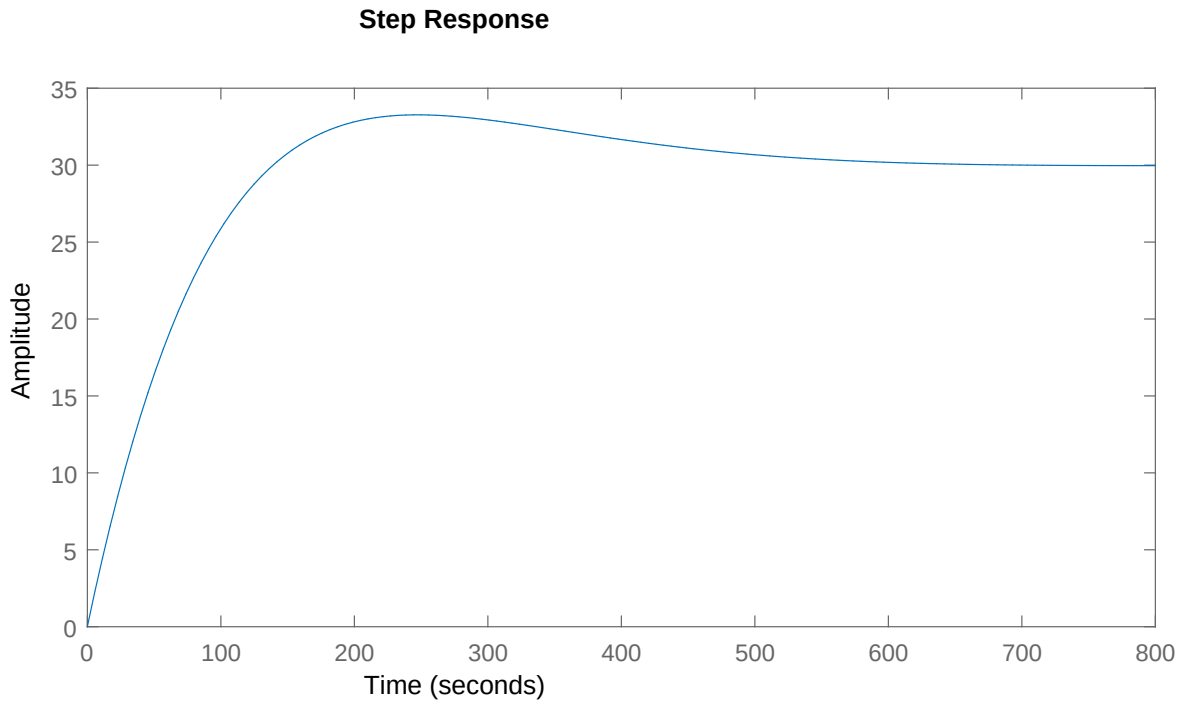

Figure 6.8: Step response simulation of PID controlled system with a conductance of 50 $W/(m^2 \cdot K)$.

system at a specified value of T. However, in determining the thermal conductivity, the value of T is not very important- the goal is to have $dT/dt = 0$, i.e. steady state. Therefore, it is more convenient to design a control scheme that seeks to minimize the absolute value of the derivative without strict requirements on the final value of T. The process works as follows:

1. Set the duty cycle to 255/255 (maximum power) for 90 seconds.
2. Use the value of $T(t)$ after 90 seconds of heating, $T(90)$, to estimate the conductance.
3. Using the estimated conductance, estimate the duty cycle (power) required to hold T at $T(90)$.

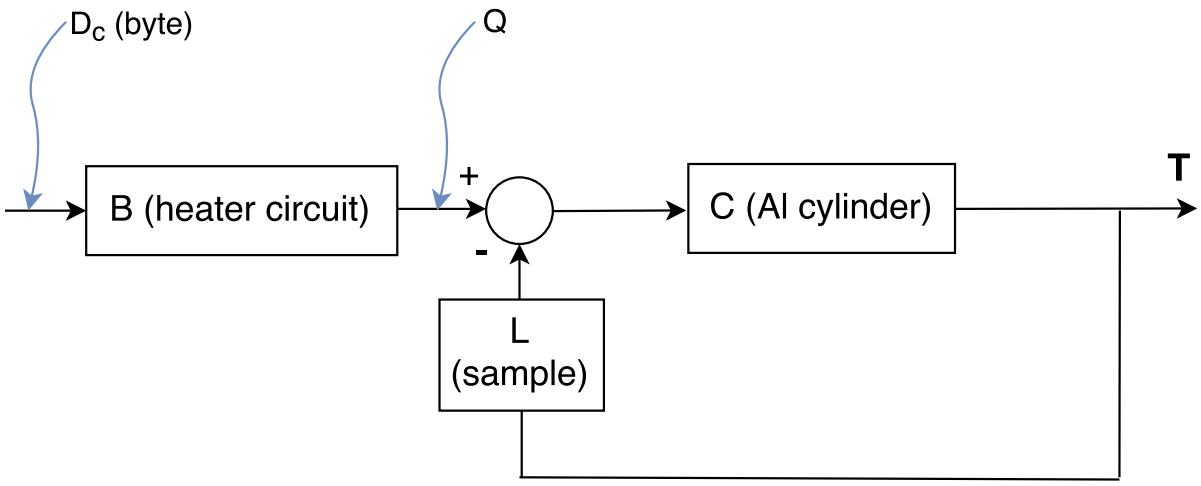

Figure 6.9: Block diagram of new system without feedback.

4. Algorithmically determine the duty cycle (power) that makes $dT/dt = 0$.

Once $dT/dt = 0$, the value of T can be used to compute the conductivity using equation (6.4). To implement this process, I created a method of estimating the conductance from the value of T for transient period (first 90 seconds). This is based on the system model developed earlier, except without the feedback controller. The block diagram is shown in figure (6.9).

The system in figure (6.9) was simulated for a step input of $D_c = 255$ for 30 values of conductance linearly spaced in the set [0, 1000]. The value conductance as a function of $T(90)$ was fit by the 4^{th} degree polynomial

$$c_{est}(T(90)) = 0.0056T^4 - 0.5195T^3 + 18.5135T^2 - 321.4312T + 2525.0 \tag{6.16}$$

Both the simulation data points and curve fit are plotted in figure (6.10).

Once the conductance is estimated, the power transmitted through the sample at steady state with $T(t \to \infty) = T(90)$ is estimated to be

$$Q_{est} = c_{est}AT(90) \tag{6.17}$$

Since this is only an estimate based on simulation, it is not guaranteed to yield $dT/dt = 0$ when the power is set to $Qest$. However, this value is used as a starting point for the algorithm that zeros the derivative through iteration. Finding the zero of a function is a common problem; in this case, the function is evaluated by actually measuring T using the thermocouples. The algorithm works as follows:

1. Begin with $Q_1 = Q_{est}$
2. Compute $d_1 = \frac{d^2T}{dt^2}$
3. If $d_1 > 0$, set $Q_2 = 0.8Q_{est}$, else $Q_2 = 1.2Q_{est}$
4. Set the duty cycle to output power Q_2
5. Wait 100 ms for the system to settle, then compute the new derivative $d_2 = \frac{d^2T}{dt^2}$

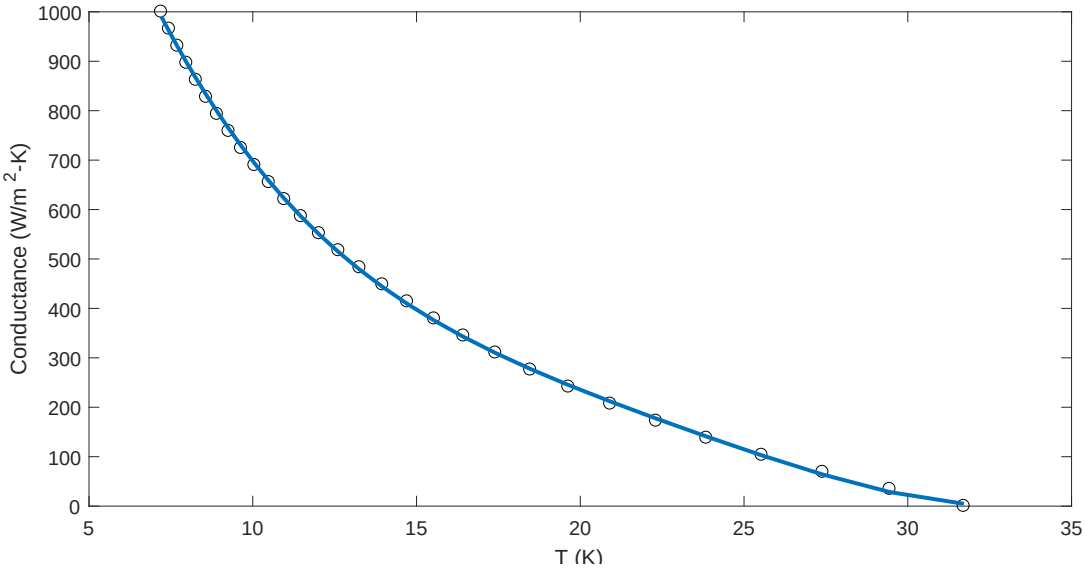

Figure 6.10: Step response of aluminum cylinder to a $3.7u_s(t)$ input.

6. Compute and set the power Q_0 that will make dT/dt closer to 0 by $Q_0 = Q_1 - \frac{d_1(Q_1 - Q_2}{d_1 - d_2}$
7. Compute $d_0 = \frac{d^2T}{dt^2}$. If $d_2 <$ tolerance, exit. Otherwise, perform steps 8-9
8. For next iteration, set: $Q_1 = Q_2$, $Q_2 = Q_0$, $d_1 = d_2$, $d_2 = d_0$
9. Repeat steps 6-9

The code that implements this method is shown here for convenience:

```
// compute derivatives to get optimization started
  float d1 = Tderivative();
  float Q2;
  float d0 = d1;
  float Q1 = powerByte;
  if (abs(d1) < tol) return;
  if (d1 > 0) {
    Q2 = 0.8 * Q1;
  } else {
    Q2 = 1.2 * Q1;
  }
  analogWrite(gatePin, Q2);
  delay(100); // give it a chance for derivative to settle
  float d2 = Tderivative();
  // now start loop to converge on zero derivative
  Serial.print("@Loop to make derivative zero@");
  while (abs(d0) > tol) {
    powerByte = Q1 - d1 * (Q1 - Q2) / (d1 - d2);
    analogWrite(gatePin, powerByte);
    d0 = Tderivative();
```

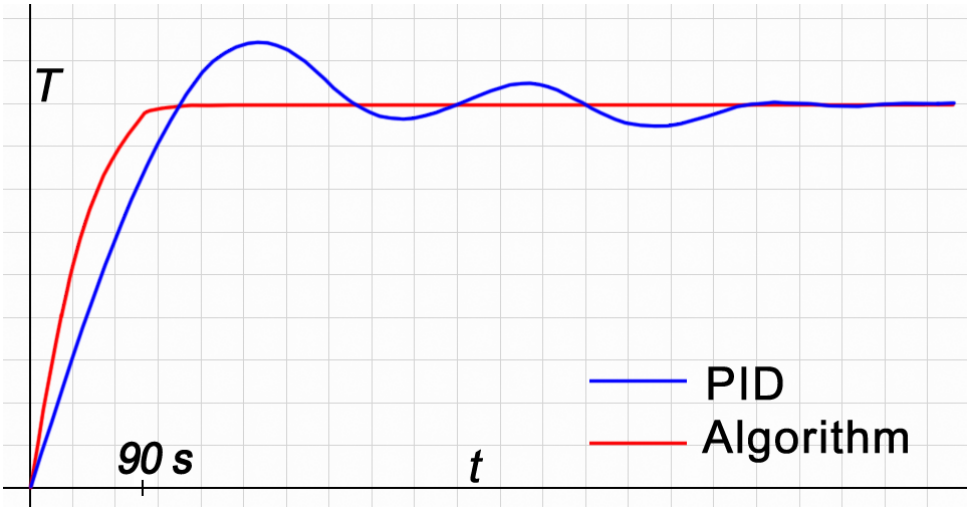

Figure 6.11: Sketch of a $T(t)$ plot showing the difference between a PID controller and my method. Note the difference in settling time.

```
    Q1 = Q2;
    Q2 = powerByte;
    d1 = d2;
    d2 = d0;
    printDeltaT(); printTime(); printHotCold();
}
```

This algorithm converges on a zero-derivative very rapidly. Since the system is rather slow, the steady state value of T is very close to $T(90)$. Figure (6.11) is a sketch of the difference between PID control and my method. Notice that the rise time is also faster since the power input is maximum for the first 90 seconds, while the proportional term in the PID controller causes the power input be maximum only at $t = 0$.

6.5 Android Application

To communicate with the microcontroller, I made an Android application that transmits data using streams via a Bluetooth proxy. The software is used as follows:

- Open the Android application called ”Thermal Conductivity”
- Power ON the microcontroller
- Click the “Connect” button in the app. If successful, the top-right rectangle will turn green.
- The app will prompt the user for the sample thickness (figure (6.12) left). Enter the thickness in the text box and press the SEND button.
- Next, the user is asked to enter in the sample area in mm^2. Enter and send it in the same way.

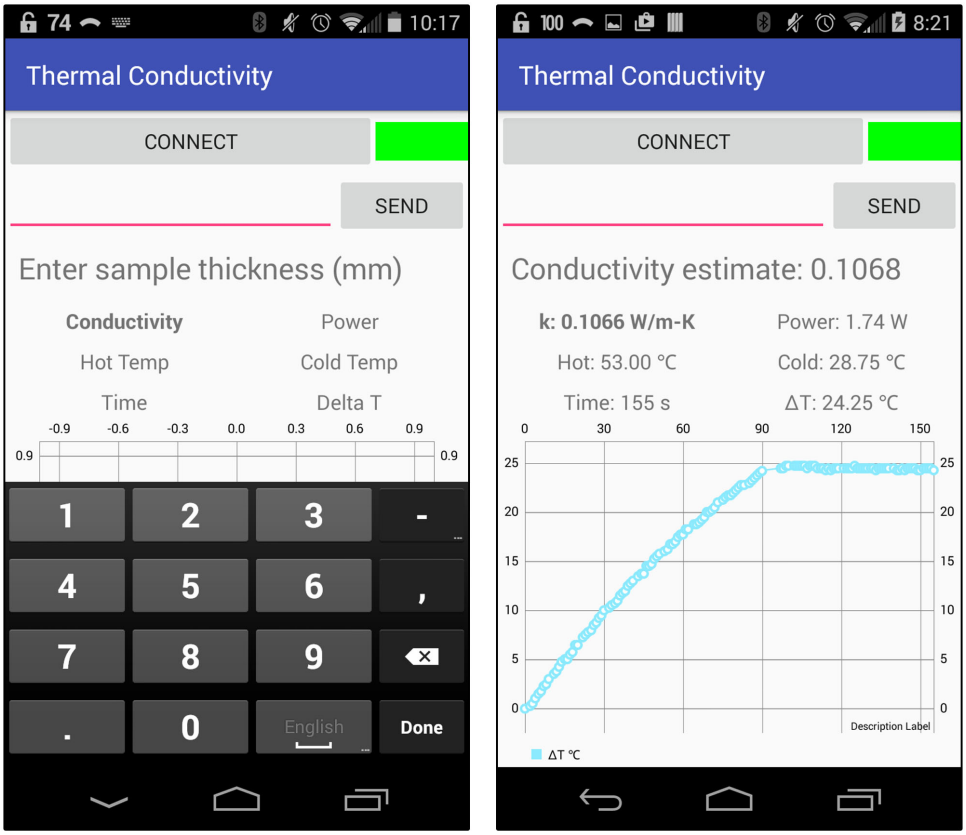

Figure 6.12: Left: Android application prompting user for sample thickness. Right: Apparatus running with information being updated on the screen in real-time. Note the plot of T at $t = 90$ showing the effectiveness of the control algorithm in achieving steady-state soon after the transient heating period of 90 seconds.

- The apparatus will run and update the information in real-time. $T(t)$ will be plotted in the Chart View (figure (6.12) right).

6.6 Area Calculation Using Image Processing

The apparatus must know the sample's area to compute the thermal conductivity. Unless the sample is cut into a shape that is accurately described by a circle or primitive polygon, it is very difficult to measure its area. For this reason, I wrote a MATLAB script that automates this process.

To use the MATLAB code, a U.S. quarter coin is placed on the top left of a dark sheet, and the samples are placed without overlap on the lower right side of the coin. The MATLAB script takes in the name of the image file and runs. The software will display a figure where each sample's boundary is encircled and the area (in mm^2) will be displayed right on the image.

The script is based on finding the connected regions of a logical array - a common algorithm in image processing. The coin's purpose is to calibrate the units from pixels to mm^2. An example image is shown in figure (6.13).

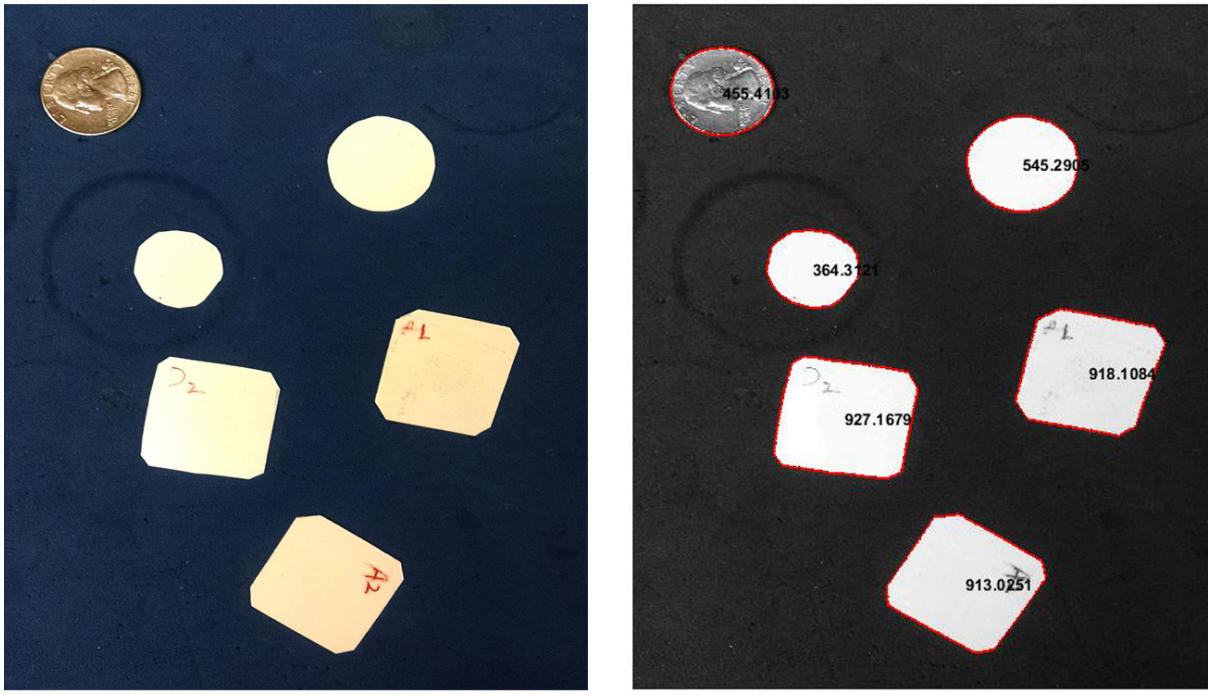

Figure 6.13: Left: Photograph of several PEI sample with a U.S. quarter coin on the top left. Right: Figure outputted by the image processing script displaying the areas of each sample (mm^2) and its boundary (red).

6.7 Experimental Results for PEI Foam

PEI foam samples of both the opaque and transparent variety were tested in this thermal conductivity apparatus. First, a raw PEI sample was tested to verify the accuracy of the device. The literature value for PEI's thermal conductivity is 0.22 W/m-K. The apparatus measured it to be 0.2214 W/m-K. The number of decimal points reported are not due to any particular significance but rather by picking an arbitrary truncation of the 32-bit floating point number returned by the microcontroller.

Three different samples at each foaming temperature were tested. The error bars represent the span of data (minimum to maximum value obtained) while the marker is at the mean. The raw data is also presented in table (6.1). Transparent PEI samples were saturated at 5.0 MPa and -30°C while Opaque PEI samples were saturated at 5.0 MPa and 25°C.

The curve fit for the plot in figure (6.14) is

$$k(T) = 0.1665 - 0.0176T \tag{6.18}$$

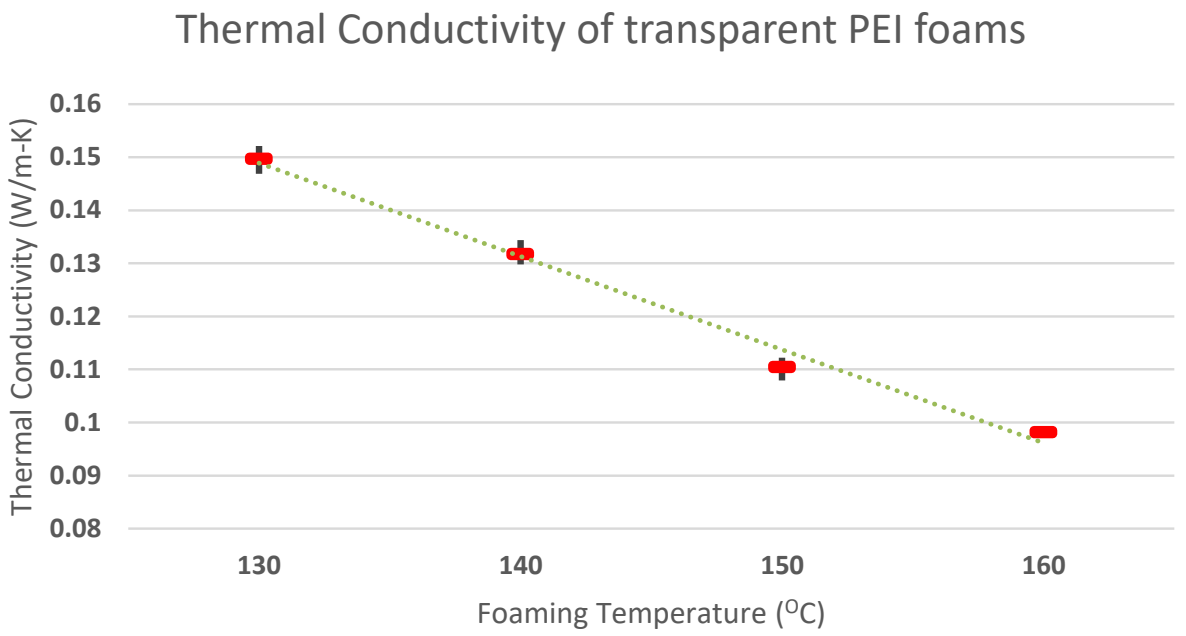

Figure 6.14: Thermal conductivity for PEI foam samples prepared from 0.50 mm thick sheets, saturated at -30°C with CO_2 at 5.0 MPa and foamed at several temperatures.

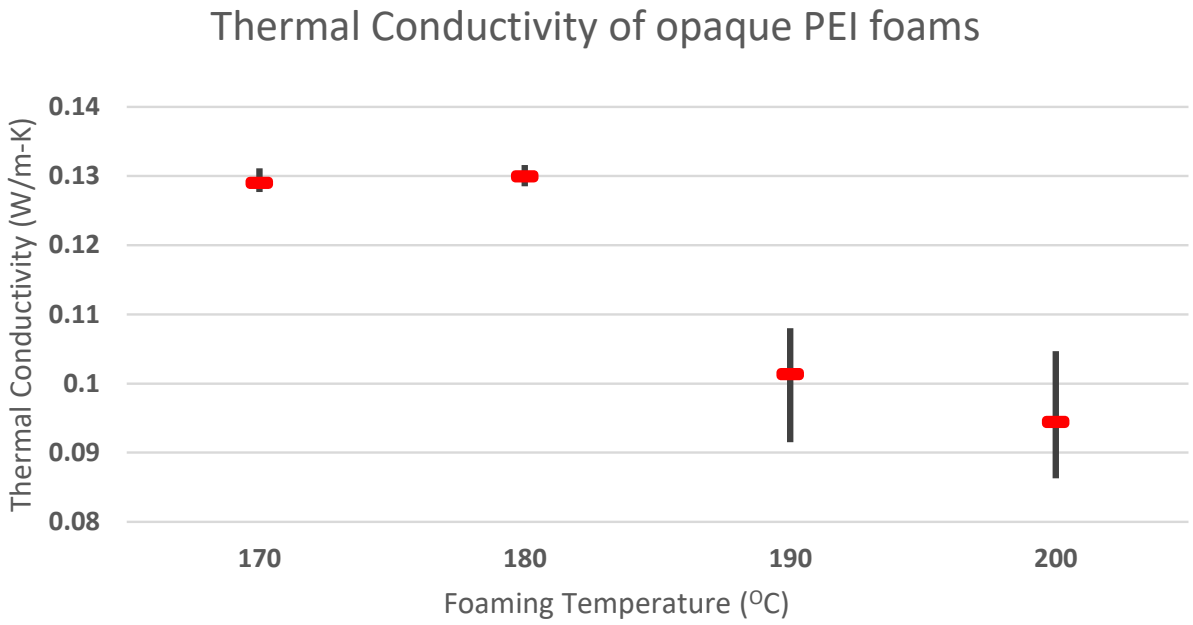

Figure 6.15: Thermal conductivity for PEI foam samples prepared from 0.50 mm thick sheets, saturated at -30°C with CO_2 at 5.0 MPa and foamed at several temperatures.

Table 6.1: Thermal conductivity of PEI processed at various conditions.

Type / Foaming Temp $(^{\circ}C)$**:**	**Thermal conductivity** k **W/m-K**	k **st.dev**	**Relative Density** ρ_{foam}/ρ_{PEI}
Transparent 130	0.150	1.89E-3	0.449
Transparent 140	0.132	1.95E-3	0.414
Transparent 150	0.110	1.85E-3	0.377
Transparent 160	0.098	0.77E-3	0.347
Transparent 170	0.101	2.19E-3	0.359
Opaque 170	0.129	1.49E-3	0.531
Opaque 180	0.130	1.26E-3	0.499
Opaque 190	0.101	7.11E-3	0.450
Opaque 200	0.094	7.66E-3	0.414

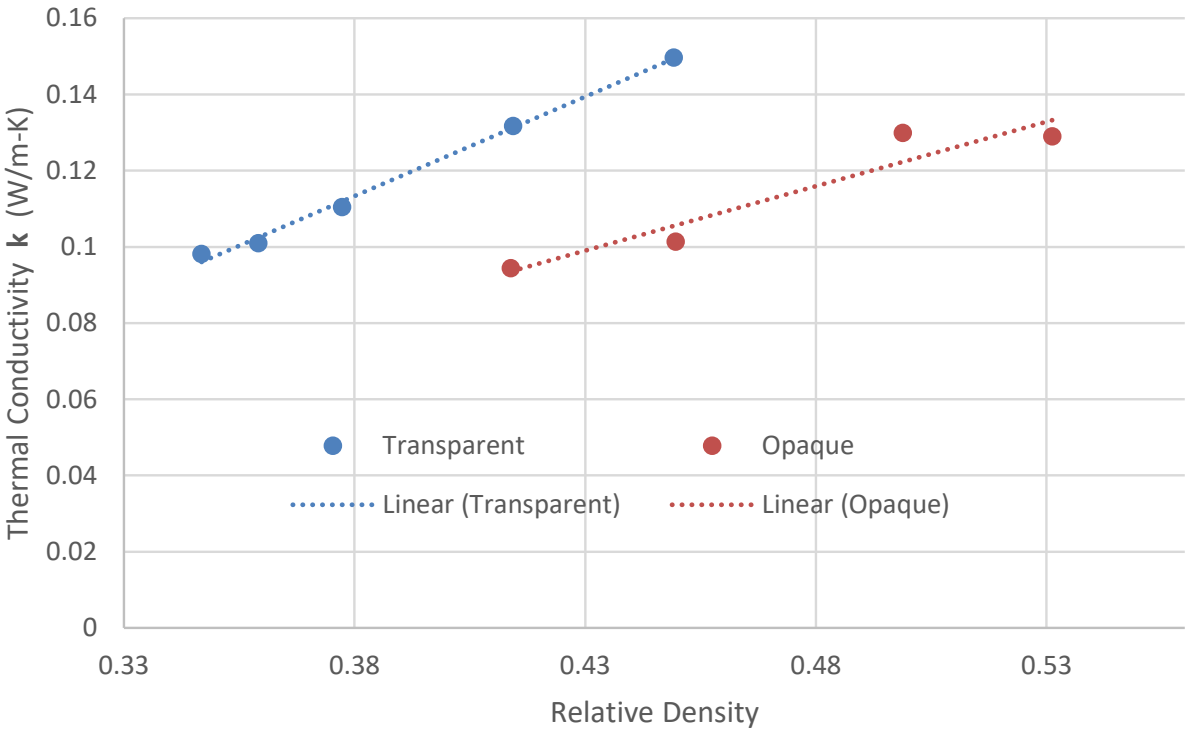

Figure 6.16: Thermal conductivity of all samples on the same plot, as function of relative density.

Chapter 7

Applications

7.1 PEI Samples Under High Pressure Water

Since my initial laboratory tests (section 2.8) to determine the water intrusion pressure in PEI have failed to force water into the nanostructure, I found a more capable pressure vessel in the Ocean Sciences department at the UW. This vessel was originally built to test autonomous submarine subsystems and is capable of producing water pressures in excess of 8000 psi. PEI foam samples of 0.50 mm starting thickness and saturated at 5.0 MPa / room temperature were prepared as follows: foamed at 170, 180, 190, and 200 °C, pierced and non-pierced versions, for testing at 1000, 2000, 4000, and 8000 psi, for a total of 32 samples. The samples were double-bagged in flexible zip-style plastic bags filled with fresh water in order to avoid contamination with the oily water in the pressure vessel. The vessel was then pressurized and held for 10 minutes at each specified pressure.

The pierced samples were then wiped dry using paper towel and weighed in order to determine if they absorbed water. After drying for 10 days, the samples were weighed again to obtain the dry mass. A rough calculation that shows how much pore space was filled by water can be performed as follows: let m_s and m_d be the mass of soaked and dry samples respectively. Let V denote volume and ρ density. If water fills the entire pore volume, then

$$m_s = V_{pores}\rho_{water} + V_{solid}\rho_{PEI} \tag{7.1}$$

$$m_d = V_{pores}\rho_{air} + V_{solid}\rho_{PEI} \tag{7.2}$$

and

$$\frac{m_d}{V_{pores} + V_{solid}} = \rho_{foam} \tag{7.3}$$

Since the density of air is approximately very small compared to that of water or polymer, it can be ignored in this context. Therefore

$$m_s - m_d = V_{pores}\rho_{water} = \rho_{water} m_d \left(\frac{1}{\rho_{foam}} - \frac{1}{\rho_{PEI}}\right) \tag{7.4}$$

Therefore, equation (7.4) gives the maximum mass of water that can be absorbed by a sample with relative density ρ_r. Table (7.1) shows the ratio of actual water mass absorbed by the sample and the maximum amount as computed by equation (7.4). It should be noted that this

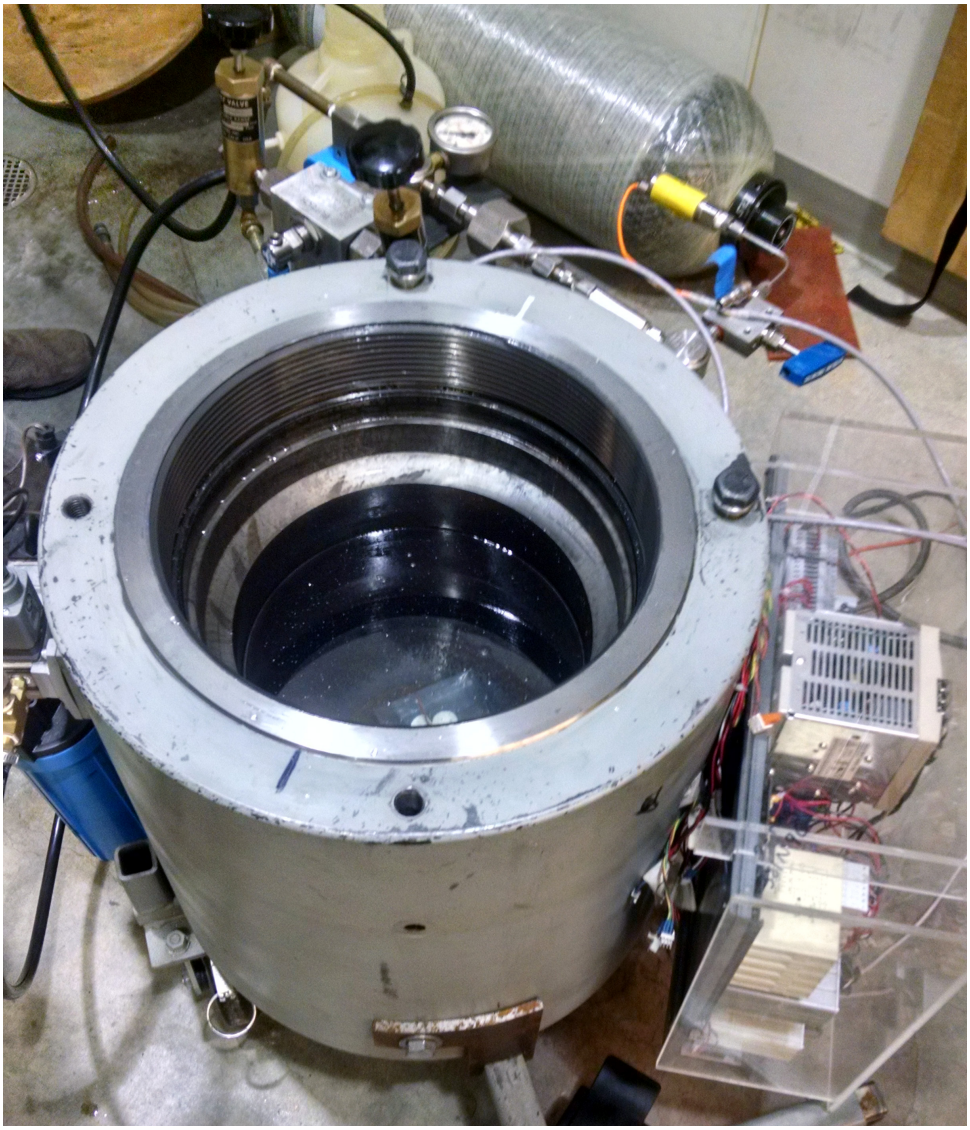

Figure 7.1: Pressure vessel used for the tests described in this section. The system uses tap water and is capable of pressurizing it to over 8000 psi.

is a very rough calculation since there are many sources of error. Firstly, the value of ρ_r is used as the average value computed for samples foamed at the respective temperatures; this could vary from sample to sample and I did not measure the relative density of all 32 samples in this experiment. Secondly, not all water intruded has remained inside the structure until weighing; some was pushed out by the residual gas in the voids, while another part evaporated during the hours between the experiment and weighing event. However, it does show clear evidence that water did indeed intrude the nanostructure.

Since every entry in table (7.1) is greater than zero, it means that water did intrude the nanostructure for all pressures. Thus, it can be concluded that 1000 psi is sufficient pressure for water to enter PEI nanostructures foamed between 170 and 200 Celsius. Since the previous tests showed that water did not go in at 800 psi, it can be concluded that the intrusion pressure is between 800 and 1000 psi. This corresponds to an underwater depth of approximately 550 meters.

	170°C	180°C	190°C	200°C
1000 psi	92.9%	96.2%	93.6%	97.7%
2000 psi	90.4%	91.9%	91.6%	76.7%
4000 psi	88.8%	62.7%	91.7%	89.6%
8000 psi	91.3%	92.8%	89.2%	84.6%

Table 7.1: Table showing a rough calculation of the pore space that was intruded by water in pressure tests.

Another observation made from these experiments is the compression of the samples due to the water pressure exceeding the yield stress of the foam structure. The deformation was most apparent in the samples foamed at 200 ° C, since they had the lowest density and therefore the lowest strength.

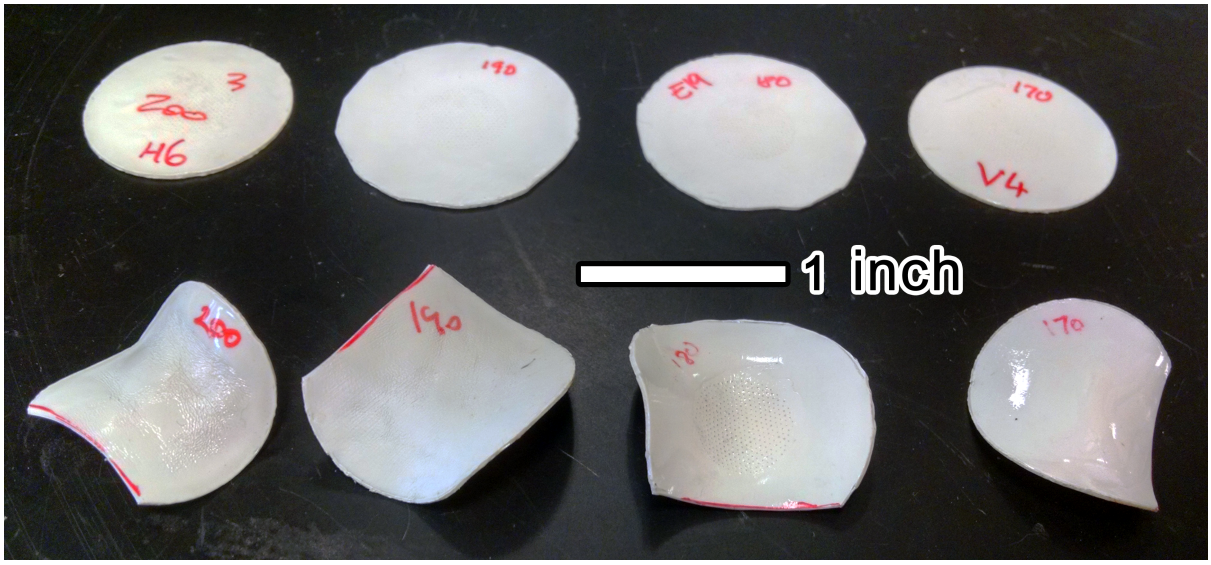

Figure 7.2: Samples after being exposed to 8000 psi of water pressure for 10 minutes. Samples on the top row had their skin pierced by needles. In both rows, from left to right, the foaming temperature is 200°C, 190°C, 180°C, 170°C.

Figure (7.2) shows the samples after the experiment. The first thing to note is that the samples with pierced skins (top row) were not deformed since water was able to enter the nanostructure and equalize the pressure. The other samples were deformed in a similar shape as potato chips. The density of these deformed samples was measured and shown in table (7.2).

SEM analysis was also performed on the samples from table (7.2) to visually examine the nanostructure of these compressed samples.

The goal of study was to determine the limitations of this material in underwater applications. It was shown that the material is not waterproof at 1000 psi or above, and the structure itself can resist 1000 psi but collapses by 13% at 2000 psi. Therefore, this material is suitable for medium-depth underwater applications which don't exceed 500 meters depth.

Pressure	Foam density $[kg/m^3]$	Relative Density	Density Increase
1000 psi	0.527	0.415	0%
2000 psi	0.593	0.467	13%
4000 psi	0.755	0.594	44%
8000 psi	0.828	0.652	58%

Table 7.2: Data showing the effect of high hydrostatic pressure on the samples foamed at 200C.

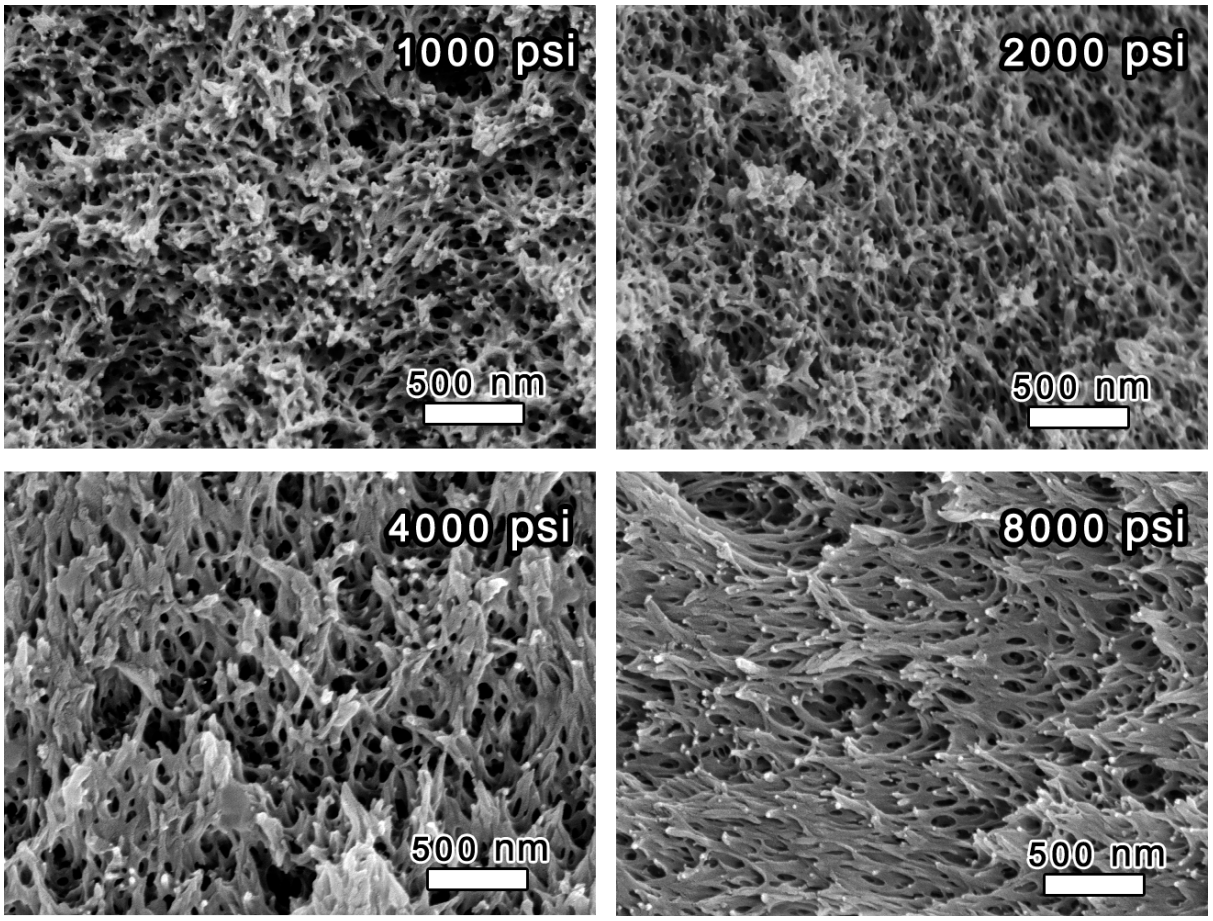

Figure 7.3: SEM images of the samples in table (7.2).

7.2 Expanding the Permeability Range

To expand the application space, I set out to determine if the foaming process could be tweaked to produce samples with higher permeability k. Since previous results showed that permeability increases with porosity, it was hypothesized that creating nanofoams with increased porosity (lower density) would increase permeability.

One idea to increase porosity was to start with a closed-cell foam and then saturate it and expand it again. First, the closed cell foam was made using low saturation pressure (such as 1.0 MPa) following Miller [10]. This yielded high-density closed cell foams at all foaming

temperatures. Then, these samples were put back into the pressure vessel, this time at higher pressure (e.g. 5.0 MPa) and foamed at temperatures between 170-200 °C following Miller's recipe of open-cell nanofoams [10]. Figure 7.4 shows SEMs after the first and second foaming steps for two different samples.

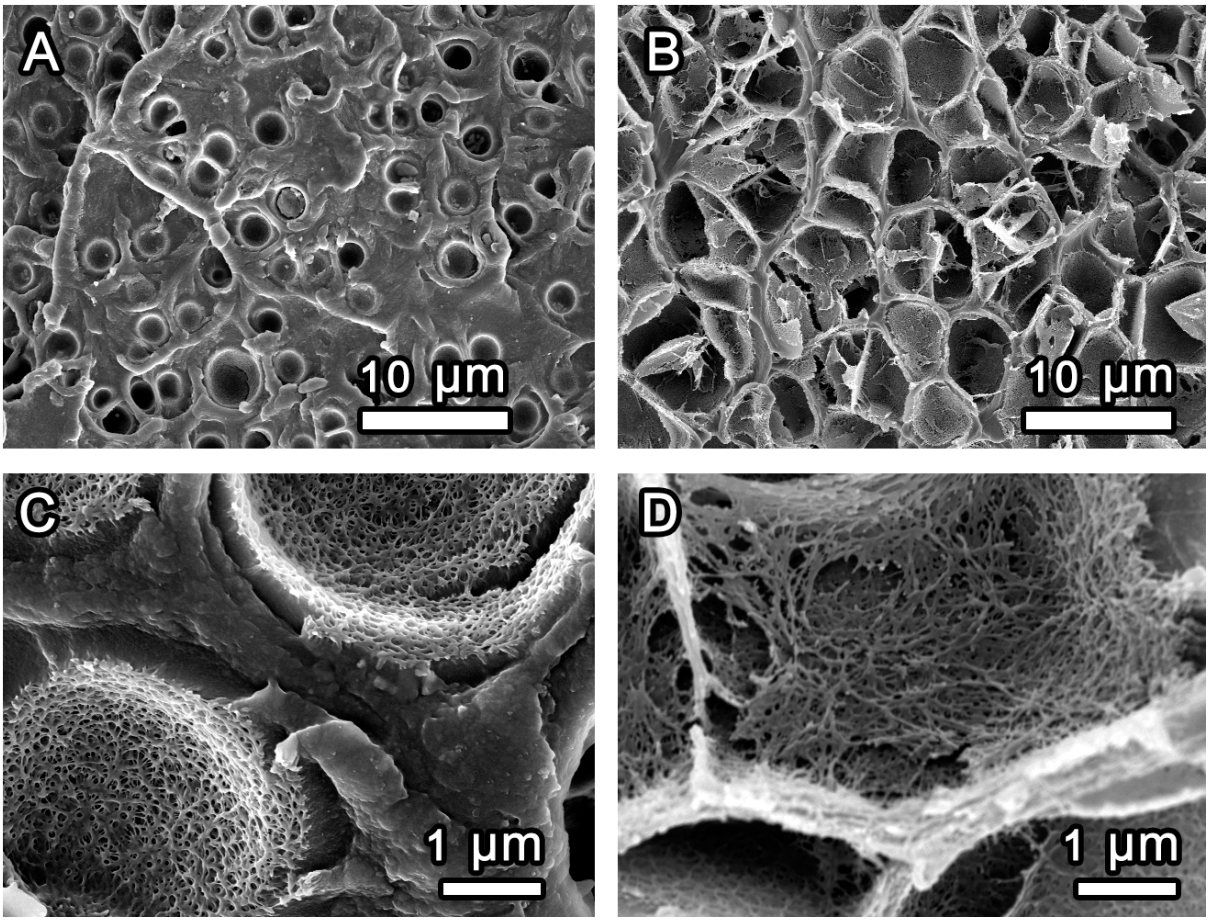

Figure 7.4: SEM images of PEI samples showing the effect of the double foaming technique. The saturation pressure and foaming conditions are as follows: A)1.0 MPa, 170°C. B) 1.0 MPa, 170°C, then 5.0 MPa, 190°C. C) 1 MPa, 190°C. D) 1 MPa 190°C, then 5 MPa, 190°C.

During the second saturation (of the closed-cell foamed samples) the gas saturates the polymer but also enters the voids. Upon heating, these voids expand (like a balloon) to increase their size. This is apparent in the SEMs of figure (7.4). During this expansion, the cell walls rupture and create an open structure.

Permeability tests were performed on these samples with different combinations of first and second foaming conditions. The results are in table (7.3).

This data shows that the porosity is greatly increased, as expected. The highest porosity reported by Miller is 0.605 in a PEI sample saturated at 5.0 MPa and foamed at 200°C. However, the double-foamed sample in the third last row of table (7.3) has a porosity of 0.83, or 37% higher. However, the permeability of this same sample, at 9.28E-16 m^2 is 44% less than that of a sample foamed once at 200 °C after 5.0 MPa saturation (which has a permeability of 1.66E-15 m^2). Perhaps the reason for this is that, although the cells in the porous structure are larger, the paths through the cell wall are still small.

First foam	Second foam	Porosity	Permeability$[m^2]$
1.0 MPa, 160°C	5.0 MPa, 190°C	0.68	1.20E-17
1.0 MPa, 190°C	5.0 MPa, 190°C	0.75	3.77E-17
1.0 MPa, 190°C	5.0 MPa, 200°C	0.83	9.29E-16

Table 7.3: Data showing the porosity and permeability of twice-foamed samples.

These experiments also led to some interesting observations on how the porous structure developed. Figure (7.5) shows some intriguing images of a few cellular structures. Image (A) shows some obscure artifacts inside the cells after the first foaming step; note that these cells are of round/oval shape and sparsely spaced (figure 7.4). After the second foaming, the cells expand until only a thin wall separates them, and are no longer round but have polygonal wall shapes. The wall is also broken and frayed, giving the notion of an interconnected open structure. However, figure (7.5 D) shows that in some cases this cell-expansion phenomena doesn't occur, and instead an open-cell interconnected structure is nucleated between the original large cells. These two interesting foaming regimes were not further investigated but could be a very interesting future study.

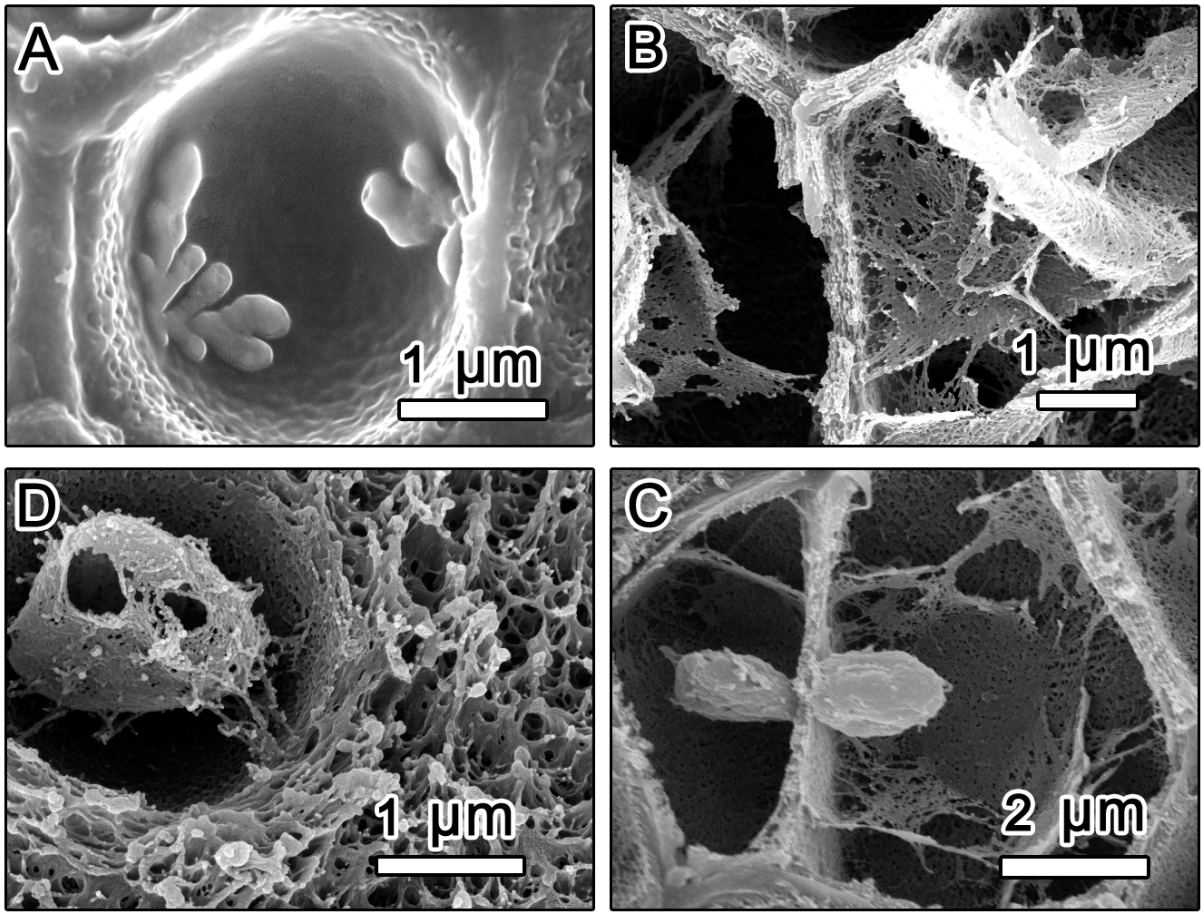

Figure 7.5: Interesting SEM images of samples. The saturation pressure and foaming conditions are as follows: A)1.0 MPa, 180°C. B) 1.0 MPa, 170°C, then 5.0 MPa, 190°C. C) 1 MPa, 160°C, then 5.4 MPa, 200°C. D) 1 MPa 160°C, then 5.4 MPa, 210°C.

7.3 Thermoforming Nanoporous PEI

Some applications of this nanoporous material needs curved shapes rather than flat sheets. Since PEI is a thermoplastic polymer, it can be thermoformed. However, it was hypothesized that the thermoforming process may destroy the nanostructure. To examine the effect of thermoforming on PEI's nanostructure, an undergraduate team was recruited to design and perform some experiments that test this hypothesis under my supervision. I then performed scanning-electron microscopy and analysis.

PEI samples cut from 0.5 mm sheets were saturated at 5.0 MPa and foamed at 180°C. The molds were then preheated in a convention oven at 230°C. Samples were placed in the molds with a 4 kg mass to provide the clamping force. The samples were left in the oven for 15 minutes and then cooled at room temperature.

The mold shapes were: dome, cylindrical, and V-shaped and are shown in figure (7.6). The motivation for these shapes is that the spherical shape has non-zero Gaussian curvature, the cylinder has zero Gaussian curvature, and the V-shape has a small radius of curvature at the apex.

The SEM results show that even after heating the samples above its 217°C glass transition

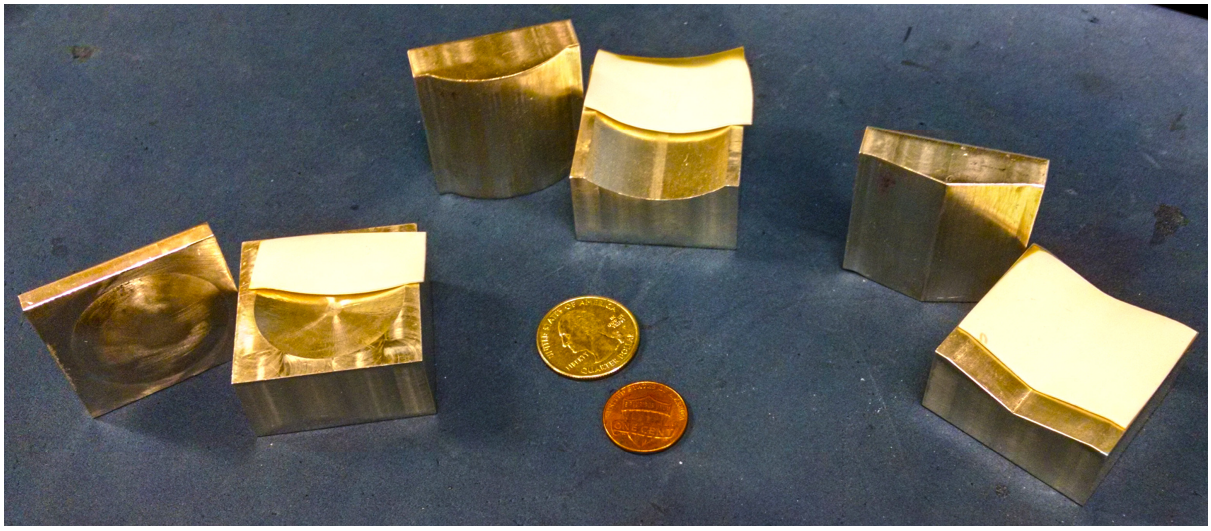

Figure 7.6: Photograph of the three molds used in the thermoforming experiments. The PEI samples are placed on the molds (they were cut for SEM). The coins are for length scale reference.

temperature, the cellular structure remained after the thermoforming process with all three molds.

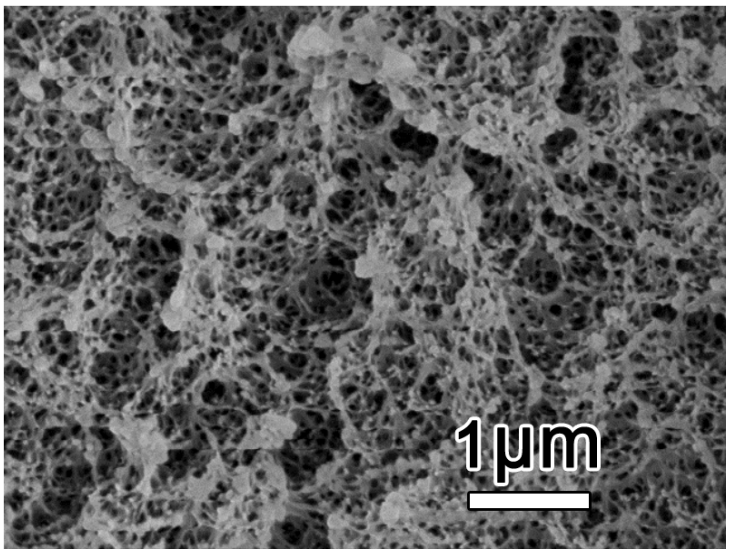

Figure 7.7: SEM of the sample made using the dome shaped mold.

This study showed feasibility of thermoforming nanoporous PEI sheets into more useful products.

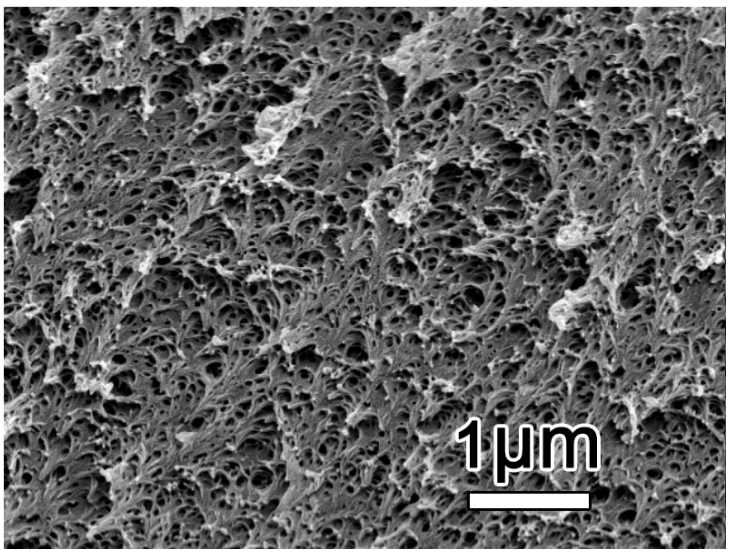

Figure 7.8: SEM of the sample made using the cylindrical shaped mold.

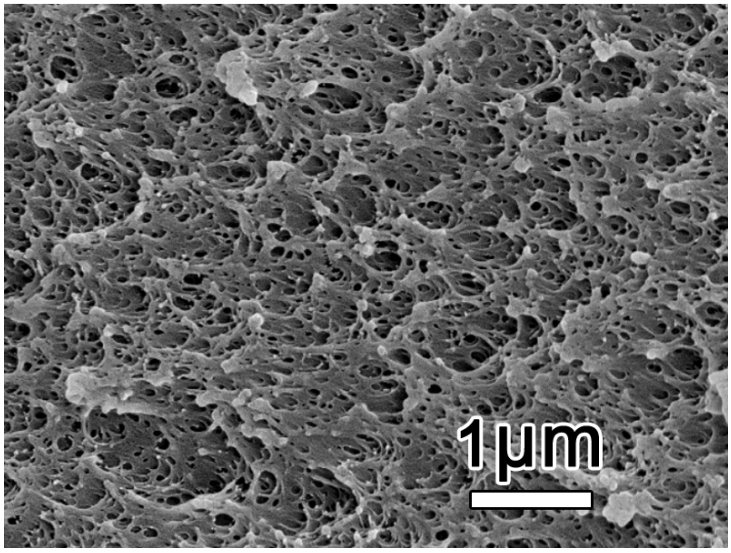

Figure 7.9: SEM of the sample made using the V-shaped shaped mold.

Chapter 8

Additional Experiments

The quest to find an appropriate Ph.D project consisted of much exploratory work and experimentation in diverse areas. The most significant are concisely described in this chapter.

8.1 Foaming by Hot Wire

Traditional foaming in a heated fluid bath is approximately an isothermal process in which knowledge of energy transfer is not obtainable. An alternative method is to put a known amount of thermal energy into the system (a saturated polymer sample) and study how nucleation occurs. The idea was simple: embed a wire into a sample, saturate it with gas, and heat the wire by passing an electric current. Samples were created by two methods, as depicted in figure (8.1). The sheet fusing method proved to be much more effective by producing samples of highly uniform thickness.

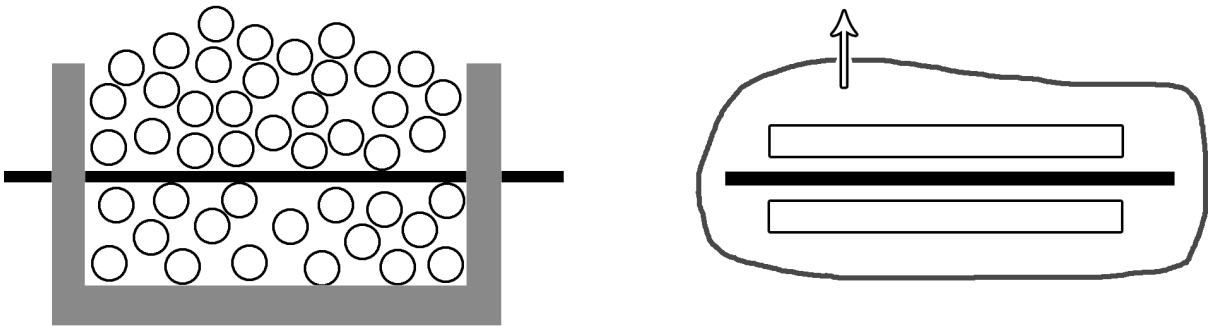

Figure 8.1: Methods of producing polycarbonate samples with an embedded wire. Left: Filling a mold with pellets and heating to 200°C. Right: Placing the wire between two sheets, enclosing in a vacuum bag, and heating to 160°C.

The power input could be controlled by pulse width modulation (PWM). To obtain the wire's temperature, the wire's resistance is measured by using a Wheatstone bridge and amplifier, keeping in mind that metals such as Nichrome (NiCr alloy) have a temperature dependent resistance, although small. Referring to the circuit diagram in figure (8.2), $R2$ is the wire and $R1$ a small valued resistance $(< 1\Omega)$. At room temperature,

$$\frac{R2}{R1} = \frac{R4}{R3} \tag{8.1}$$

If $R5/R6 = R7/R8$ the amplifier gain A is given by

$$A = R6/R5 \tag{8.2}$$

To improve the measurement's accuracy, the amplifier must draw a very low current. Therefore, it should be true that $R5 >> R4$. Also, to save power, $R4 >> R2$. An injective function between the wire's temperature T and amplifier output voltage V_O exists. Since the temperature range of interest is approximately $20 - 200°C$ which is small compared to the wire's functional temperature range, this mapping function can be well approximated by its first two Taylor series terms

$$T = a + bV_O + cV_O^2 \tag{8.3}$$

The constants a, b, c are to be obtained by placing a sample at various known temperature and measuring the respective V_O at equilibrium. In general, the output is dependent on the length of the wire used since $dR1/dT$ is not constant but rather $(dR1/dT)/R$.

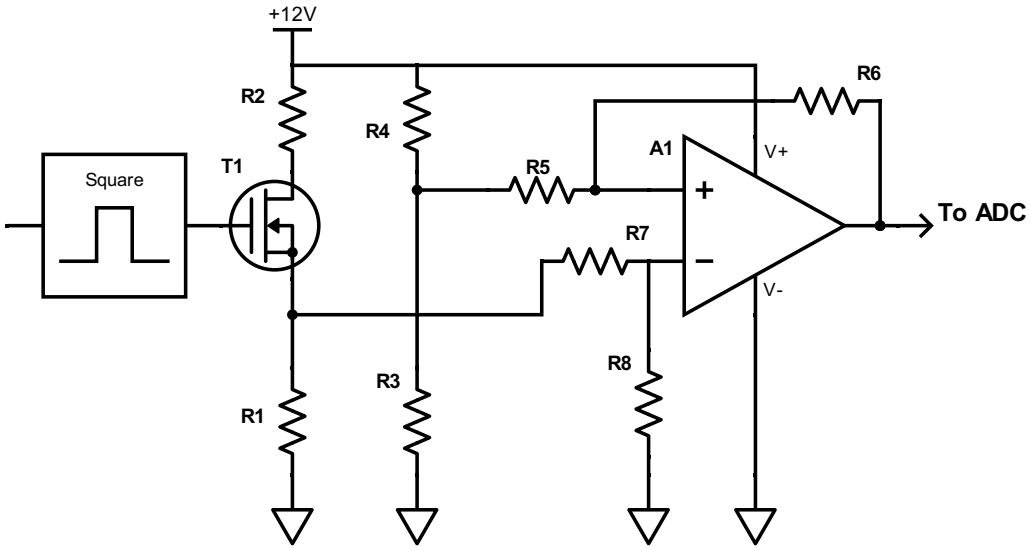

Figure 8.2: Circuit diagram of the embedded wire $R2$ and its power delivery and resistance measurement technique.

With this setup, some research goals were as follows: 2

- Determine the energy absorbed by the system during the foaming process. This is done by plotting energy vs. temperature for saturated and unsaturated specimen and subtracting, similar to the differential scanning calorimetry (DSC) technique but with much quicker heating rates.

- Create images of cell structure development through time by creating multiple samples and foaming for different amounts of time. Since the wire has a very low heat capacity and the electronics can be controlled to a microsecond order accuracy, the foaming event could be captured with good time resolution.

- Analyze the cellular structure as a function of radial distance from the wire and create a model. The temperature profile in the polymer around the wire during the transient heating period is 2D axisymetric, so the structure was expected to reflect this fact.

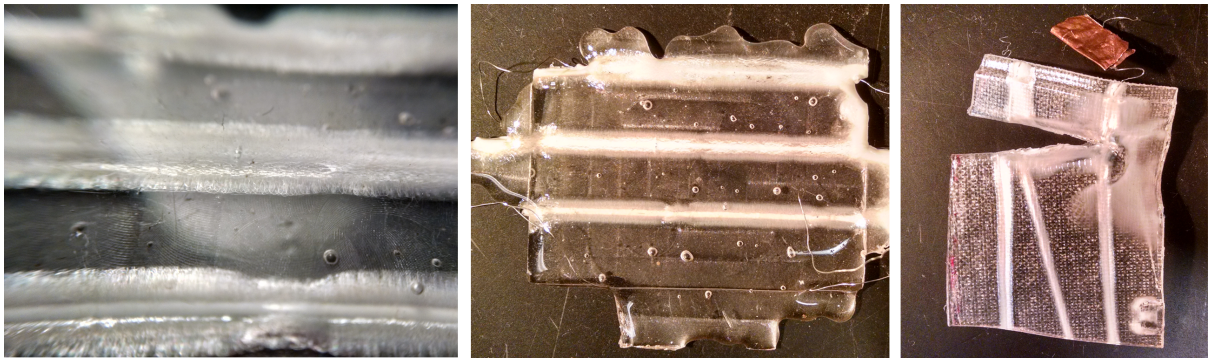

Figure 8.3: Photographs of samples after foaming with a hot embedded Nichrome wire.

After the first experimental test, the cellular structure was found to not be as expected. Figure (8.3) shows photographs of the samples after foaming, and figure (8.4) shows the microstructure. The wire was removed prior to SEM imaging. While there is a radial dependence on cell size, the cell density is much lower than that seen in polycarbonate processed by the conventional hot bath method. Subsequent tests revealed that the polymer was easily melted or burned by the wire and it did not produce the expected varying cellular structure. For these reasons, work on this idea was halted.

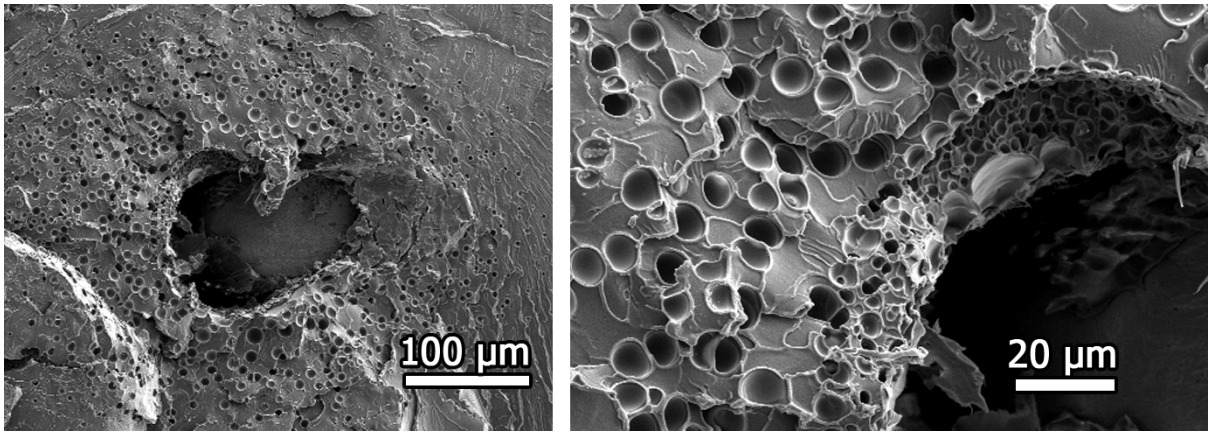

Figure 8.4: SEM images of the cellular structure after foaming with a hot embedded Nichrome wire.

8.2 Time Dependent Saturation Pressure

All previous research in the laboratory involved gas saturation at constant pressure. The proposed question is whether varying the gas pressure (concentration boundary condition) during saturation could yield gas concentration profiles in the polymer that have never been created

before. Since cellular size and structure is dependent on the pre-foaming gas concentration, it was hypothesized that a controllable gradient structure could be produced using this technique.

First, this was posed as a mathematical question as follows: suppose that a polymer sheet of thickness $2L$ is to have a final concentration profile $f(x)$. What must be the concentration boundary condition $c(t)$ be, such that after some time t_1, the concentration profile is $f(x)$ when starting with zero concentration at $t = 0$? This problem was solved numerically.

8.2.1 PDE

The heat-diffusion equation governs diffusion of gasses into solids (in this case, polymers). It has been found that the diffusion coefficient is a scalar function of concentration since the presence of dissolved gas changes the interaction between polymer chains. Thus, the governing equation is

$$\textbf{div}(D\ \textbf{grad}\ u) = \frac{\partial u}{\partial t} \tag{8.4}$$

For a large, uniform sheet of material, only the one dimensional case is considered (through the thickness of the polymer). The equation takes the form

$$\frac{\partial}{\partial x}\left(D\frac{\partial u}{\partial x}\right) = D\frac{\partial^2 u}{\partial x^2} + \frac{\partial u}{\partial x}\frac{\partial D}{\partial x} = \frac{\partial u}{\partial t} \tag{8.5}$$

But $D = D(u)$ so the final equation that must be solved can be written as

$$D(u)\frac{\partial^2 u(x,t)}{\partial x^2} + \left(\frac{\partial u(x,t)}{\partial x}\right)^2 \frac{dD(u)}{du} = \frac{\partial u(x,t)}{\partial t} \tag{8.6}$$

8.2.2 Boundary Conditions

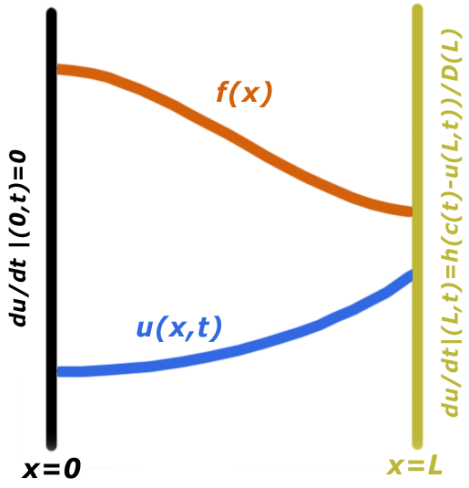

Figure 8.5: Summary of the boundary conditions for the PDE

Let the desired concentration profile in the polymer after absorption be described by $f(x)$. This defines

$$u(x, t_1) = f(x)\ ,\ t_1 > 0\ ,\ x \in [0, L] \tag{8.7}$$

Here, t_1 signifies the diffusion time for this process and is not prescribed but for practical reasons should be minimized. The problem is cut by symmetry, which gives

$$\frac{\partial u}{\partial x}(0,t) = 0 \ , \quad \left.\frac{df(x)}{dx}\right|_{x=0} = 0 \tag{8.8}$$

and the external boundary condition is "convective", namely

$$D(L)\frac{\partial u}{\partial x}(L,t) = h(c(t) - u(L,t)) \tag{8.9}$$

where h is the "mass transfer coefficient", a constant determined experimentally. The function $c(t)$ is what must be found. Physically this is related to the gas pressure outside the polymer, so the constraint is

$$c(t) \geq 0 \ \ \forall \, t \in [0, t1] \tag{8.10}$$

The polymer is initially pure, meaning

$$u(x,0) = 0 \tag{8.11}$$

This fully defines the boundary conditions necessary for a unique solution of the PDE, summarized graphically in figure (8.5). The unknown is the function $c(t)$ which must be found.

8.2.3 Solution Approach

Discretization

Due to the nature of the boundary conditions, the technique applied was the method of lines. Spectral methods were not applicable because the boundary conditions are not known functions of space, but rather they are unknown functions of time, so the boundary condition flexibility of finite difference methods had to be used. Finite element methods could also theoretically be used, but the nature of the problem requires solving for a boundary condition iteratively (explained later) which could not be done programatically with the PDE Toolbox of MATLAB: a script must be developed to perform the search. Using finite difference, the function $u(x,t)$ was discretized in space uniformly and the derivatives approximated by $O(\Delta t^2)$ central difference schemes

$$g'(x) = \frac{g(x+\Delta x) - g(x-\Delta x)}{2\Delta x} \tag{8.12}$$

$$g''(x) = \frac{g(x+\Delta x) - 2g(x) + g(x+\Delta x)}{\Delta x^2} \tag{8.13}$$

Using subscript notation with $\vec{u} = [u_0, u_1, u_2, ..., u_l]$, the boundary conditions are

$$\frac{u_{l+1} - u_{l-1}}{2\Delta x} D_l = h(c(t) - u_l) \tag{8.14}$$

$$u_1 = u_{-1} \tag{8.15}$$

and initial condition

$$\vec{u}(t=0) = \vec{0} \tag{8.16}$$

This expresses the partial differential equation as a system of ordinary differential equations

$$\frac{d\vec{u}}{dt} = \vec{F}(\vec{x}, t) \tag{8.17}$$

Where $\vec{F}$ is the discretized lhs of (8.6). This can be written as linear operators acting on $\vec{x}$ added to the initial conditions in the last term, i.e.

$$\vec{F} = \vec{D} \circ (S\vec{u} + \vec{a}) + \vec{D} \circ (F\vec{u} + \vec{b}) \circ (F\vec{u} + \vec{b}) \tag{8.18}$$

Here, $\vec{a}$ and $\vec{b}$ are vectors with all components zero except for the last which include the convective boundary conditions (8.9), and $\circ$ denotes element-wise multiplication (the Hadamard product), or alternatively $\vec{p} \circ \vec{q} = \mathrm{tr}(\vec{p} \otimes \vec{q})$. S and F are sparse matrices which take second and first order derivatives, respectively:

$$S = \begin{bmatrix} -2 & 2 & 0 & & & & \\ 1 & -2 & 1 & 0 & & & \\ 0 & 1 & -2 & 1 & 0 & & \\ & \cdot & \cdot & \cdot & \cdot & \cdot & \\ & & \cdot & \cdot & \cdot & \cdot & \cdot \\ & & & 0 & 0 & 0 & 0 \end{bmatrix} \tag{8.19}$$

$$F = \begin{bmatrix} 0 & 0 & 0 & & & & \\ -1 & 0 & 1 & 0 & & & \\ 0 & -1 & 0 & 1 & 0 & & \\ & \cdot & \cdot & \cdot & \cdot & \cdot & \\ & & \cdot & \cdot & \cdot & \cdot & \cdot \\ & & & 0 & 0 & 0 & 0 \end{bmatrix} \tag{8.20}$$

Boundary Shooting

A "shooting" style approach was employed to find a solution for $c(t)$ as follows:

- The unknown boundary concentration function $c(t)$ is taken to be a 5th degree polynomial

$$c(t) = \sum_{n=1}^{6} P_n t^{n-1}$$

- Given the "target" function $f(x)$, an error function was constructed which is the minimum of $(f(x) - u(x, t_i))^2$, keeping in mind that $f(x), u(x, t) > 0 \; \forall t > 0$
- A multidimensional nonlinear minimization algorithm was used to find P_n with $u(x, t)$ obtained in each iteration by stepping the discretized PDE to an arbitrary t_2 using the aforementioned initial conditions.

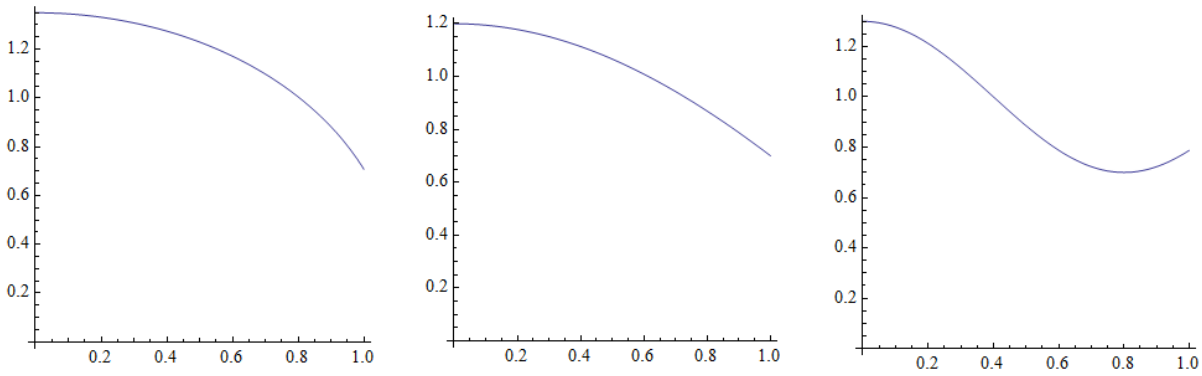

Figure 8.6: Three target functions $f(x)$ which were tested with the code. Left: $f_1(x) = 0.25 + \sqrt{1.1^2 - x^2}$, Center: $f_2(x) = 0.2 + \cos(2\pi x/6)$, Right: $f_3(x) = 1 + 0.3\cos(2\pi x/1.6)$

Implementation

In MATLAB, the function `fminsearch` was utilized to find the polynomial coefficients P_n of $c(t)$ that minimized the error mentioned above. The function requires an initial guess (Pn_0) which is an algorithm parameter `pn0;`. The function was called as follows

```
coeffs{j}=fminsearch(@errorfunc,pn0);
```

Here, `errorfunc` is the function that returns the error, specifically

```
for i=1:tsteps
error(i,1)=norm(U(i,:)-target);
end
minerror=min(error);
```

with U obtained by solving the PDE using `ode15s`. In each step, the boundary condition was computed as $c(t) = \vec{P} \cdot [1\ t\ t^2\ t^3\ t^4\ t^5]$ with $\vec{P}$ (the vector containing the polynomial coefficients) passed to the function by `fminsearch`. The full code is shown in the appendix.

8.2.4 Results

MATLAB Plots and Numerical Results

The method was tested using several target functions as shown in figure (8.6). The diffusion coefficient was taken as the constant function $D(x) = 1$. The execution was speedy, with the code running for less than 15 seconds in most cases on a second-gen Intel Core i5 processor. The controllable parameters were run time t_2, number of time steps evaluated `tsteps`, and initial guess $\vec{P}_0$, and the effects of each were studied in the examples shown in figures (8.7),(8.8), and (8.9). The target functions are shown in red, while the blue curves are solutions of $u(x,t)$ for all values of $t \in tspan$ as described in the captions. The raw numerical results are shown below:

Example 1 shown in figure (8.7):

```
Elapsed time is 13.762998 seconds.
Elapsed time is 14.229944 seconds.
```

```
Elapsed time is 14.456324 seconds.
Elapsed time is 14.091880 seconds.
>> [error1 error2 error3 error4]
    0.0943    0.0213    0.1097    0.7175
```

Example 2 shown in figure (8.8):

```
Elapsed time is 14.655981 seconds.
Elapsed time is 16.785839 seconds.
Elapsed time is 13.093053 seconds.
Elapsed time is 34.401956 seconds.
>> [error1 error2 error3 error4]
    0.0238    0.0244    0.1225    0.6981
```

Example 3 shown in figure (8.9):

```
Elapsed time is 15.639072 seconds.
Elapsed time is 20.878540 seconds.
Elapsed time is 46.488626 seconds.
Elapsed time is 63.146409 seconds.
>> [error1 error2 error3 error4]
    0.0679    0.0520    0.0031    0.1396
```

By comparing `error1` and `error2` in the examples, it is seen that the accuracy is not affected strongly by the initial guess provided to the function `fminsearch` except when the integration time was increased (and thus ΔT increased proportionally). This says that the minimization function works best when there are more time points (therefore more curves) to check as possible matches to the target function. Example 3 shows this effect, with very accurate results for $f_3(x)$ and $f_4(x)$. However, this will in turn increase computation time since the error must be evaluated for the solution $u(x,t)$ at each time point and for each iteration carried out by the function `fminsearch`. The effects can be seen by comparing the timing results given by `tic toc` above. Also it is important to note that the target $f_4(x)$ (the short wavelength cosine) was unable to be matched by any $u(x,t)$ in examples 1 and 2, which is expected since the function changes concavity on the domain $x \in [0,1]$ and demonstrates a limitation of the algorithm. Example 3 does successfully provide a $u(x,t) \approx f_4(x)$, however in order to achieve it, the function $c(t)$ and $u(x,t)$ were negative for some values of t, which violates the constraint in equation (8.10). Therefore, care must be exercised when choosing t_2, `tsteps`, and `pn0` to ensure that the constraints are satisfied in order to provide physically meaningful results.

The algorithm proved to be very effective and robust in finding a function of time to be used as boundary conditions for the outside concentration that yields a desired concentration profile in the polymer. Since the target profiles desired for the manufacturing process research are well behaved (constant concavity and smooth), the numerical method presented here is well suited for the application despite its demonstrated limitations. Although a finite-difference method was utilized, the computation time is acceptable since the PDE is 1-dimensional in space. This will prove very useful in running optimization scripts for optimizing the manufacturing process being researched. Numerical methods have done a remarkable job here where analytic methods are not applicable.

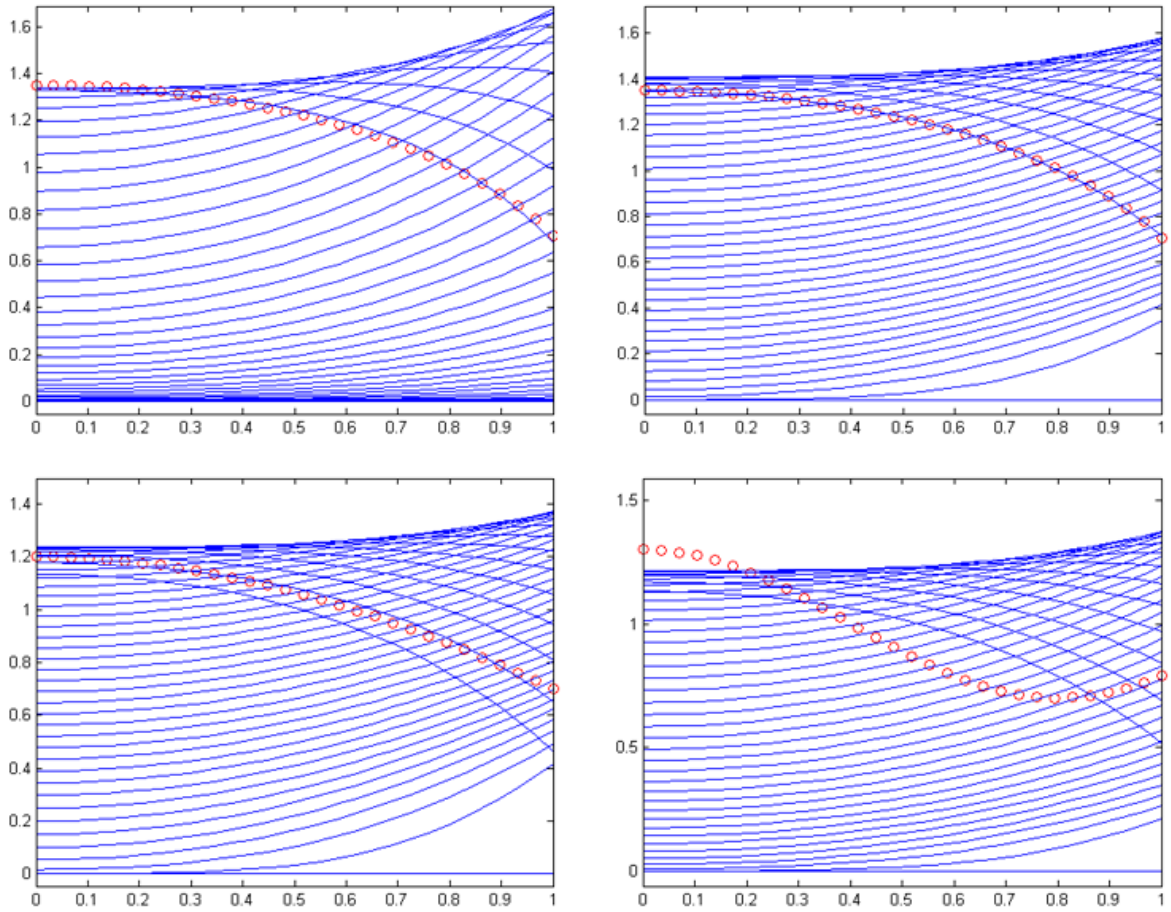

Figure 8.7: Algorithm results for total integration time $t_2 = 2$, and `tsteps=40`. **Top left:** target=$f_1(x)$, $\vec{P}_0 = [1\ 1\ 1\ 1\ 1\ 1]/10$ **Top right:** target=$f_2(x)$, $\vec{P}_0 = [1\ 1\ 1\ 1\ 1\ 1]/10$ **Bottom left:** target=$f_3(x)$, $\vec{P}_0 = [1\ 1\ 1\ 1\ 1\ 1]/5$ **Bottom right:** target=$f_4(x)$, $\vec{P}_0 = [1\ 1\ 1\ 1\ 1\ 1]/10$

8.2.5 Map Between Cell Size and Gas Concentration

The previous section describes how to compute a boundary condition, as function of time, that will yield a desired concentration profile after some time. However, the controllable parameter is pressure rather than concentration in the polymer. These two can be trivially linked by using the polymer's gas solubility which maps saturation pressure to concentration in the polymer. This is readily available from previous research.

The end goal is to produce a desired gradient cell structure. The problem must be approached in the following order:

1. Declare the desired cell size as function of material space $d(x), x \in [0, L]$.

2. Compute the necessary polymer gas concentration $c(x)$ that will yield cell structure $d(x)$ after foaming.

3. Compute the boundary condition $\hat{c}(t)$ that will yield polymer gas concentration $f(x)$ after some time t_1.

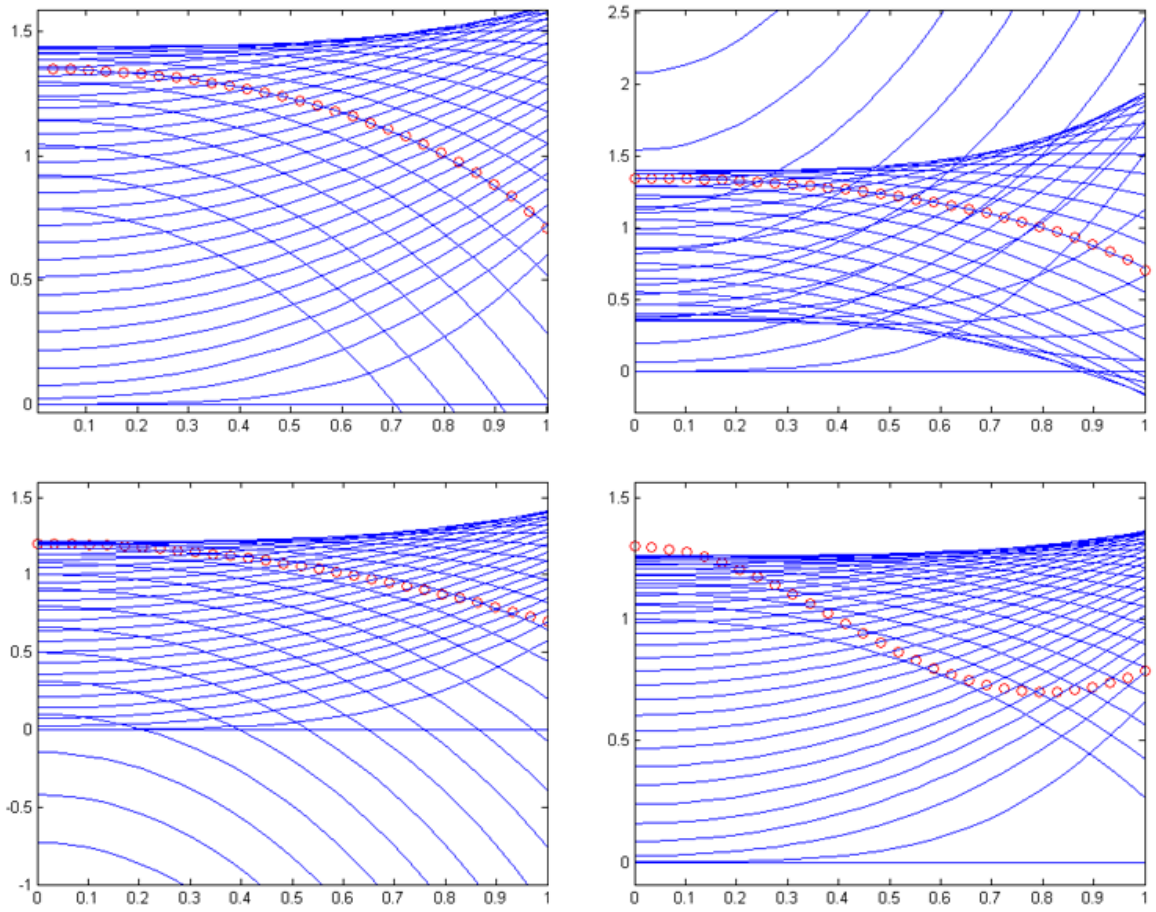

Figure 8.8: Algorithm results for total integration time $t_2 = 1$, and `tsteps=40;`. **Top left:** target=$f_1(x)$, $\vec{P}_0 = [3\ 3\ 3\ 3\ 3\ 3]/10$ **Top right:** target=$f_2(x)$, $\vec{P}_0 = [3\ 3\ 3\ 3\ 3\ 3]/10$ **Bottom left:** target=$f_3(x)$, $\vec{P}_0 = [3\ 3\ 3\ 3\ 3\ 3]/5$ **Bottom right:** target=$f_4(x)$, $\vec{P}_0 = [3\ 3\ 3\ 3\ 3\ 3]/10$

4. Compute the necessary saturation pressure $p(t)$ that will place boundary condition $\hat{c}(t)$ on the polymer.

The procedure for performing step 2 above is now given. Previous research has shown that cell size is a strong function of gas concentration. Let q be a one-to-one function that maps gas concentration to cell size. Then,

$$d(x) = q(c(x)) \tag{8.21}$$

Finding q can be done experimentally as follows: saturate a polymer sample of known thickness and then allow it to desorb for an arbitrary amount of time, then foam at the desired temperature. Since the desorption time is known, the concentration profile $c(x)$ of the sample can be computed by solving the PDE with the initial condition $c(x, 0) = c_0$ and boundary condition $c(L, t) = 0$. After foaming, SEM images can be used to measure the cell size at various points, and thus obtain data for $d(x)$. Then, q is expanded into its Taylor series

$$q(c) = \sum_{n=0}^{N} a_n c^n \tag{8.22}$$

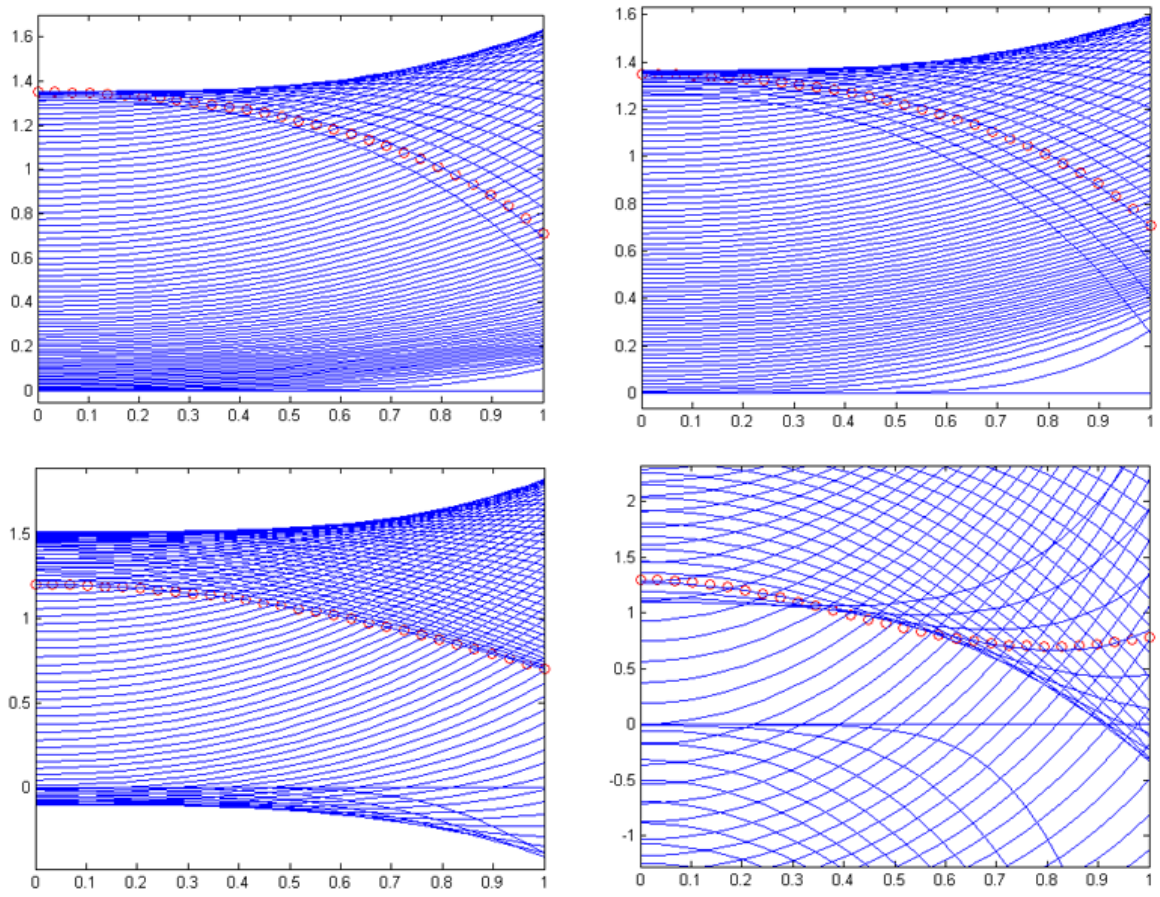

Figure 8.9: Algorithm results for total integration time $t_2 = 1$, and `tsteps=80;`. **Top left:** target=$f_1(x), \vec{P}_0 = [1\ 1\ 1\ 1\ 1\ 1]/10$ **Top right:** target=$f_2(x), \vec{P}_0 = [1\ 1\ 1\ 1\ 1\ 1]/10$ **Bottom left:** target=$f_3(x), \vec{P}_0 = [1\ 1\ 1\ 1\ 1\ 1]/5$ **Bottom right:** target=$f_4(x), \vec{P}_0 = [1\ 1\ 1\ 1\ 1\ 1]/10$

where N must be less than or equal to the number of data points obtained for $d(x)$ using the SEM. The coefficients a_n can be trivially computed by linear regression.

This method was attempted experimentally, and the resulting SEM could be seen in figure (8.10). The result is unexpected, since the concentration varied from 0 to 0.11 mass fraction (saturation concentration at 5.0 MPa CO_2 pressure). The expected range of cell sizes was 100 nm to 1 μm, however, all the cells in the SEM have diameters on the same order of magnitude. The hypothesis of why this occurred is as follows: the diffusion coefficient (mass diffusivity) $\mathcal{D}$ has dimensions $[L/T^2]$ and increases rapidly with temperature. The thermal diffusivity $\alpha = k/\rho c_p$ also has dimensions $[L/T^2]$. Therefore, the quotient $\mathcal{D}/\alpha$ is a non dimensional number that describes the ratio of mass transfer to heat transfer. For PEI, $\mathcal{D} \approx 10^{-8}$ and $\alpha \approx 10^{-7}$ at room temperature. However, this ratio increases with temperature, meaning that what was once a concentration profile $c(x)$ will now be smoothed out to an approximately constant value, i.e.

$$\frac{1}{c}\frac{dc}{dx} << 1 \tag{8.23}$$

This is precisely what is seen in figure (8.10) which shows a rather even cell structure even though the polymer gas concentration $c(x)$ at heating time was highly uneven.

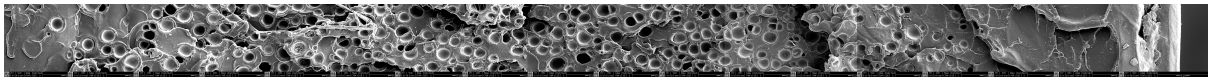

Figure 8.10: Digitally stitched SEM images of a 1.0 mm PEI sample foamed after 20 minutes of desorption.

8.2.6 Experimental Apparatus

An apparatus was designed and built to carry out the aforementioned experiments. The system was built using parts in the laboratory. A photograph, mechanical diagram, and circuit diagram are given in figures (8.11 - 8.13). It has the capability of controlling pressure and temperature inside a pressure vessel by following target values from an SD card in .csv format. The file must contain information in the following order: time (milliseconds), pressure (0 - 1000 psi mapped 0 - 1023), and degrees (Celsius). A MATLAB script was written to generate these files based on simple inputs from the user, including a function of time. This script, along with the microcontroller code are given in the appendix.

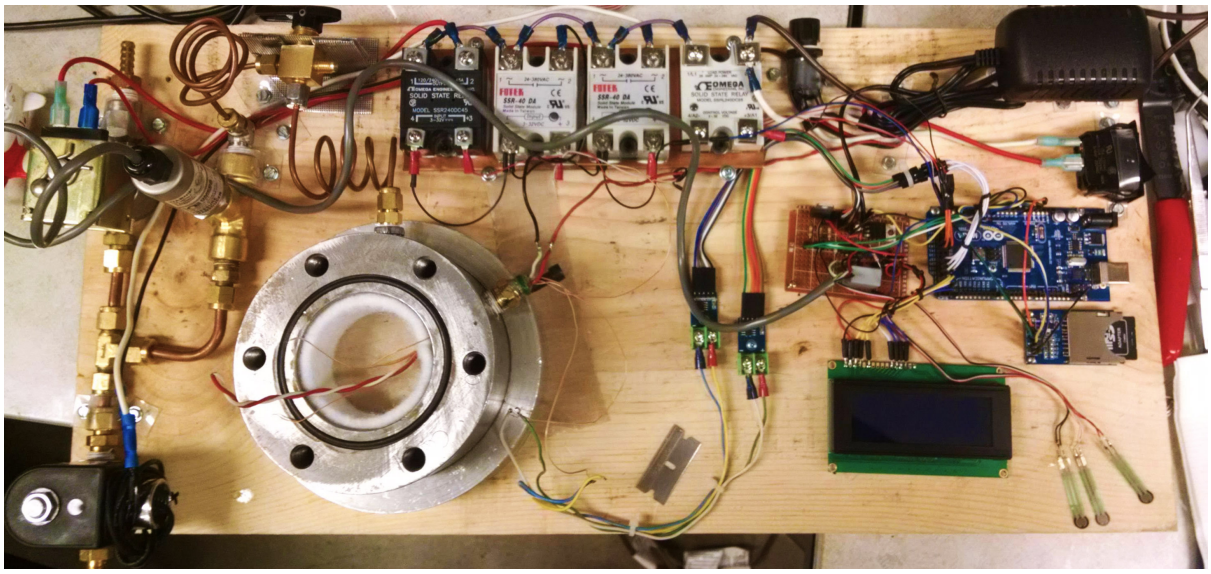

Figure 8.11: Custom made control system for regulating gas pressure and temperature as a function of time.

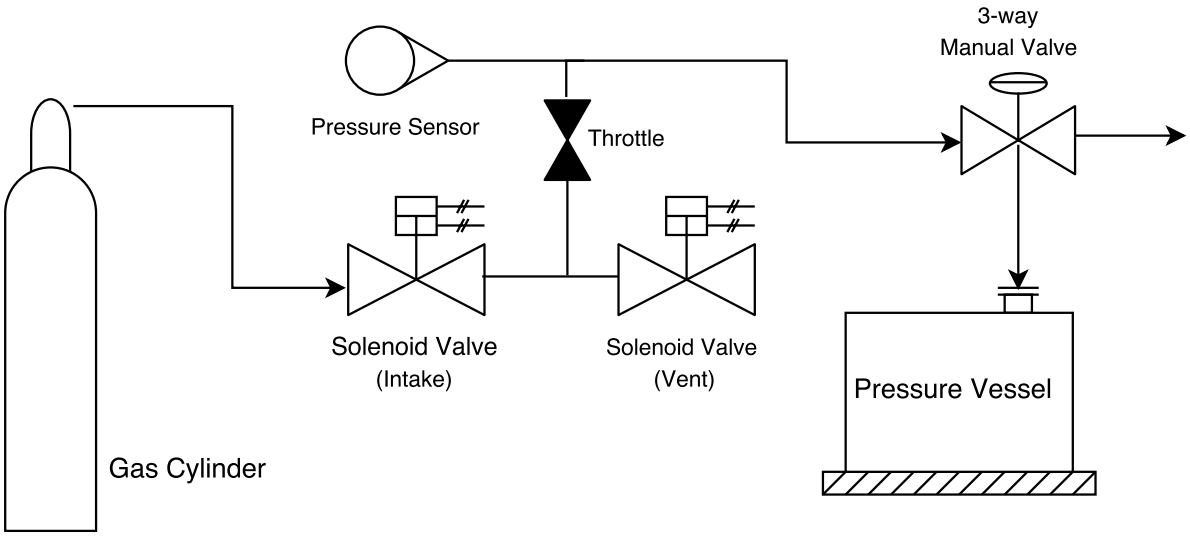

Figure 8.12: Diagram of the gas system.

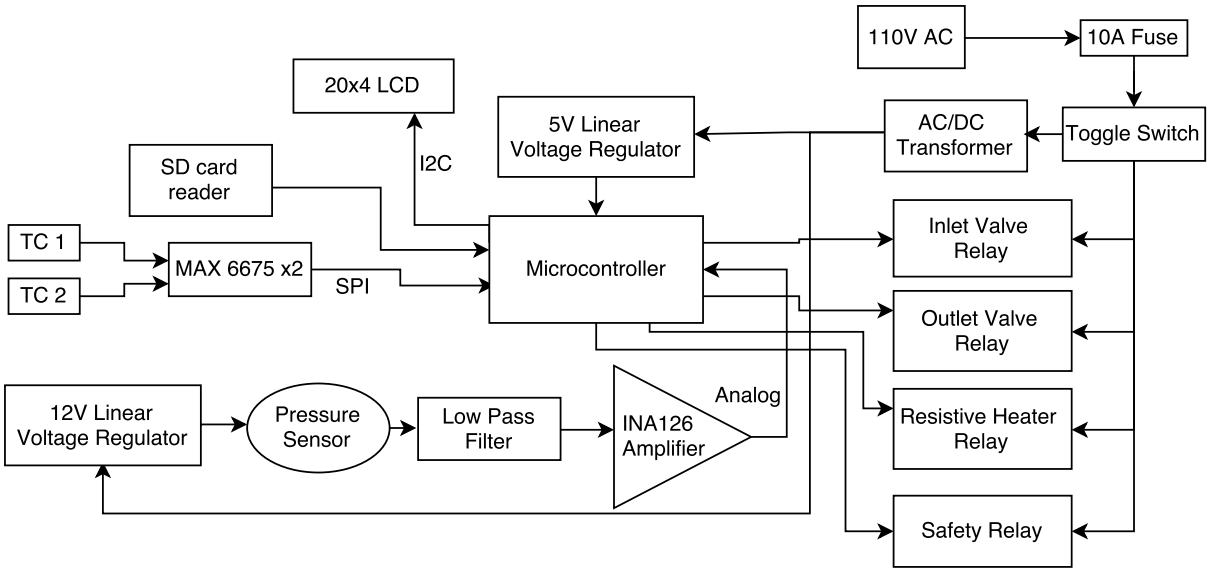

Figure 8.13: Diagram of the system's electronics.

8.3 Impact Strength of Liquid Filled Cells

Filling a microcellular structure with a nearly-incompressible fluid (such as water) will greatly increase its compressive strength since the liquid can support a large fraction of the load. This can be most easily demonstrated using figure (8.14). Force balance in the vertical direction of a central cell gives

$$\sigma t + pw = L \tag{8.24}$$

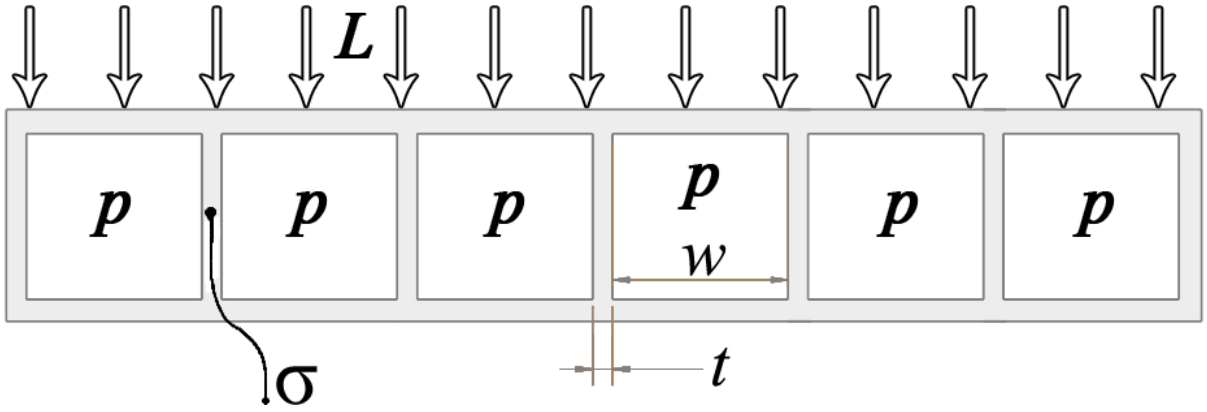

Figure 8.14: Cell array filled with a fluid. L is an applied load of uniform distribution with dimensions [Force / Length], and p is the pressure of the fluid.

Since p and L are arbitrary, the stress in the cell wall σ can be positive, negative, or zero. This seemingly counter-intuitive fact, that a load can be supported by fluid with zero stress in the solid wall, is similar to that of a car tire filled with air.

The challenge lies in filling closed-cell foams with a liquid such as water. One idea was to diffuse the liquid through the polymer cell wall by submerging the foam sample in water and applying pressure. Since polymers such as PMMA are hydroscopic (meaning they absorb water), in theory the liquid should fill the cells, although at a slow rate. This was attempted for several PMMA and PET samples. It was found that increasing the water pressure above 2.0 MPa would cause the cells to collapse, since the foam's macroscopic compressive strength was exceeded. However, even 2.0 MPa was not sufficient to show any signs of liquid filling after 14 days under pressure. Figure (8.15) shows the unchanged structure. Density measurements showed no significant increases after the process.

Since diffusivity increases with temperature, an attempt was made to heat the samples while submerged under pressure in order to expedite the water absorbtion process. This was found to be ineffective even with temperatures approaching the glass transition of the polymer, at which point the structure collapsed due to softening and lowering the mechanical strength.

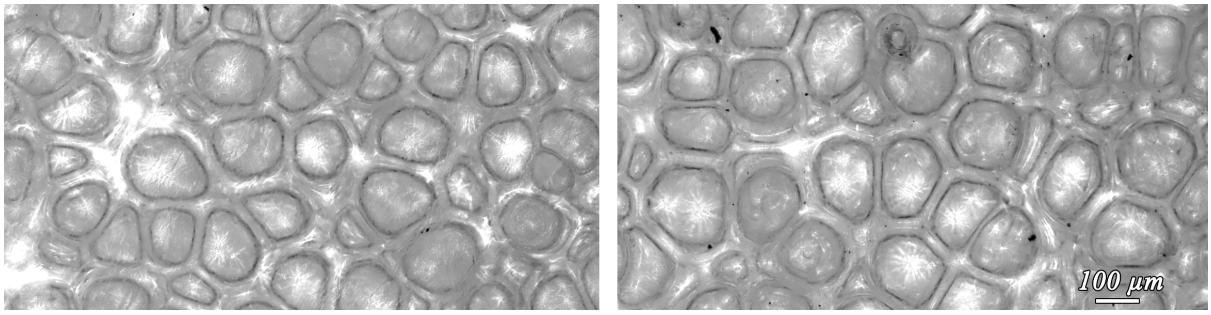

Figure 8.15: Left: PET sample right after foaming. Right: Same sample after 14 days submerged in water at 2.0 MPa.

8.4 Laser Induced Foaming

Since nucleation begins when the gas-saturated polymer reaches its glass transition temperature, one method to quickly reach this temperature is by exposing a sample to a high energy laser beam. PEI was chosen because of its naturally yellow tint which absorbs the laser energy more effectively than clear polymers such as PC and PMMA. The samples used in the experiments presented here were 1.0 mm thick sheets saturated at 5.0 MPa.

The main motivation to explore foaming using a laser as the heat source is the precise spatial control of where nucleation happens. Since the laser beam can be focused to a very small spot using optics, it was hypothesized that a single bubble could be nucleated which is of great interest in the field of nucleation theory. Additionally, new cellular structures could be fabricated where only a desired portion of the polymer is foamed. Figure (8.16) shows concepts of selectively placed bubbles. Since the cellular structure in PEI was previously observed to be open-porous, one idea was to create porous channels embedded in the polymer, acting as a microfluidic device.

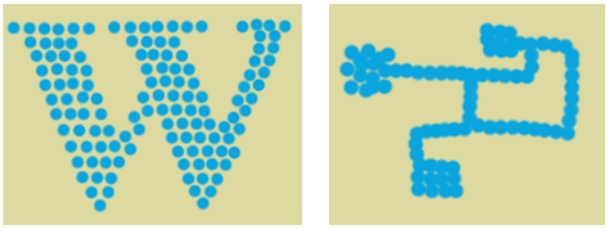

Figure 8.16: Concept of selectively nucleating bubbles in a polymer sample. Left: UW logo, Right: porous channels for fluid flow.

PEI samples saturated at 5 MPa were mounted a motorized stage. A 532 nm, 1 W Nd:YAG pulsed laser was used along with focusing lenses that reduced the spot size to a Gaussian intensity distribution of 200 μm characteristic diameter. The laser was triggered to fire by the rising edge of a voltage square wave produced by a function generator. The duration of each pulse is 450 picoseconds.

Experiments at several values of firing frequency, ND filter, and stage speed were performed. Figure (8.18)

As the frequency is increased, the spacing between the bubble centers decreases and eventually the bubble boundaries touch. Although bubbles are seen using the optical microscope, it is

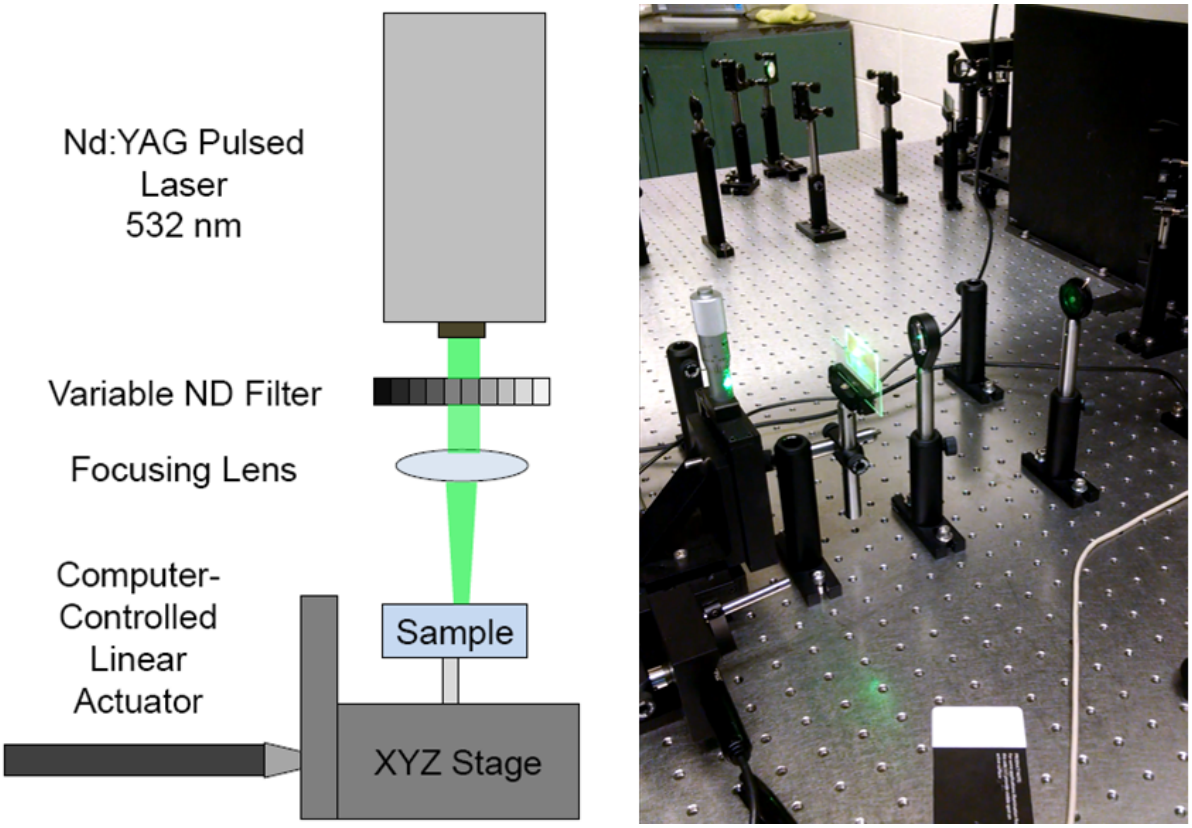

Figure 8.17: Schematic and photograph of the laser setup.

unknown what the resulting microstructure is and how deep inside the specimen these bubbles are. Therefore, SEM images were taken and shown in figure (8.19).

The SEM images show some unexpected behavior: the bubbles are formed on the top face of the sample and protrude outward - a behavior never before seen in the microcellular process. Also, the top right image in figure (8.19) clearly shows a through hole in the bubble's surface. It is probable that the gas escaped through this hole after diffusing out of the solution.

Next, the frequency was increased by an order of magnitude in hopes of creating interconnected channels rather than individual bubbles. While a channel was visible on the optical microscope, the SEM showed that trough shape was cut into the surface by the laser. The bottom right image in figure (8.20) shows the channel on surface rather than at the midplane of the sample where the laser beam was focused. Additionally, the microstructure on the wall of this channel resembles the nanoporous open-cell one seen in conventional foaming of PEI. This result is highly significant since it is the first time that this structure was observed on the surface of a sample without the solid skin. Whether it indeed is the same structure and not a mere resemblance is a question yet to be answered.

Since the focused laser beam appeared to vaporize or otherwise annihilate the polymer at the surface, the next experiment was to remove the focusing optics and use the beam as it comes from the laser. Since the laser spot size is 1 mm as opposed to 200 μm when using optics, the heating was much slower and it required several seconds of continuous firing at the maximum rate of 50 kHz in order to reach the glass transition temperature and begin cell nucleation.

Although the samples used in this experiment were identical to the last (PEI saturated at 5

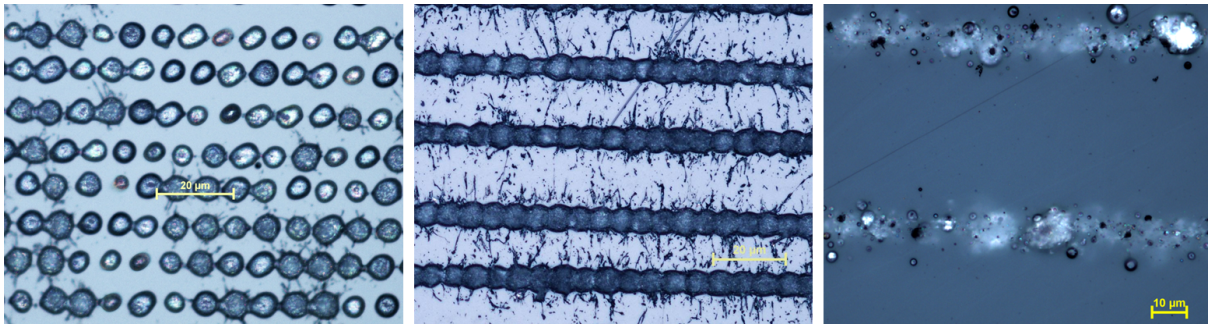

Figure 8.18: Optical microscope images of the resulting bubble structure created by the laser. Left: 100 Hz, 2 mm/s stage speed, no ND filter. Center: 100 Hz, 1.5 mm/s stage speed, no ND filter. Right: 100 Hz, 2 mm/s stage speed, 50% ND filter. In all cases, the beam was focused on the sample's midplane using a 50 mm focal length lens.

MPa), the cell structure created by this method is altogether different. Figure (8.21) shows this unique structure wherein bubbles are nucleated in a scattered fashion, most being several diameters apart from its neighbors. By varying the focal plane of the microscope, it was observed that the structure is not planar but continues many cell diameters into the depth. Since these cells were all below the surface, SEM analysis was not performed on these samples.

The key observations from figure (8.21) are:

- The cells are large, approximately 20-50 μm.
- The spacing between cells is large, approximately 1/2 - 10 diameters.
- There are interesting features visible in the bubbles, most notably a dot in the center (looking like a nucleus or a gas escape hole) and a color change as a function of radial distance form the center.
- There is an apparent burn mark where the laser intensity is highest, and the cell density is slightly larger in this region.

In order to study how this structure develops in time, multiple samples were foamed all with different laser incidence time. This was achieved by requesting various numbers of pulses from the function generator at 50 kHz. Figure (8.22) shows the increasing nucleated area as the number of pulses was increased.

An interesting observation was made by looking at the laser pattern on the wall behind the samples. An interference pattern was seen as the sample nucleated cells and scattered the light. Figure (8.23) shows several time snapshots of this pattern projected onto a sheet of paper behind the sample.

These experiments raised the following questions:

- Why is cell structure different when heating by laser vs. conduction?
- Is there a hierarchical cell structure? (bubbles within the bubbles)
- What is the effect of time/energy flux on the nucleation/cell growth process?

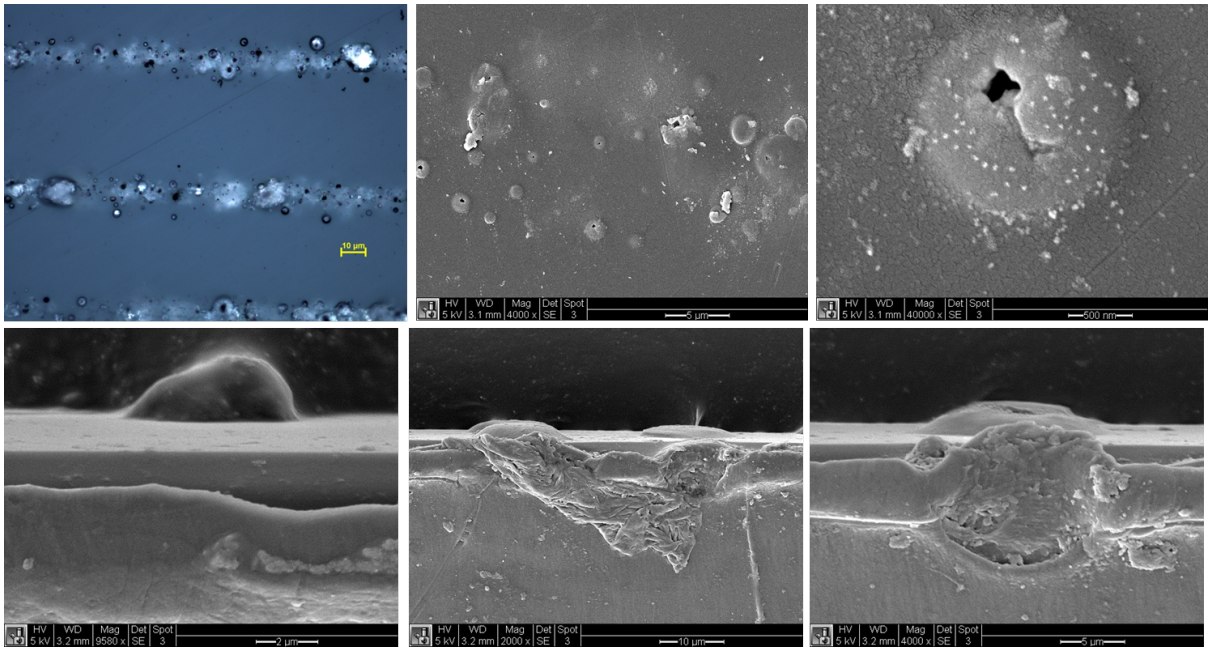

Figure 8.19: SEM images of the bubbles (except top left). Top row: top view of the sample. Bottom row: cross sectional view by freeze fracturing in liquid nitrogen.

The challenges in exploring these questions are:

- Imaging.
- Monitoring Temperature versus time/space.
- Observing real-time bubble evolution.

These results were presented at ASME McMat conference in 2015, and work was not continued further.

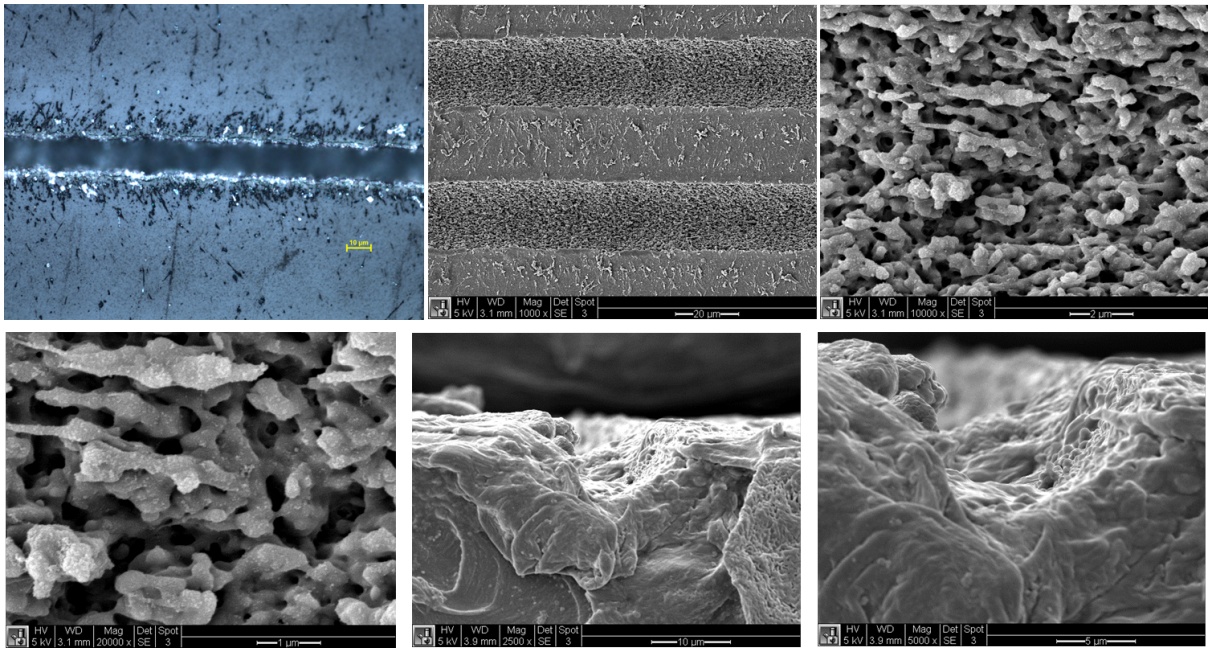

Figure 8.20: SEM images of the channels (except top left). The laser was fired at 1 kHz, stage speed was 2 mm/s, and a 63% ND filter was used.

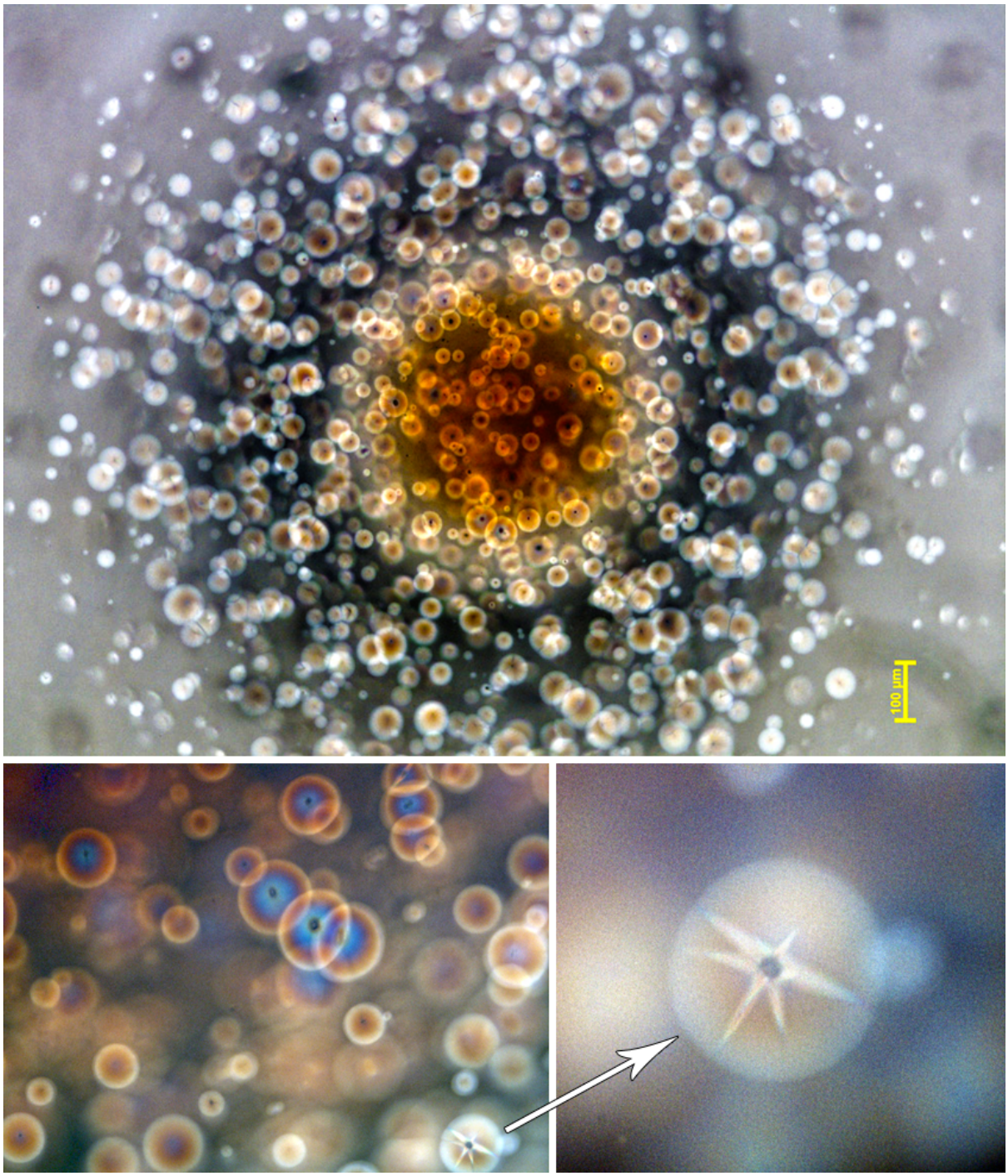

Figure 8.21: Optical microscope images of the structure created by the laser with no focusing optics. There was no ND filter, and the laser was fired at 50 kHz for 5 seconds.

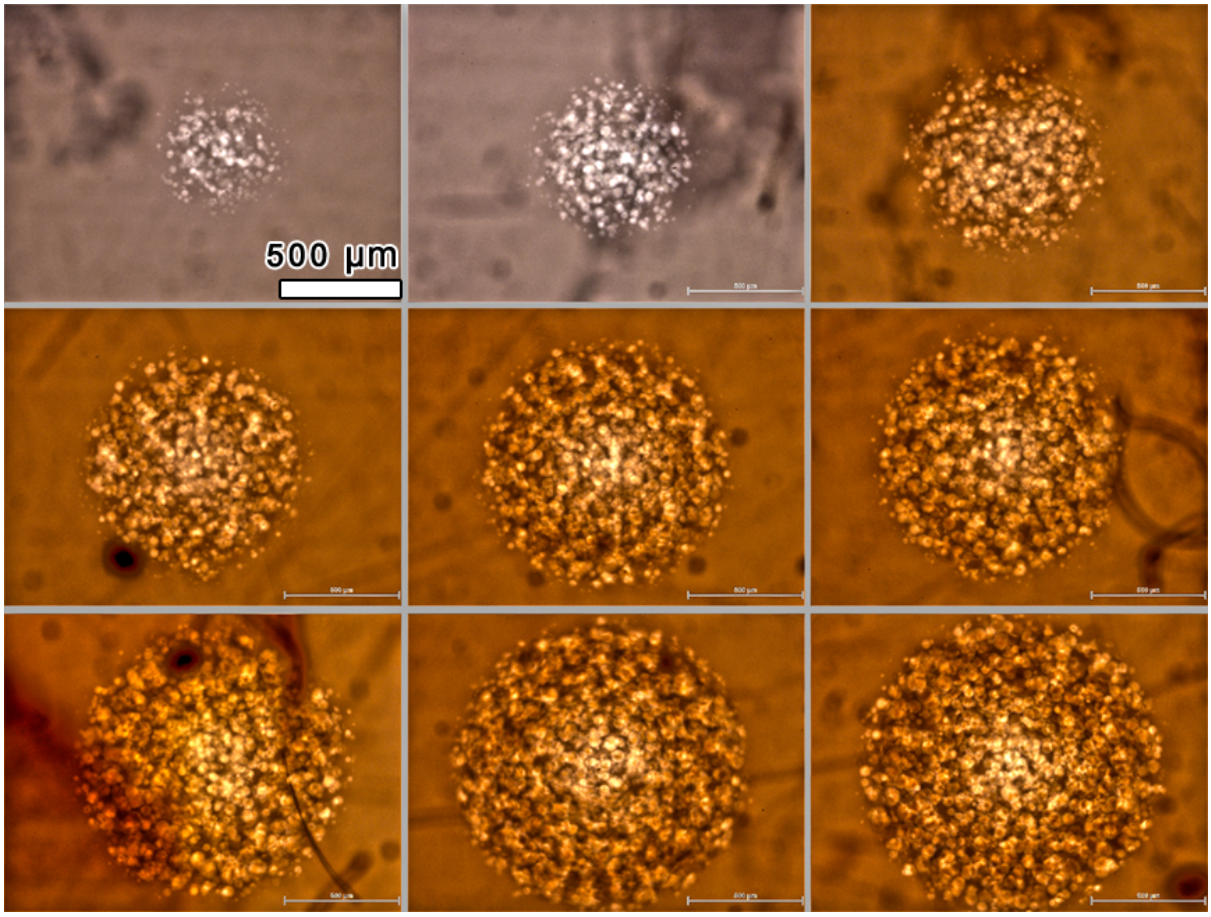

Figure 8.22: Laser fired at 50 kHz for different number of pulses in 10,000 increments. No lens or ND filter was used. Top left: 80,000 pulses. Bottom right: 160,000 pulses.

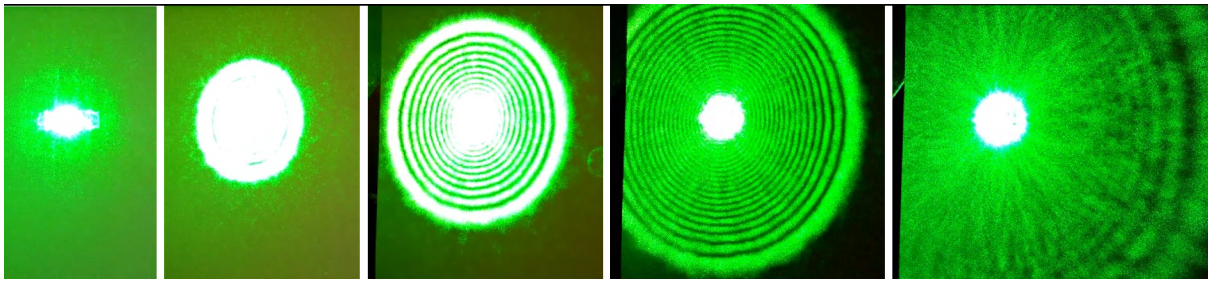

Figure 8.23: Time snapshots of the laser light interference pattern on the back wall of the room, due to scattering off the nucleated cells.

Chapter 9

3D Printing Bubbles

9.1 Amazon Catalyst Grant

A program was recently started at the University of Washington by Amazon in collaboration with CoMotion (the UW technology transfer office) to sponsor projects proposed by UW students, faculty or staff. I applied for the fund with an idea of 3D printing air-filled polymeric bubbles out of a molten thermoplastic, then depositing these bubbles individually to build a 3D object. Below is the verbatim description submitted in the grant application.

> We propose a technology that is able to 3D print air-filled plastic bubbles, thus creating products that are up to 90% air and only 10% plastic. Our proposed printer will have an array of many bubble-shooting nozzles which make printing quick enough for on-demand use. The end product will also be recyclable since the foam is expanded with pure air.
>
> The printer is novel and will "spit out small bubbles (molten plastic filled with air, similar to the soap bubbles that kids blow except much much smaller). These bubbles will flow down and deposit to form a 3D structure, just as snow falls and builds up. The end result is a lightweight, insulative, cushioning, flexible, printed product.
>
> The project is to develop a prototype nozzle system that reliably and quickly spits these bubbles. Once this is constructed, an array of these nozzles could easily be integrated into existing printing platforms to create a complete 3D printer.

The awarded grant period was from February 23, 2017 until November 23, 2017. I began by researching the science of soap bubbles using two books [40, 41]. I also drew inspiration from microfluidics research, most of which creates droplets but some also have made bubbles or two-phase droplets of immiscible liquids [42]. A chemical engineering master's student previously attempted to created microfluidic droplets out of thermoplastic polymer melts [43]. There are two relevant dimensionless parameters in droplet formation - the first is the *capillary number* Ca defined as

$$Ca = \frac{\mu U}{\gamma} \tag{9.1}$$

where γ is the surface tension, μ the dynamic viscosity, and U the characteristic velocity. This number characterizes the ratio of viscous forces to those of surface tension. The second is the

Bond number Bo defined as

$$Bo = \frac{\Delta\rho g L^2}{\gamma} \tag{9.2}$$

where $\Delta\rho$ is the difference in density of the droplet and surrounding fluid and L is a characteristic length. This number characterizes the ratio of gravitational force to surface tension. When 3D printing droplets, gravity acts to break off the bubble and accelerate it away from the nozzle.

With this information in mind, the first step was to create a nozzle that jets soap bubbles, since it was the simplest first step. After learning of the bubble making process in this robust soap-water system, the challenge was to extend the design to one that would produce bubbles out of polymer melt or other materials.

9.2 Apparatus

First, an apparatus was designed and built that was used as the main platform for all experiments. The functional requirements of this system were:

- Provide an adjustable pressure source to the inner needle (air) and liquid reservoir.
- Provide an adjustable air flow through the outer channel of the nozzle.
- Obtain pressure readings of the two aforementioned channels.
- Control on/off valves of all inputs.
- Heat the printable substance to a specified temperature.
- Adjust the position of the needle within the nozzle.
- Acquire high magnification images of the nozzle tip in real time.

The apparatus is shown in figure (9.1). The components were mounted on a half-inch thick fiberboard using self-threading screws.

The camera used is a Blackfly-S from FLIR. This camera connects via USB3 and is capable of capturing 1280 x 1024 pixel monochrome images at 170 frames per second. The exposure can be configured between 6 microseconds and 30 seconds. The camera also has a hardware trigger.

Off the shelf turn-screw pressure regulators from McMaster-Carr were used to control the air pressure in the system. To have software control over pressure, BYJ48 stepper motors were mounted, using 3D printed connections, to turn the regulators' screws. The setup is shown in figure (9.2).

Shop air at 100 psi was used as the main pressure source. The main regulator took this down between an adjustable 0.5 - 30 psi. The remaining line was the feed to the two 0.5 - 10 psi regulators which provided compressed air to the liquid reservoir and needle. DC solenoid valves served as on/off switches. Analog sensors probe the pressure at 3 locations in the system. Figure (9.3) shows a diagram of the fluid system in the apparatus.

The electrical system revolves around the microcontroller (MCU), an ATmega2560 at 16MHz. This processor is on an Arduino MEGA platform. Communication with the PC was via USB

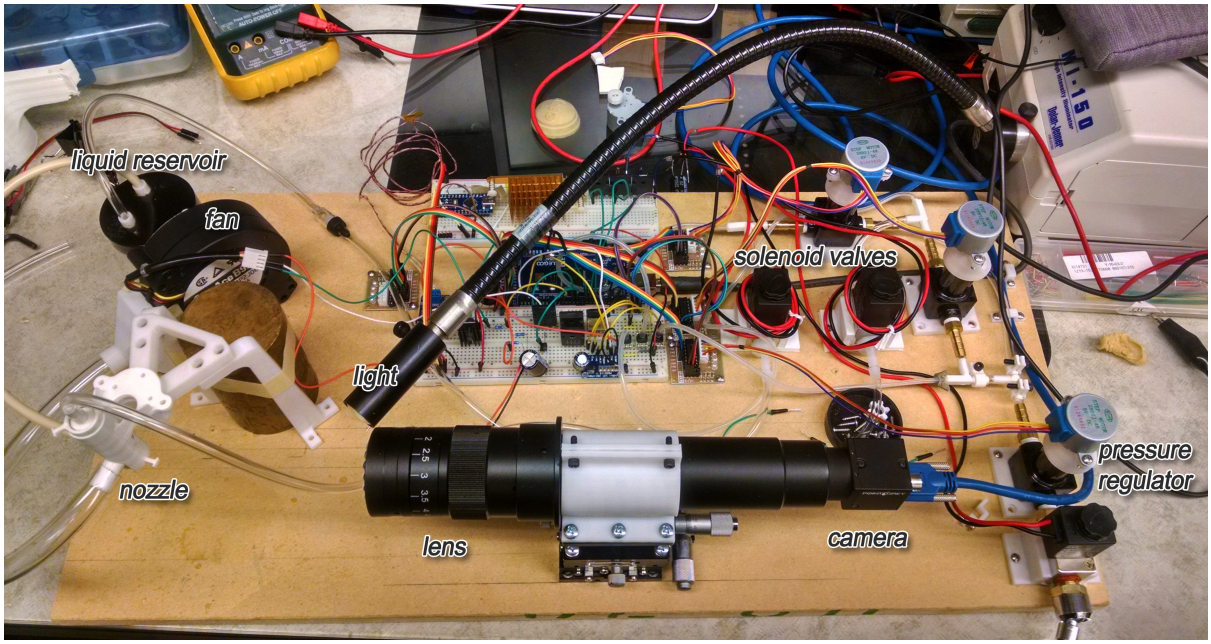

Figure 9.1: Photograph of the experimental apparatus for the 3D printing project. The nozzle is on the bottom left.

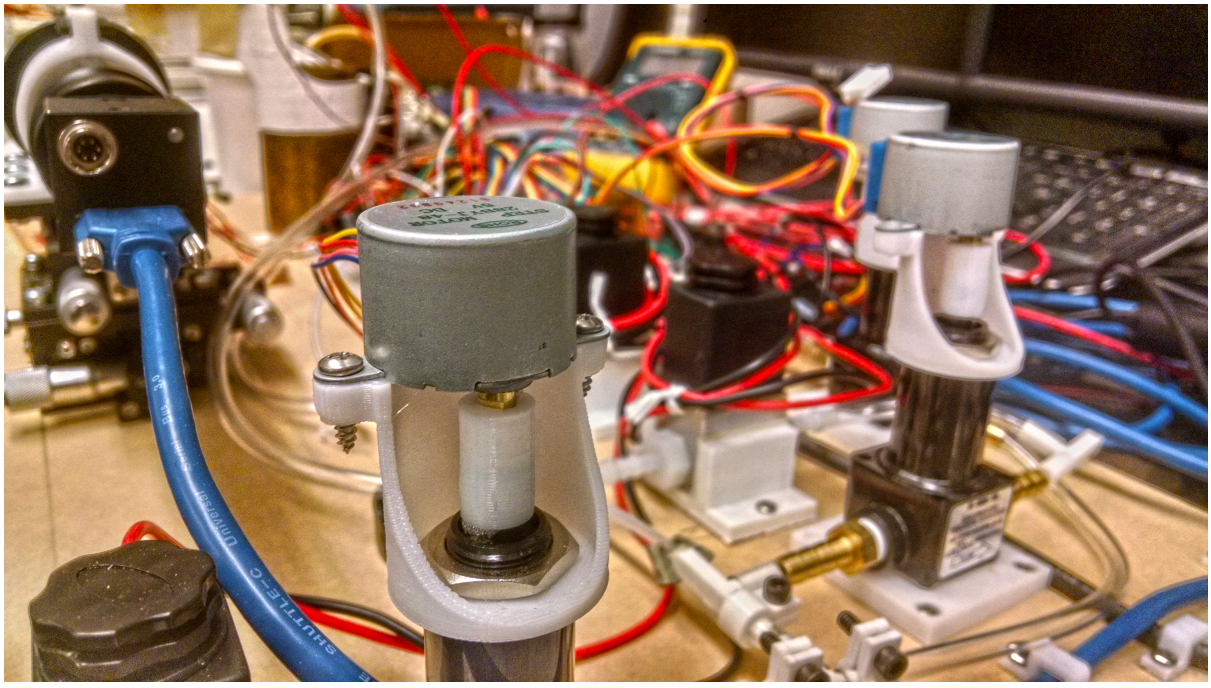

Figure 9.2: Stepper motors attached to the pressure regulator turn screw to enable pressure control through software.

at a baud rate of 115200. The temperature sensor is a K-type thermocouple read using the MAX31855 amplifier which includes an on-board cold-junction compensator and interfaces using Serial Peripheral Interface (SPI). An HP 12V brushless DC fan was used to provide air flow to the outer channel of the nozzle. The fan speed is controlled via pulse-width modulation

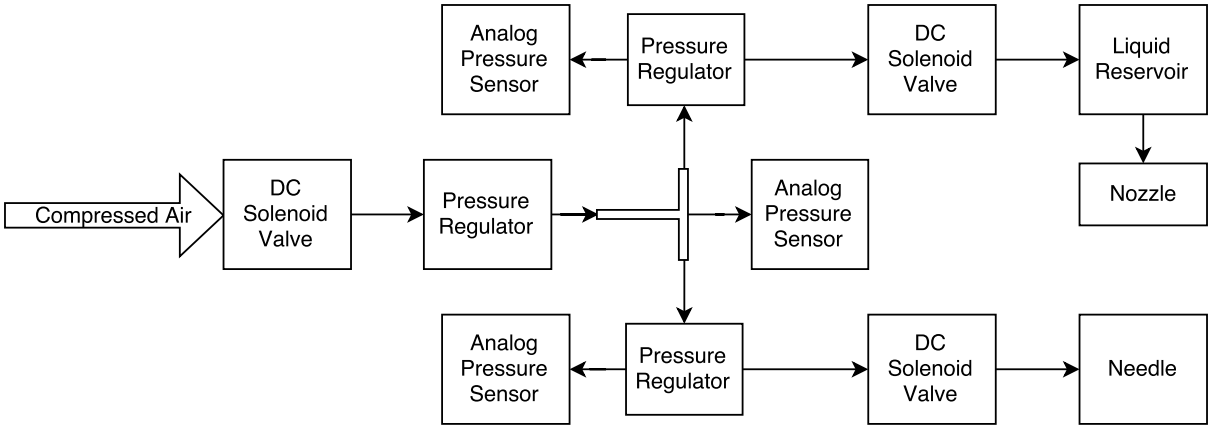

Figure 9.3: Block diagram of the compressed air system.

(PWM) at 25 kHz. The MCU's timer registers have to be re-written at setup in order to change the PWM frequency to the required 25 kHz. The stepper motors were driven using ULN2003 drivers which have a Darlingon transistor array to drive the stepper's coils. The potentiometer and pressure sensors were read using the ADS 1115, a 16-bit analog-to-digital (A/D) converter which interfaces using I2C. The MCU's digital output pins were wired directly to MOSFET gates and relay input pins to control the LED and heaters/solenoid valves, respectively. Figure (9.4) illustrates the electrical system in block-diagram format, not circuit diagram. A bench power supply was used to deliver 12 VDC as the main power source.

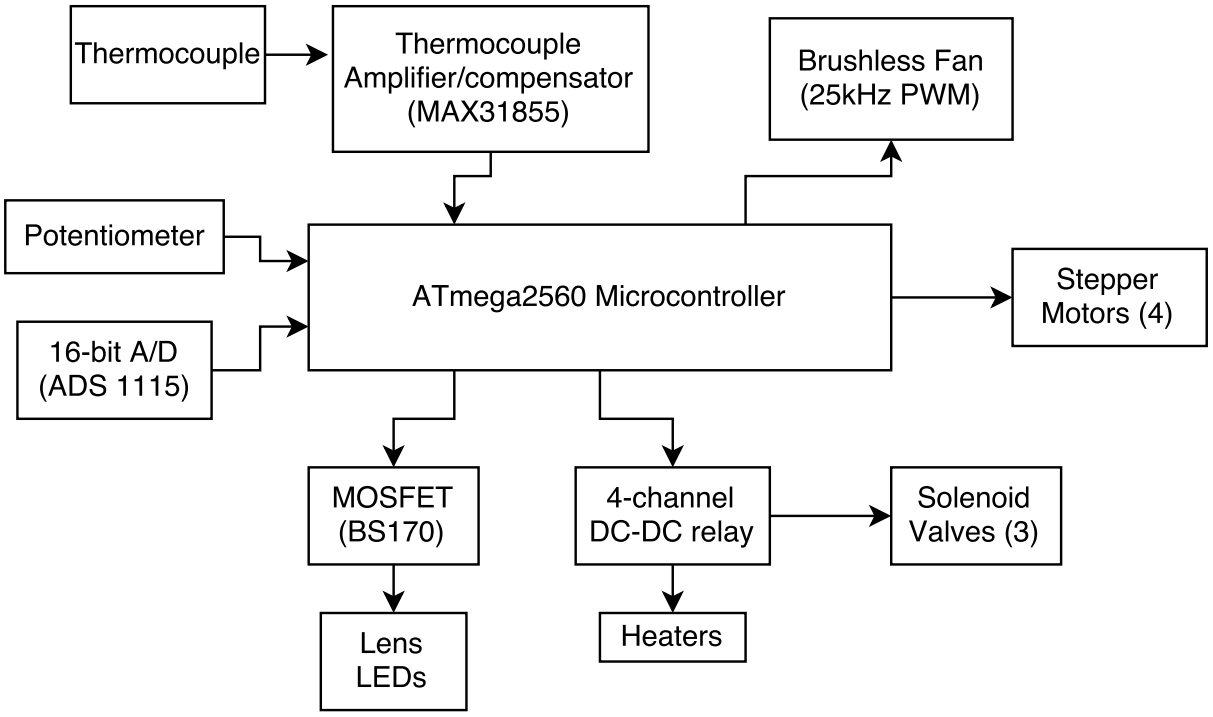

Figure 9.4: Block diagram showing the connectivity (not wires) of the main electrical components in the apparatus.

9.3 Software and Graphical User Interface (GUI)

The most number of development hours were spent coding the software for the apparatus. There were two main codes, one for the MCU and one for the Windows PC that included a graphical user interface (GUI) which allows the user to interact with the apparatus.

The GUI allows the user to:

- View live images from the camera.
- Set the camera exposure and frame rate either by button clicks or text field entry.
- Collect of *n* images triggered by clicking the *Capture* button and scroll through them using the keyboard arrow keys or mouse.
- Save .jpeg versions of these images using the *save* keyword on the command line followed by the name of the folder. The file names are integers starting at 1 for the first frame. The *save* command also writes a text file detailing the conditions (pressures, fan speed, time, etc.) at save time.
- View the temperature, pressures, needle position (calibrated by the potentiometer reading), and fan speed.
- Toggle the lens LED illumination ring using the *led* command.
- Set the control points for temperature using either buttons (increment by 5°C) or text field entry.
- Set control points for each pressure.
- Toggle the 3 solenoid valves (ON/OFF).
- Move the stepper motors by 1000 steps in either direction using the adjacent +/- buttons to turn the pressure regulator screw.
- Freeze the pressure regulator screw at the current position (regardless of control) using the *Freeze* button.
- Jog the needle position up and down using the +/- buttons using the stepper motor connected to the needle screw via gears. Note: this function was omitted in some versions.

A screenshot of the GUI is show in in figure (9.5). This was programmed in C# on the .NET Framework 4.5 using Windows Presentation Foundation (WPF) for the layout.

The MCU code and associated libraries for the sensors were written using C++. The function of the MCU was to:

- Sample the A/D converter on-demand via I2C using a customized C++ class
- Read the thermocouple temperature from the amplifier via SPI using another customized C++ class

Figure 9.5: Graphical user interface that allows user interaction and monitoring of the hardware system.

- Control the fan speed using PWM - this required a change of the microcontroller timer-based PWM frequency to 25 kHz.
- Move the stepper motors on command using the coil firing sequence described in the motor's datasheet. This had to be performed in a non-blocking way.
- Read the MCU onboard A/D converter and digital input pins.
- Write to the MCU digital output pins.
- Send and receive data through serial at 115200 baud rate using a state-machine to avoid buffer overflow and be non-blocking.

On the PC side, several threads run in the background separate form the UI thread. These handle image acquisition, in/out communication with the MCU, and some basic controls for temperature, pressure, and needle position. The program files written were as follows:

- MainWindow.xaml - XML file that defines the layout and binds elements to source properties.
- MainWindow.cs - main class where the functions of each button and text fields are defined, and other objects are instantiated.

- AnalogValue.cs - short class that holds data from analog readings (in this case, pressure sensors). It keeps a moving average which is accessible by a property that binds directly to a UI element.
- ArduinoStuff.cs - class that handles communication to the MCU. The incoming and outgoing data transfer occurs on a separate thread via a Task. Outgoing data is processed in a separate thread using timers. Incoming bytes are added to a queue until a complete command is received, which then is processed synchronously on the same thread. Control of temperature and pressure is performed entirely by this class as there is no control algorithm on the MCU - the MCU simply receives actuation commands.
- CameraStuff.cs - class that handles the communication to the USB3 camera. It requests frames, places time stamps on the image using Emgu CV, and changes camera parameters. Frame acquisition is done on a separate thread to avoid blocking the UI. The images are posted on the UI using a dispatcher.

9.4 Nozzle Design 1

The first nozzle design was 3D printed in the lab using a FlashForge 3D printer with PLA filament. This prototype was never tested, as I started improving on the design immediately after its completion. Figure (9.6) shows a cross section view of the CAD model.

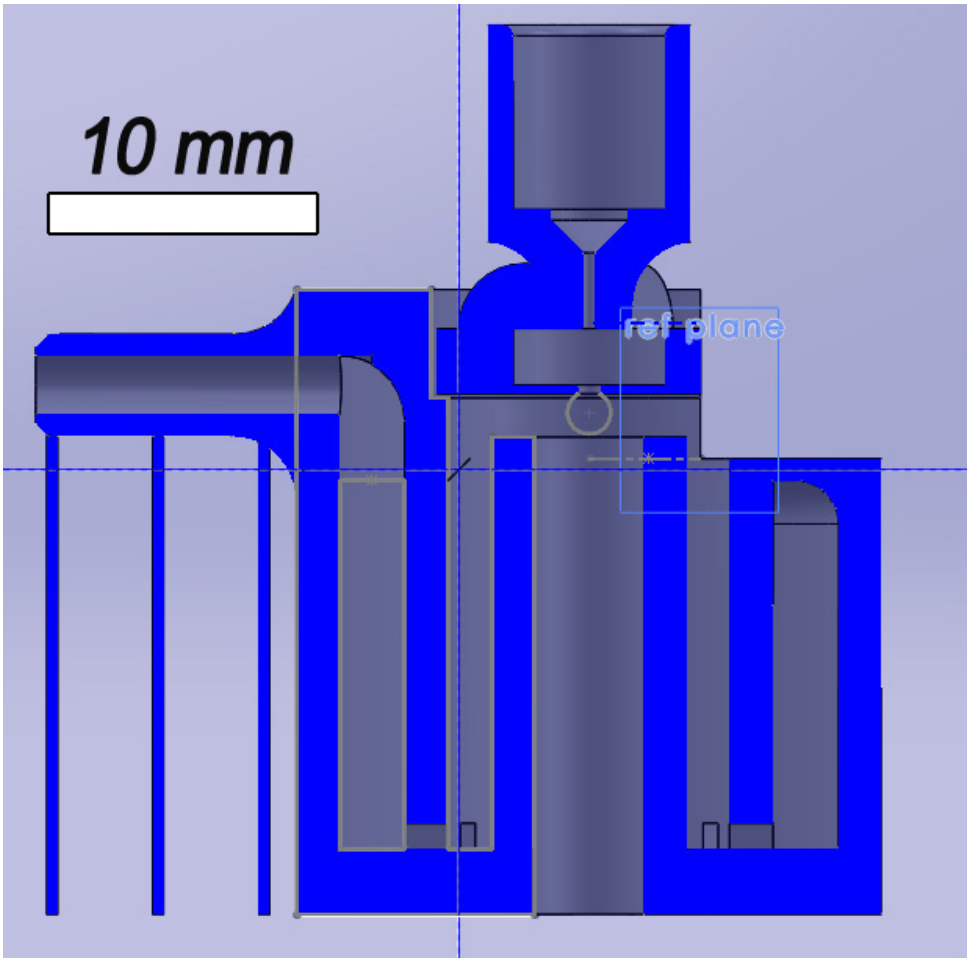

Figure 9.6: Cross section view of Nozzle Design 1 CAD model.

9.5 Nozzle Design 2

The second design improved on the first, featuring a transparent tube that allows full view of the orifice, and more compact air channels. This was tested using soap water. While bubbles were made at a rapid rate, they coalesced into larger bubbles and adhered to the walls of the acrylic exit tube.

This experiment provided very useful information for future designs - the need for a large flow, low pressure outer air duct (instead of using the main compressed air line) and the elimination of the external guide tube. Figure (9.7) shows the CAD cross section, and figure (9.8) shows a photograph. This prototype was printed as was Nozzle Design 1.

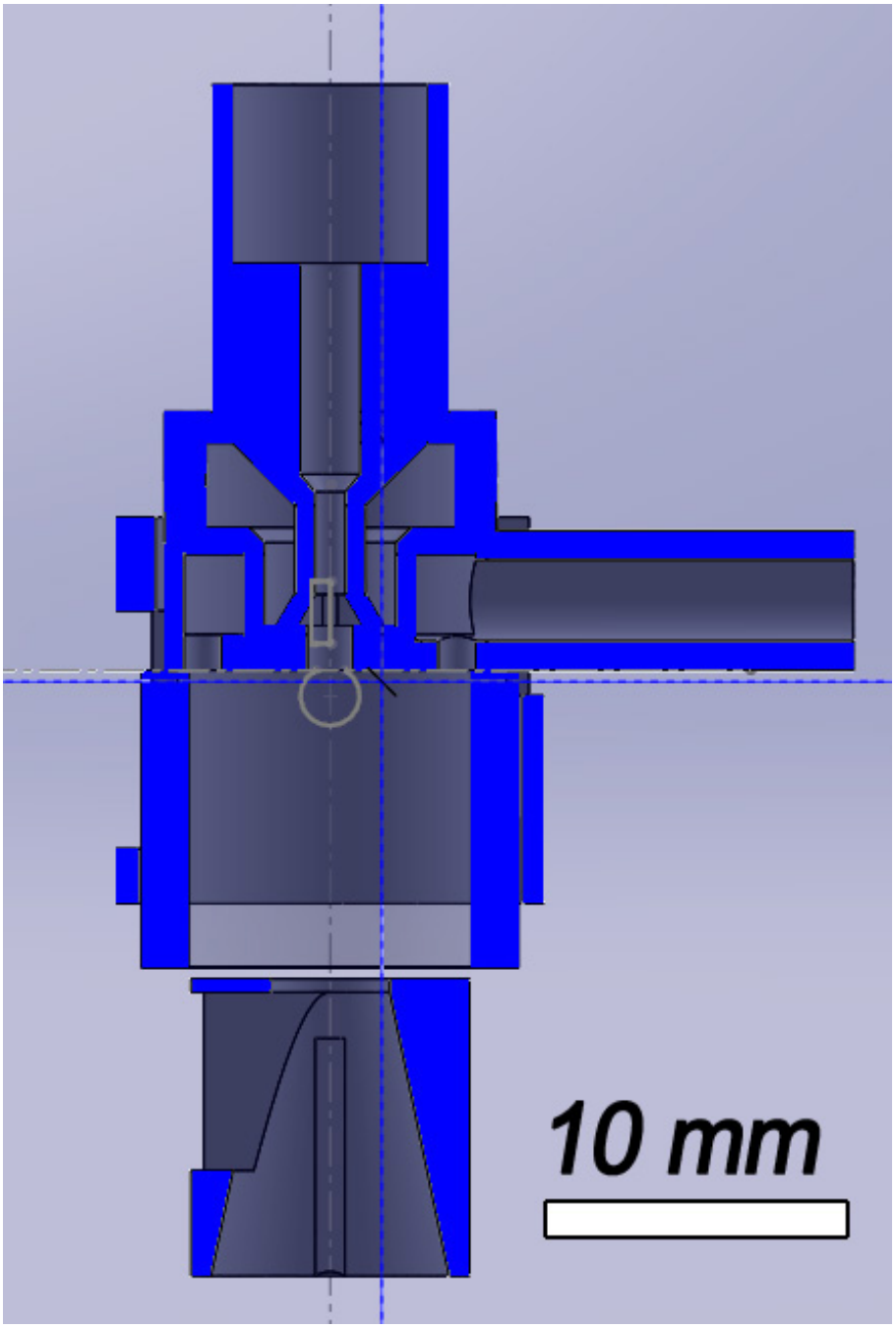

Figure 9.7: Cross section view of Nozzle 2 CAD model.

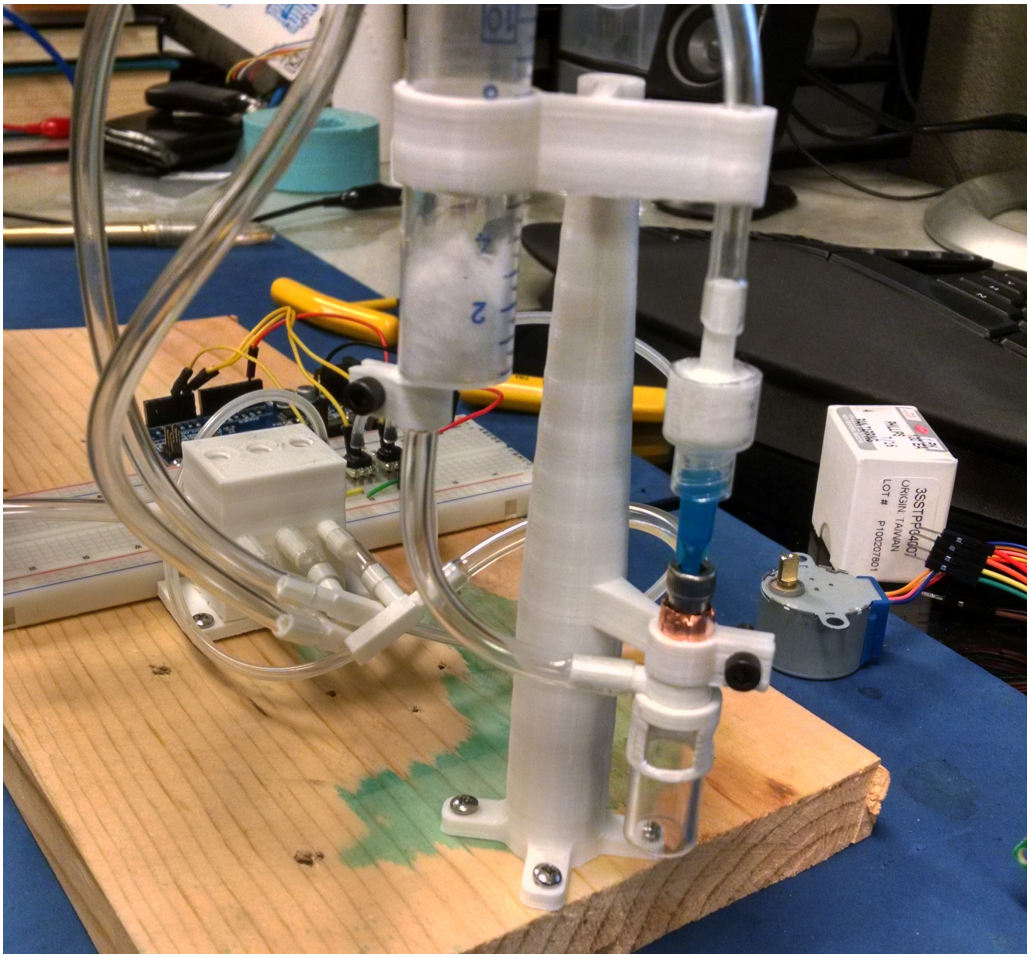

Figure 9.8: Photograph of Nozzle 2 mounted.

9.6 Nozzle Design 3

The final design for soap-bubble tests, Nozzle Design 3 featured a powerful HP 2.7A 12VDC fan for the outer duct and a pulley-driven 1/4-20 screw for needle height adjustment with stepper motor control. Again, a CAD cross section and photograph is shown in the figures. The assembly was printed as before, the inner nozzle diameter was 1.0 mm, and used a 25 gauge needle.

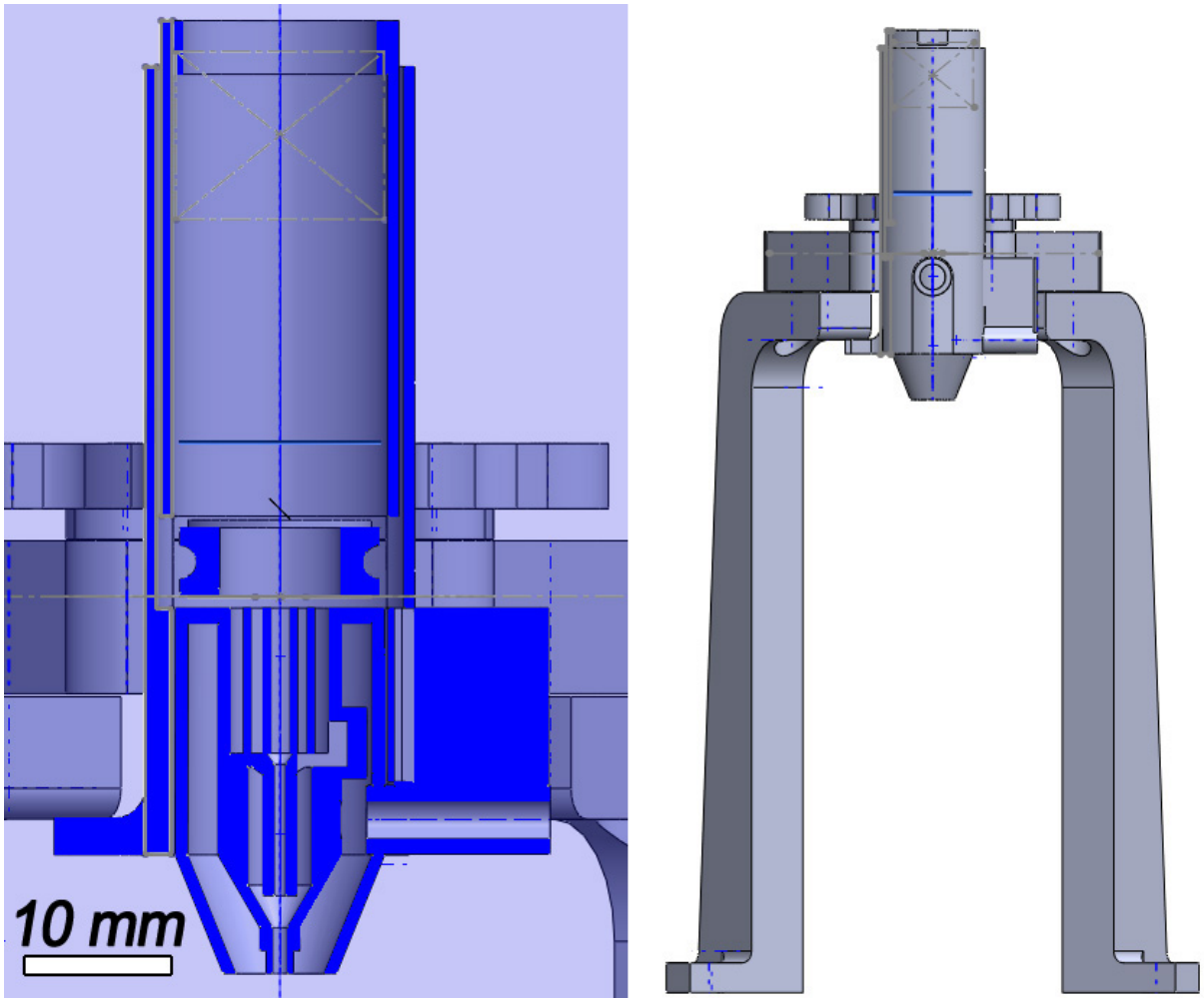

Figure 9.9: Cross section view of Nozzle 3 CAD model.

This prototype produced various bubbles using soap water, pure water, and liquid latex. The following figures show images from the apparatus' camera system. First, soap water was used to jet bubbles of various sizes at rates up to 40 Hz. All adjustable parameters - needle pressure, liquid pressure, fan speed, and needle position had a pronounced effect on bubble formation, size, rate, and integrity. The conditions had to be carefully adjusted for a bubble to form - otherwise, the liquid would either drip, spray, jet, or produce bubbles that popped before or after break-off (detach from the nozzle). Typically, the needle pressure had to be between 0.3 and 1.0 psi, while the needle pressure between 0.2 and 0.5 psi. The fan helped with bubble break-off, however it was best to keep it at the lowest setting (50% PWM duty cycle) - the powerful fan often broke the bubbles at high settings. The needle height produced the best

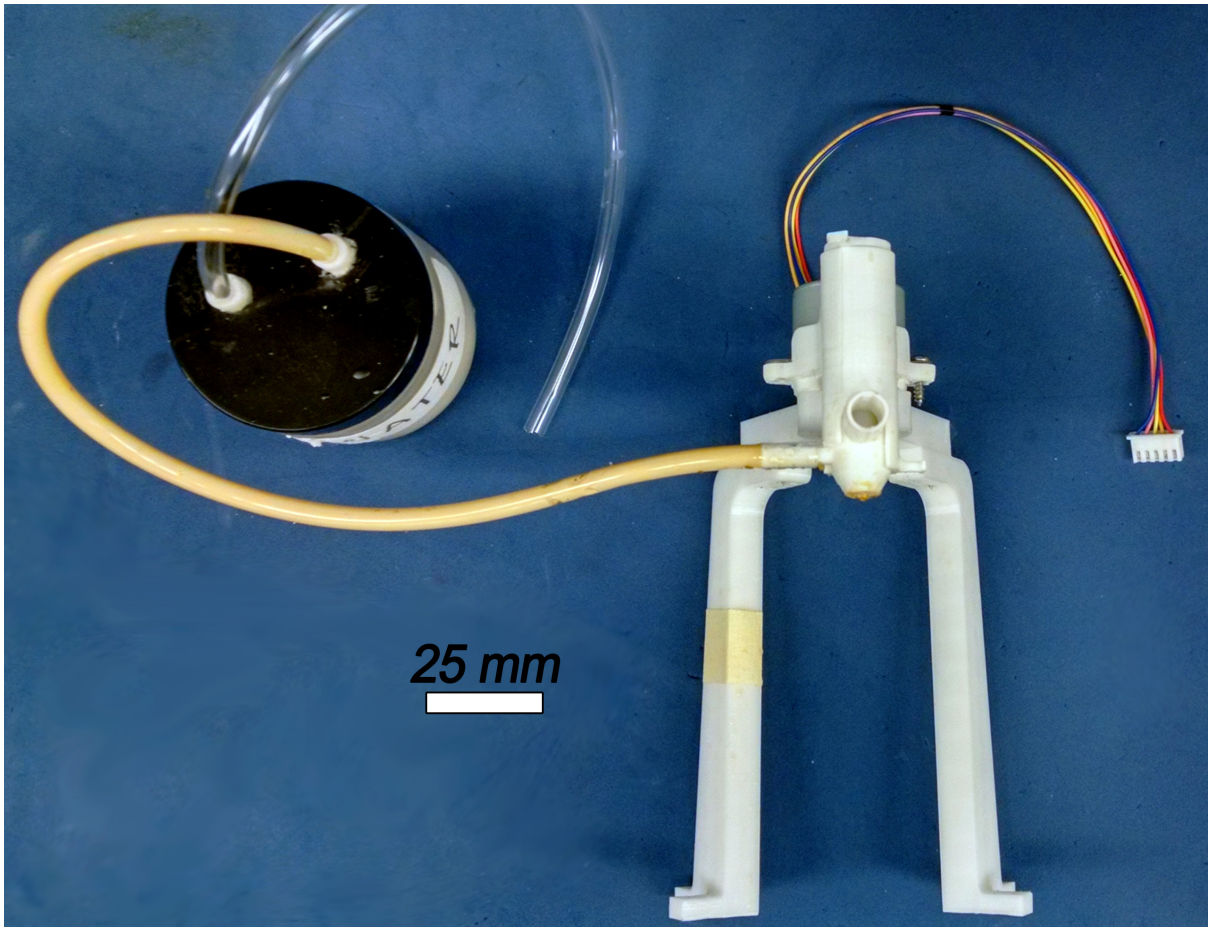

Figure 9.10: Photograph of Nozzle 3 and its stand.

bubbles when it was slightly above the nozzle exit. Lowering it was worse than raising it.

Pure water was also run through the system to see how the higher surface tension and evaporation rate (due to lack of soap molecules on the surface) would affect bubble formation. The system was capable of producing water bubbles, however they were more prone to popping and the bubble wall was much thicker - more like an air bubble trapped inside a water droplet. Interestingly, it was possible to place up to 5 individual air bubbles inside a droplet, as shown in the figures.

Next was liquid latex: rubber in a water-based solvent. The idea was to deposit bubbles and dry them into a rubber foam. Again, the controlled parameters had a significant impact on bubble formation, much like that of soap bubbles. Due to the long chain polymer molecules in the solution, bigger bubbles could be formed, as shown in the figures.

The bubbles in figure (9.23) were ejected from the nozzle and continued falling through the air while retaining shape. These were collected on a piece of paper by manually moving the sheet and stacking rows of bubble, effectively creating a ”foam” in its liquid state. Figure (9.24) shows a photograph of this. However, the bubbles popped during the drying/curing process, and the foam structure collapsed into a liquid puddle with very few bubbles hardening into solid rubber.

The ideal scenario, as described in the project proposal, was to create bubbles out of a molten polymer - therefore, the liquid would solidify by freezing rather than curing.

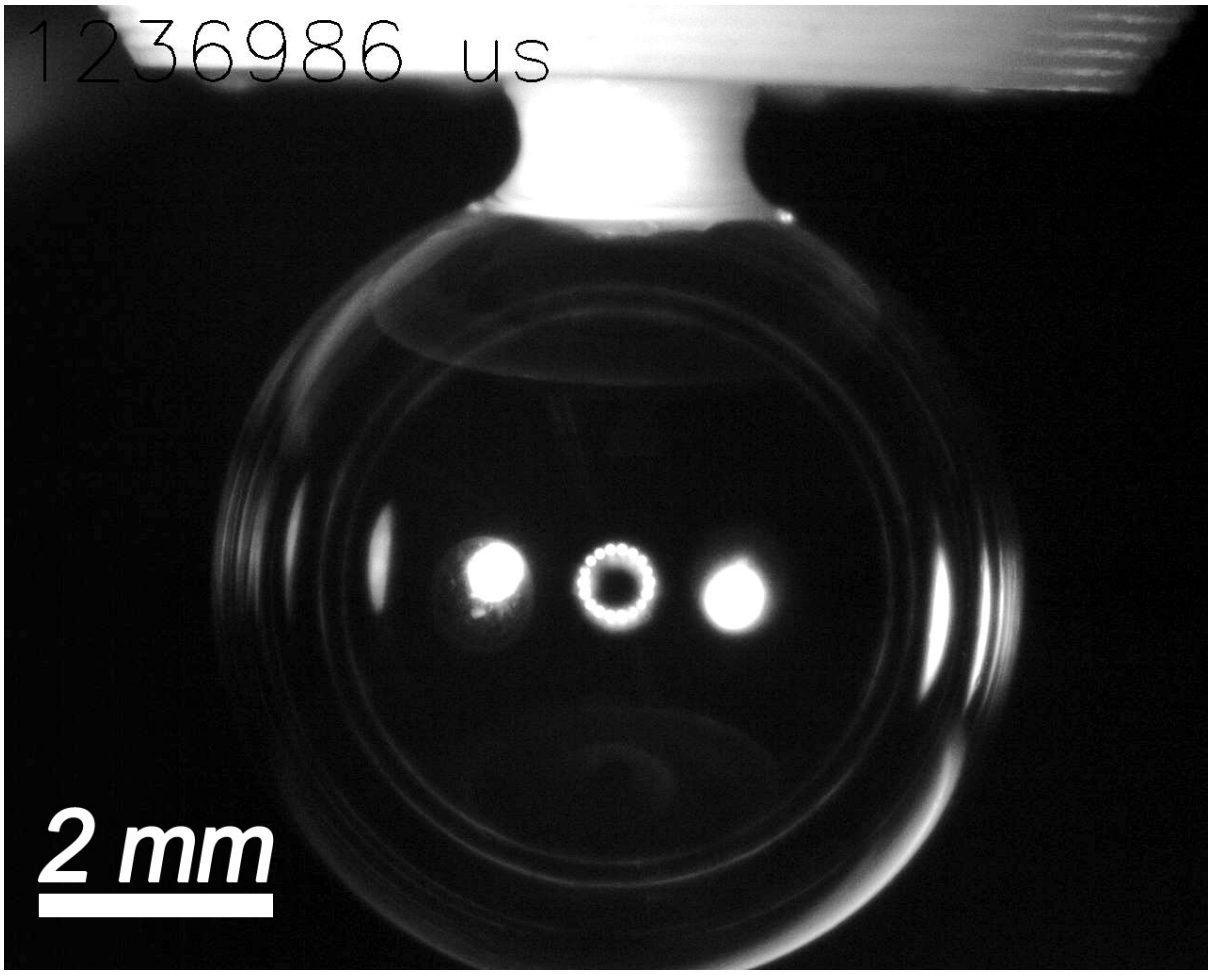

Figure 9.11: Large soap bubble produced when the liquid/needle pressures were very low and fan off.

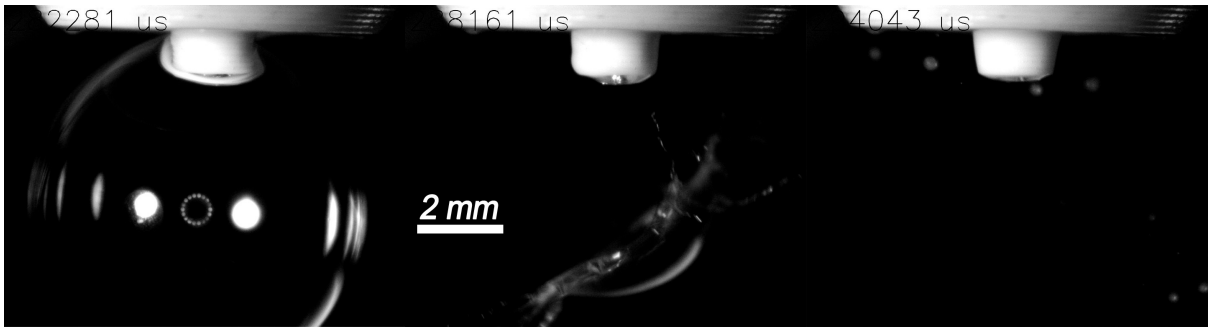

Figure 9.12: Large soap bubble bursting before detaching from the nozzle.

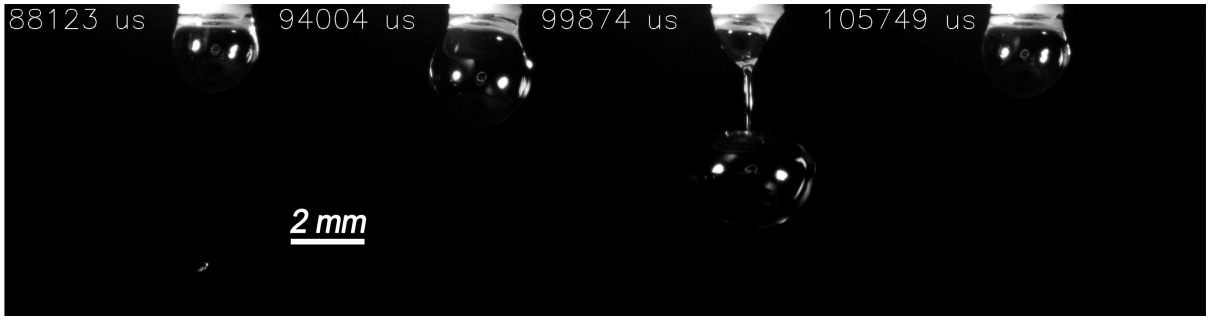

Figure 9.13: Smaller soap bubbles (around 3mm diameter) produced by larger needle pressure and turning the fan on.

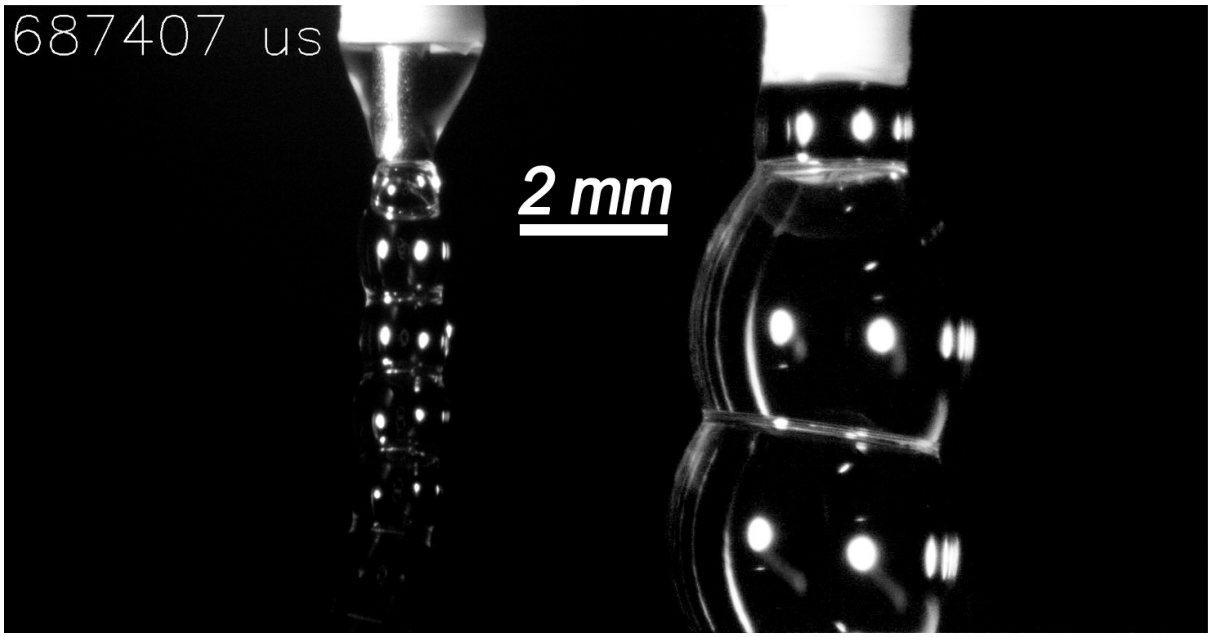

Figure 9.14: Stringed soap bubbles, falling out of the nozzle like a chain. The conditions to achieve this are very particular - a small disturbance will break the chain and resume to individual bubbles. This phenomenon can last up to 1 minute.

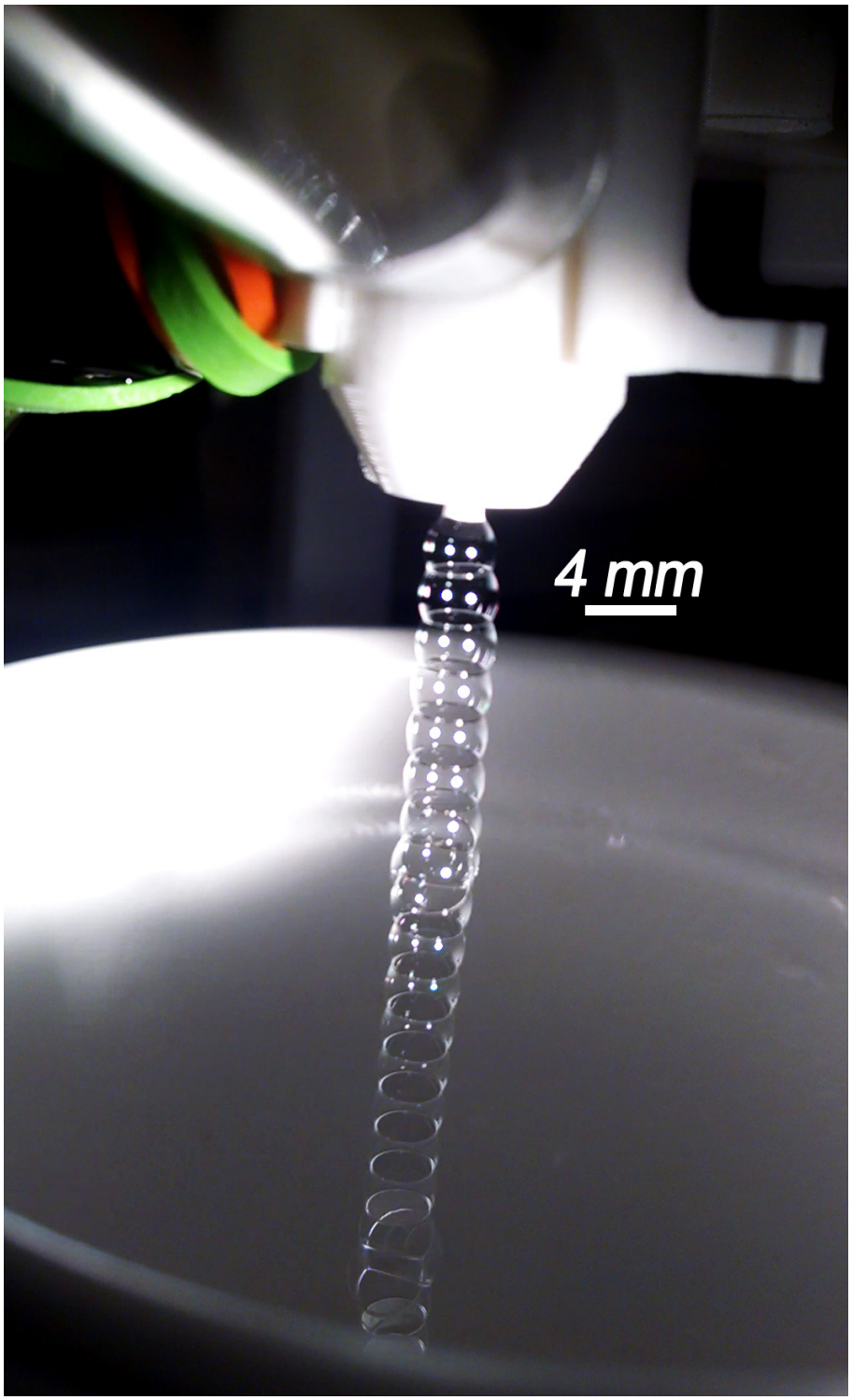

Figure 9.15: Photograph of the nozzle while the stringed bubbles phenomenon is occurring. The rate of bubble formation here was approximately 5 Hz.

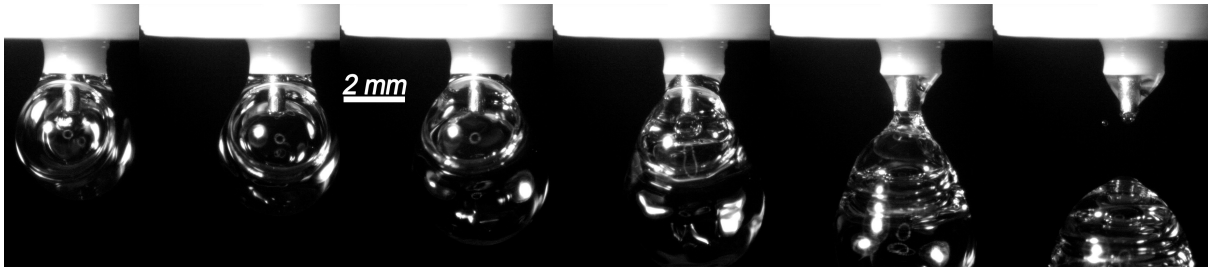

Figure 9.16: An air bubble inside a water droplet. Notice the difference in shape and wall thickness as compared to the soap bubble. The droplet has more mass when it detaches from the nozzle due to the higher surface tension of pure water.

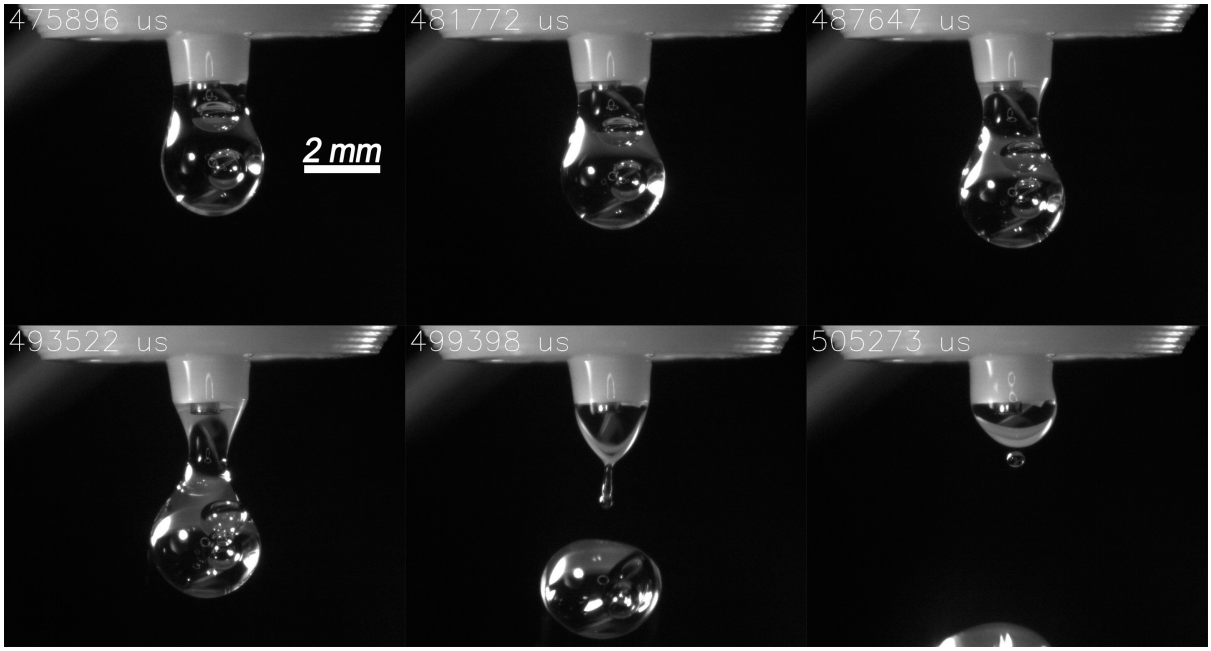

Figure 9.17: Two air bubbles inside of a water droplet. Notice the small size of the air bubbles and how they separate even after droplet detachment from the nozzle.

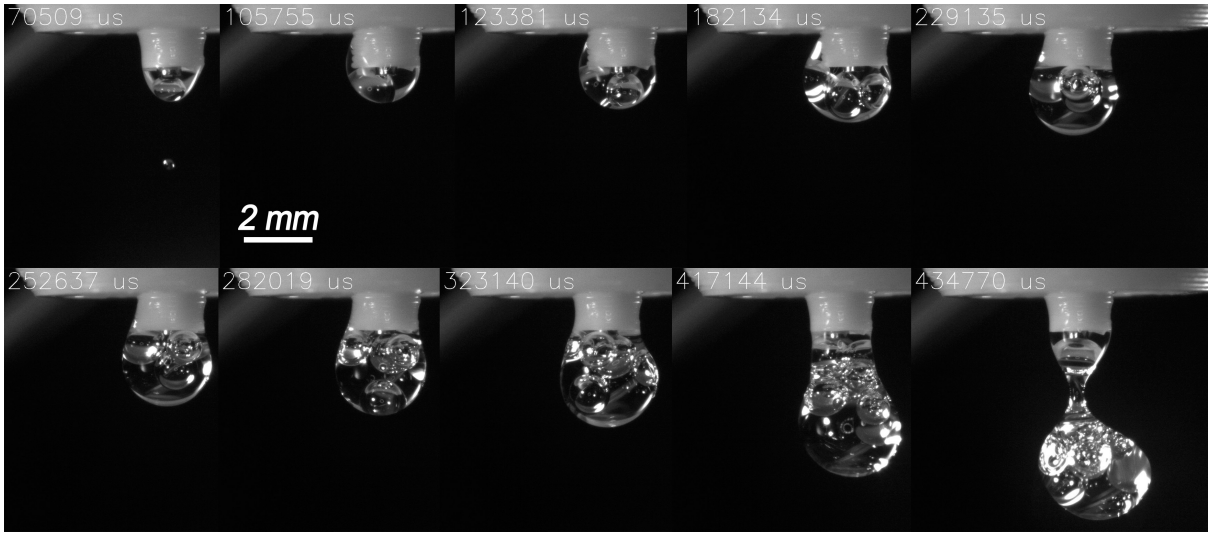

Figure 9.18: Five distinct air bubbles inside a water droplet. The bubbles are injected one at the time as the water droplet grows and eventually breaks off. The bubbles circulate inside the droplet but do not coalesce.

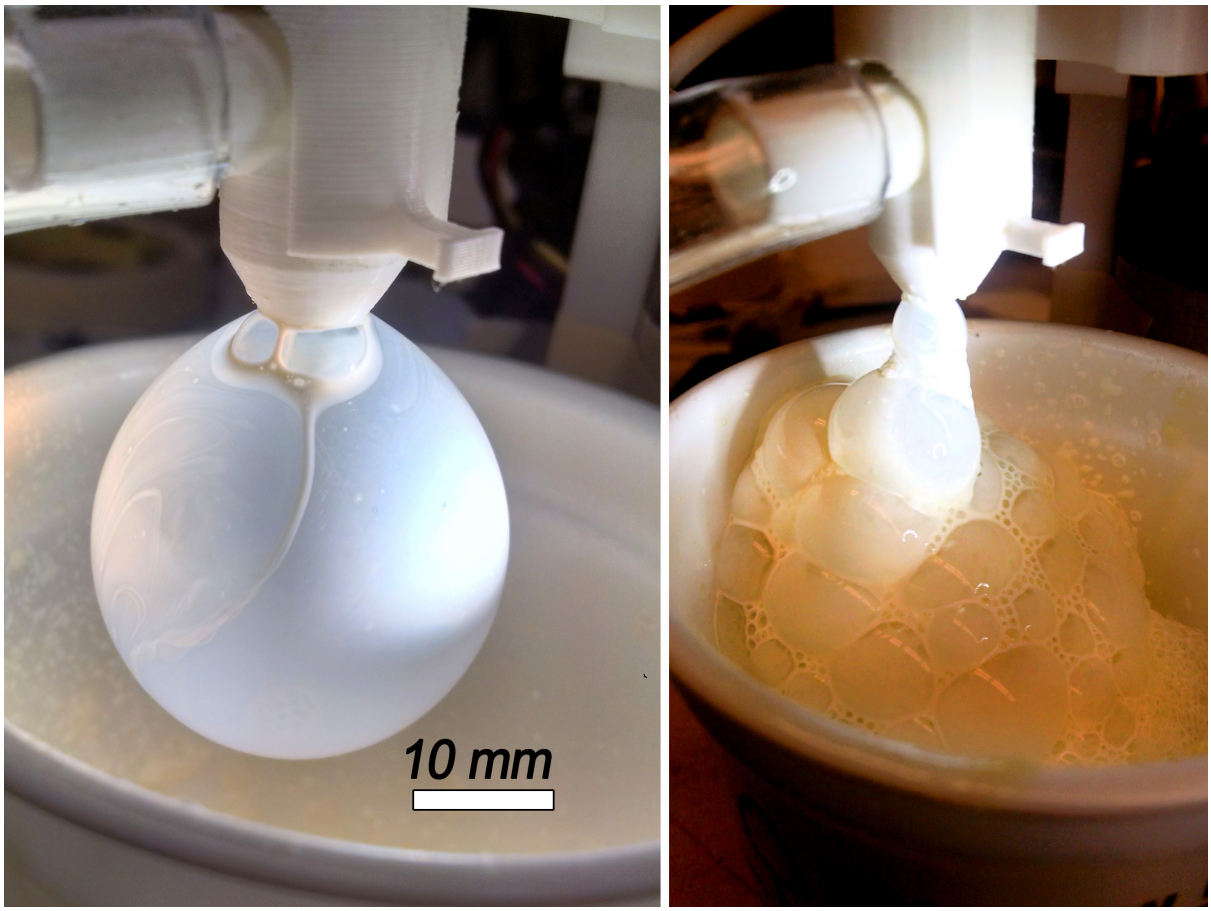

Figure 9.19: Left: Photograph of a large latex bubble formed by the nozzle at low input pressures and fan off. Right: stringed bubbles produced at slightly larger pressures and fan at low setting. These bubbles deposit into a cup to form a foam.

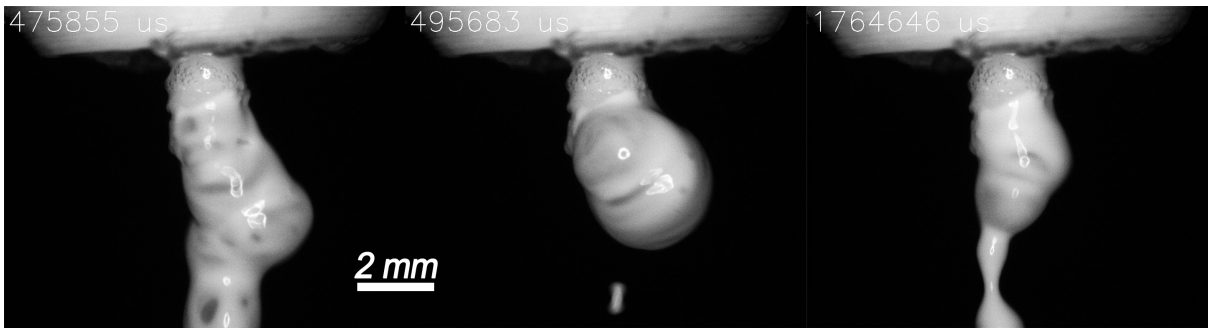

Figure 9.20: Stringed latex bubbles being ejected from the nozzle. This was an unstable/oscillatory regime.

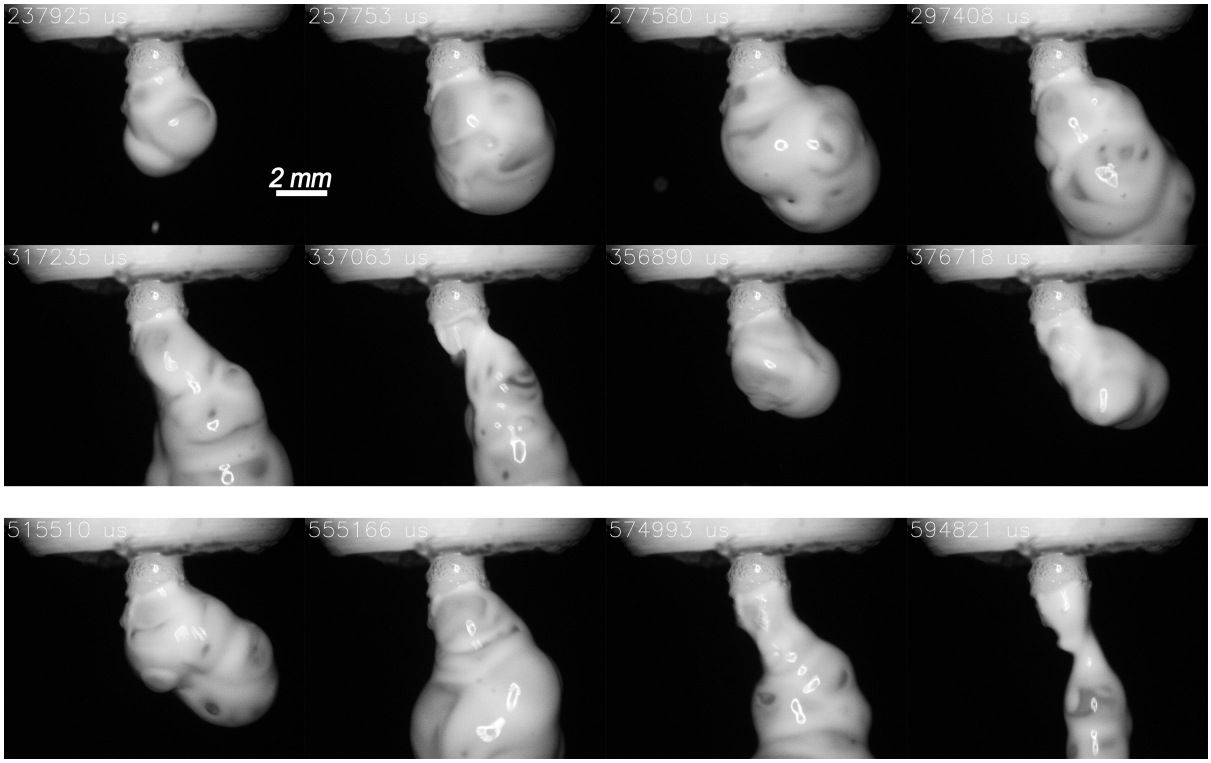

Figure 9.21: Clumps of latex bubbles produced. The bubbles grow and remain attached to the group until they are blown away by the fan.

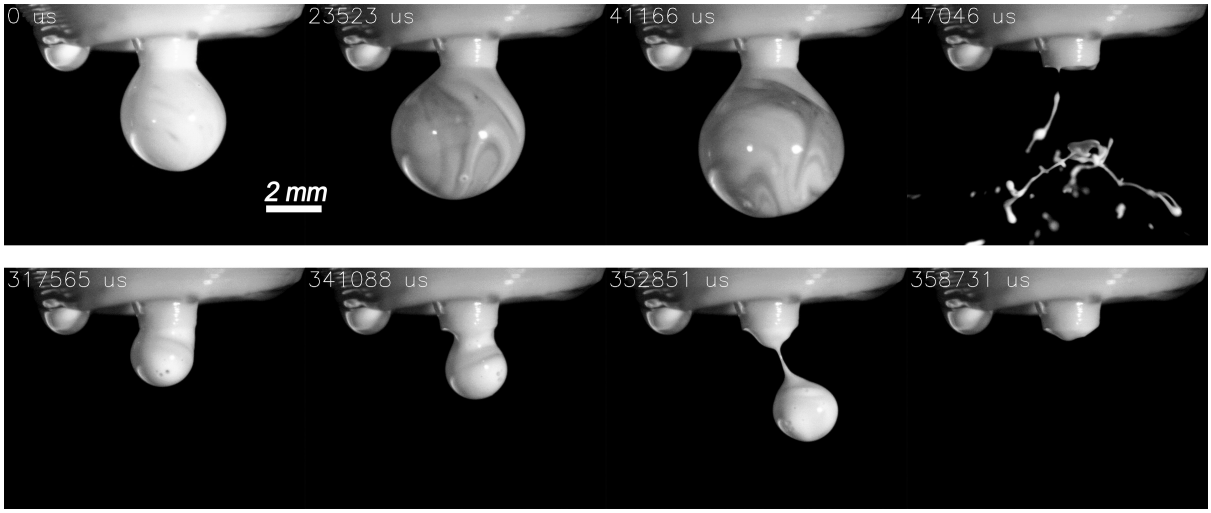

Figure 9.22: Top row: single bubble formation followed followed by burst prior to detachment. Bottom row: small single bubble forming and detaching in an unstable regime.

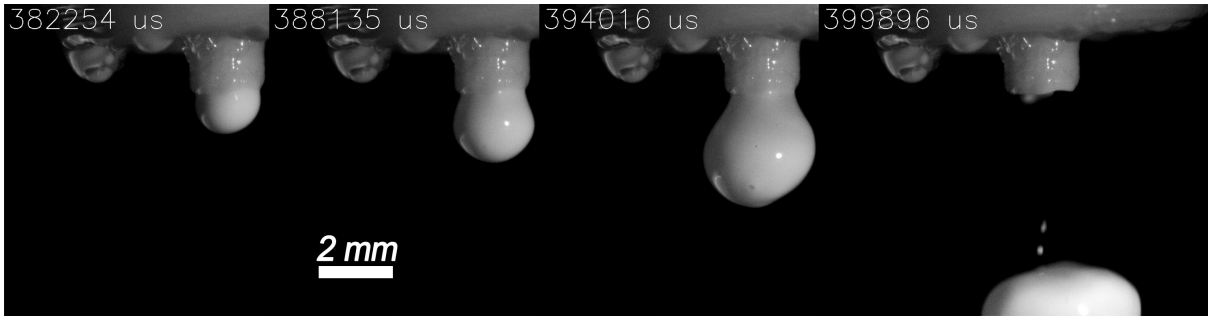

Figure 9.23: Latex bubble forming and detaching without bursting.

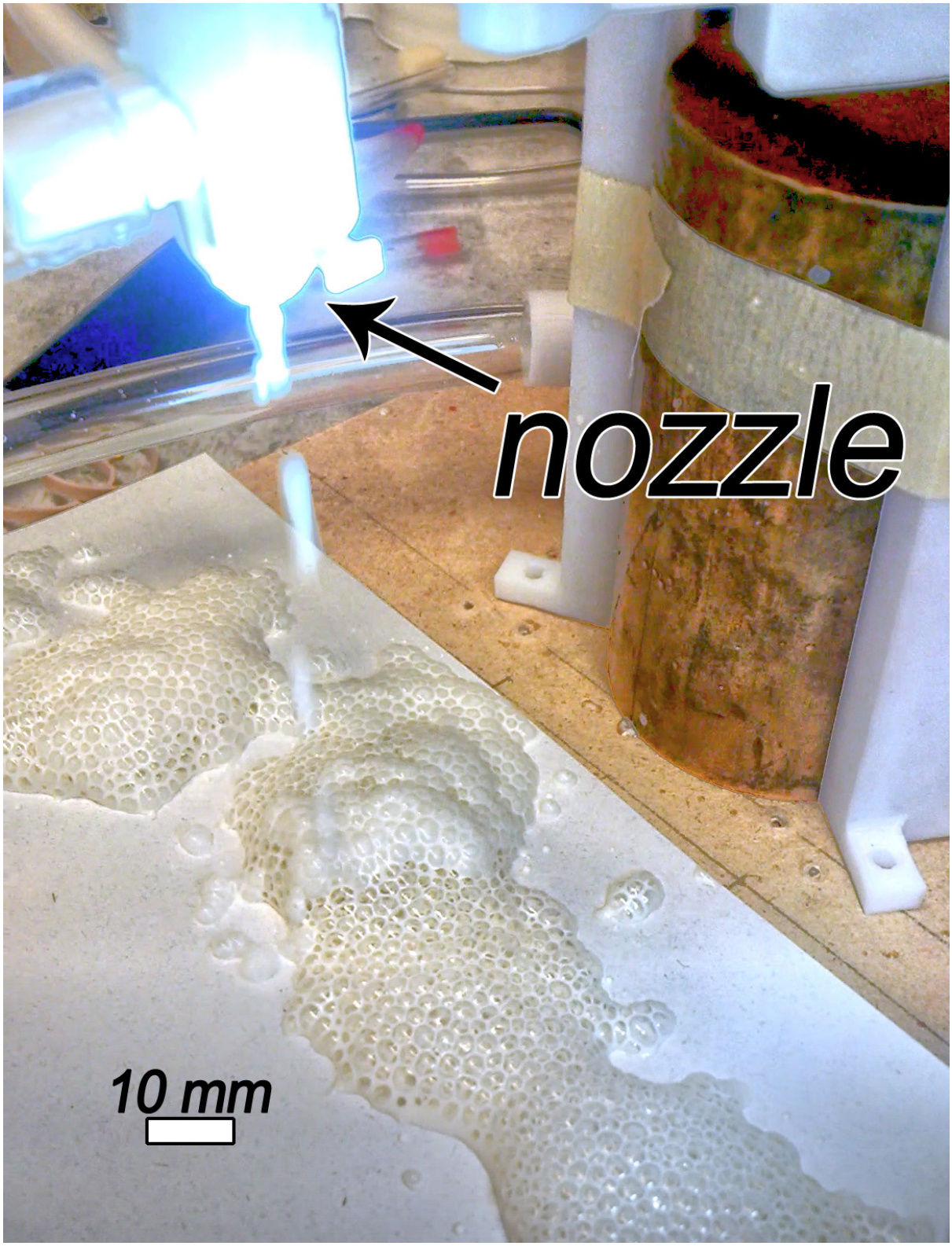

Figure 9.24: Still shot from a video demonstrating the ejection of latex bubbles and their collection on a moving sheet of paper.

9.7 Nozzle Design 4

To begin experiments with molten polymer, a heated nozzle was designed. This was more difficult as 3D printing was not an option, at least not using the process from before. Design 4 is made of high temperature materials - brass, aluminum, PTFE, silicone.

The nozzle started out as a 0.5 mm Prusa 3D printer extruder nozzle which was drilled by hand to 1.0 mm. The aluminum heated block was machined on an end mill to house two cartridge heaters (6 Ω resistors) and a channel that guided polymer melt from the reservoir to the nozzle. This design also featured a motor/gear system that would raise/lower the needle position from the GUI, and a 5 $k\Omega$ 10-turn potentiometer to keep track of the absolute position of the gears' rotation angle.

A hopper was able to store pellets or small pieces of the polymer material to be printed. A clear window allowed view into this chamber and it was pressurized using the air line from the regulator. As the pellets fell through and came in contact with the heated aluminum, they would melt and continue towards the nozzle by the aid of the pressurized air on top.

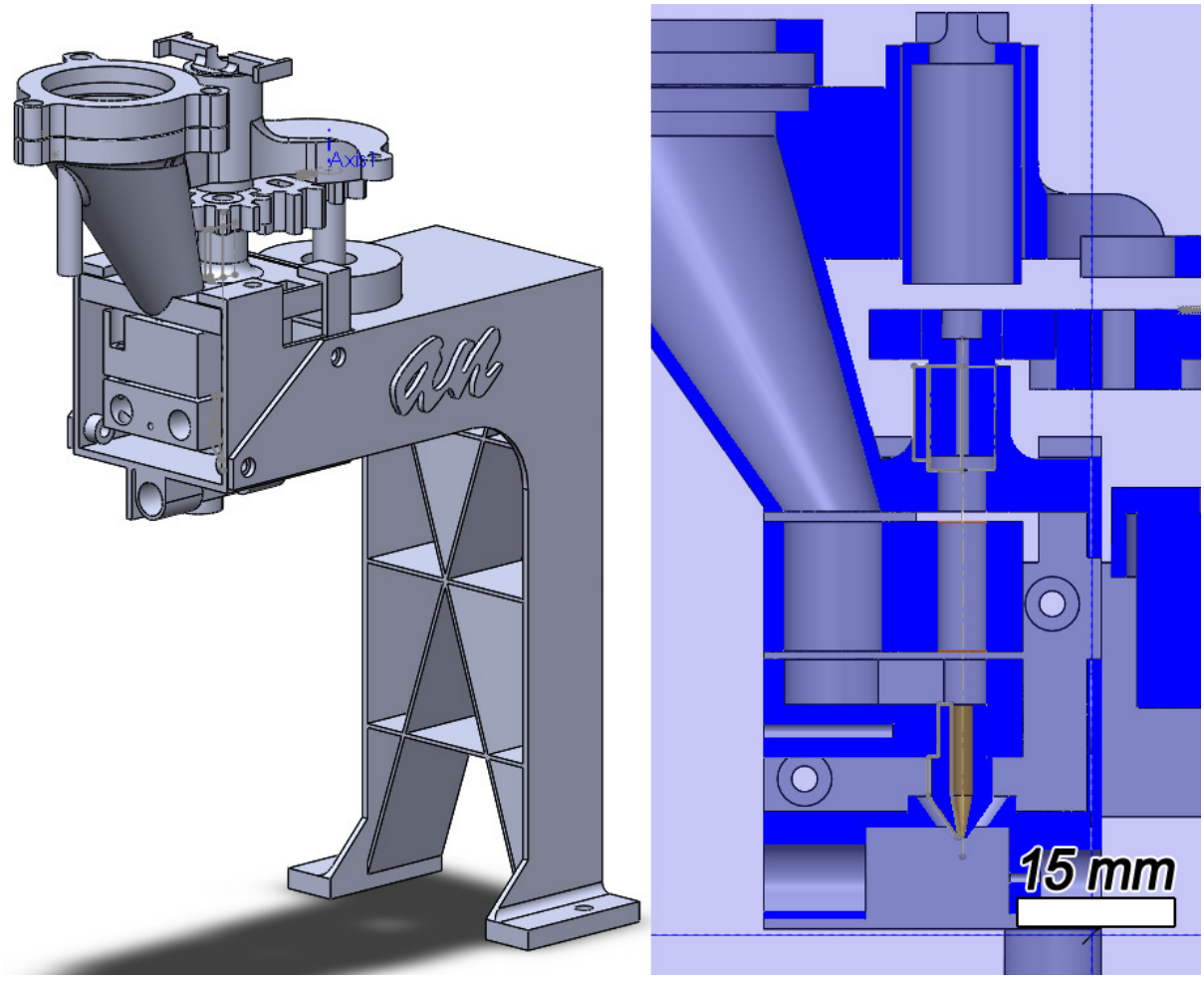

Figure 9.25: Cross section view of Nozzle 4 CAD model.

PLA was tested in the system first. The experiments showed that the material's viscosity is too high and surface tension too low to form a droplet that falls under gravity. The melt is

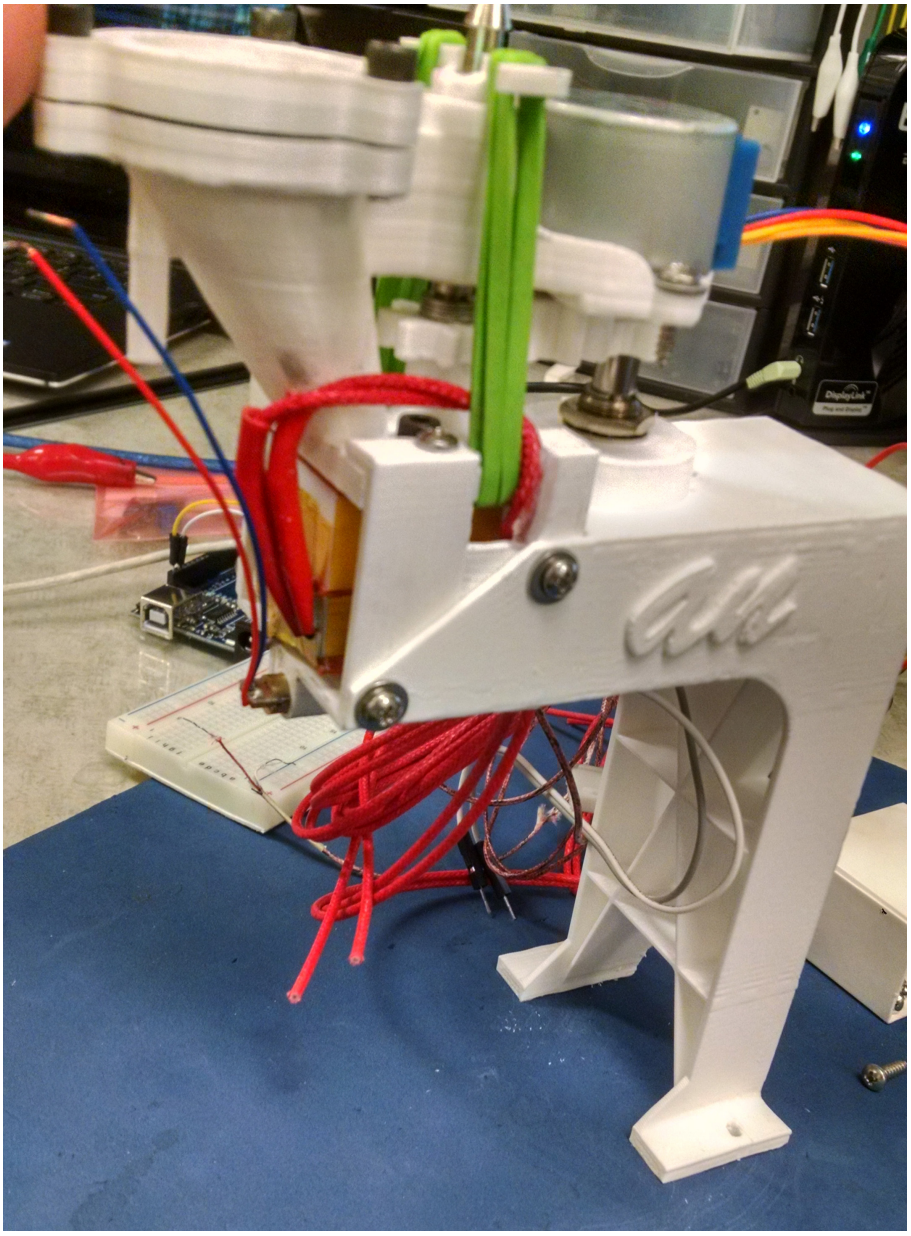

Figure 9.26: Photograph of Nozzle 4 mounted.

therefore “stringy” due to its long chain molecules. The following figures show the behavior of PLA when ejected through the nozzle.

Since viscosity of polymers is a decreasing function of temperature, the nozzle temperature was increased up to 300°C in an effort to get PLA to form droplets or even bubbles. However, the change in flow behavior of PLA was negligible with this increased temperature, and parts of the nozzle were deformed by this high temperature. These tests made it apparent that PLA would not work with the current design. Due to its high viscosity and high molecular weight (making the melt highly non-Newtonian) there was no bubble formation and no droplet break-off. Searching for materials with lower melt viscosity brought me to paraffin wax - the short hydrocarbon molecules was hypothesized to make for a more suitable material.

Indeed, the wax melt fluid at 90°C was able to produce bubbles. However, its adhesion

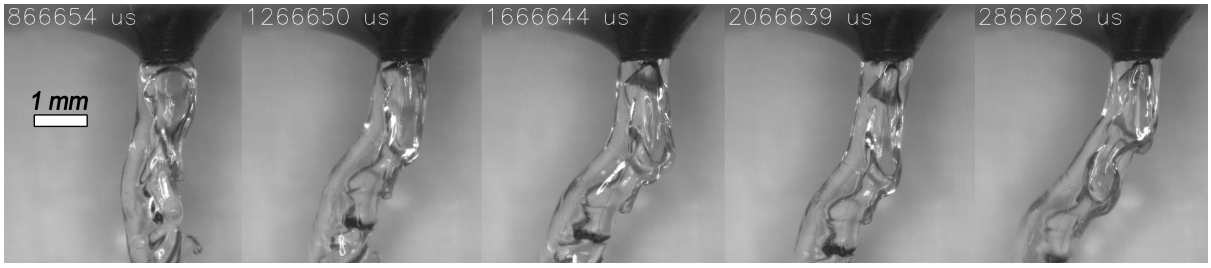

Figure 9.27: PLA coming out of the nozzle slowly in a continuous stream. The material never breaks off from the nozzle.

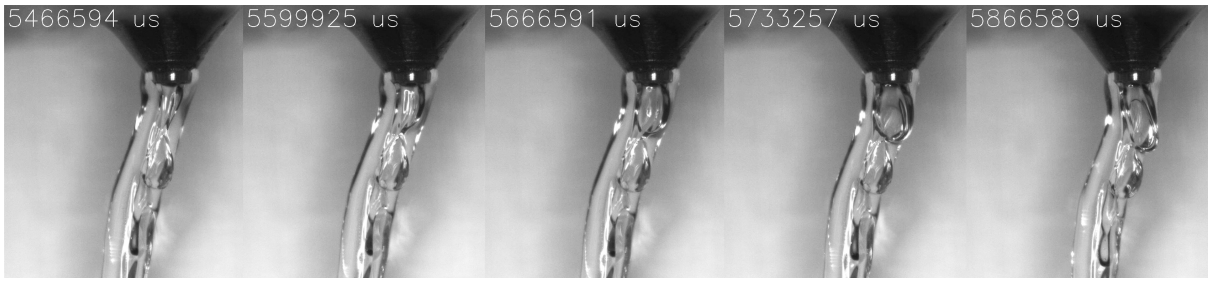

Figure 9.28: When an air bubble starts forming in the PLA stream, it stretches the thinnest part of the PLA wall until it ruptures and the air escapes.

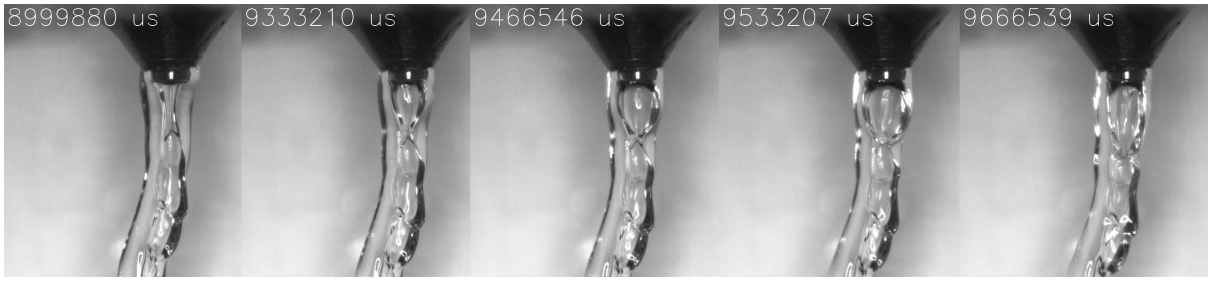

Figure 9.29: A similar scenario as in figure (9.28).

to the brass nozzle surface caused the liquid to flow upward along the nozzle surface. The following figures show how this looked like.

When the needle air supply was cut using by the solenoid valve, the wax flowed out of the nozzle in various patterns. First, there was a stream regime where a 1 mm diameter continuous stream of liquid wax flowed straight down. As the liquid pressure dropped, the stream started to break up and form periodic droplets, as seen in figure (9.33).

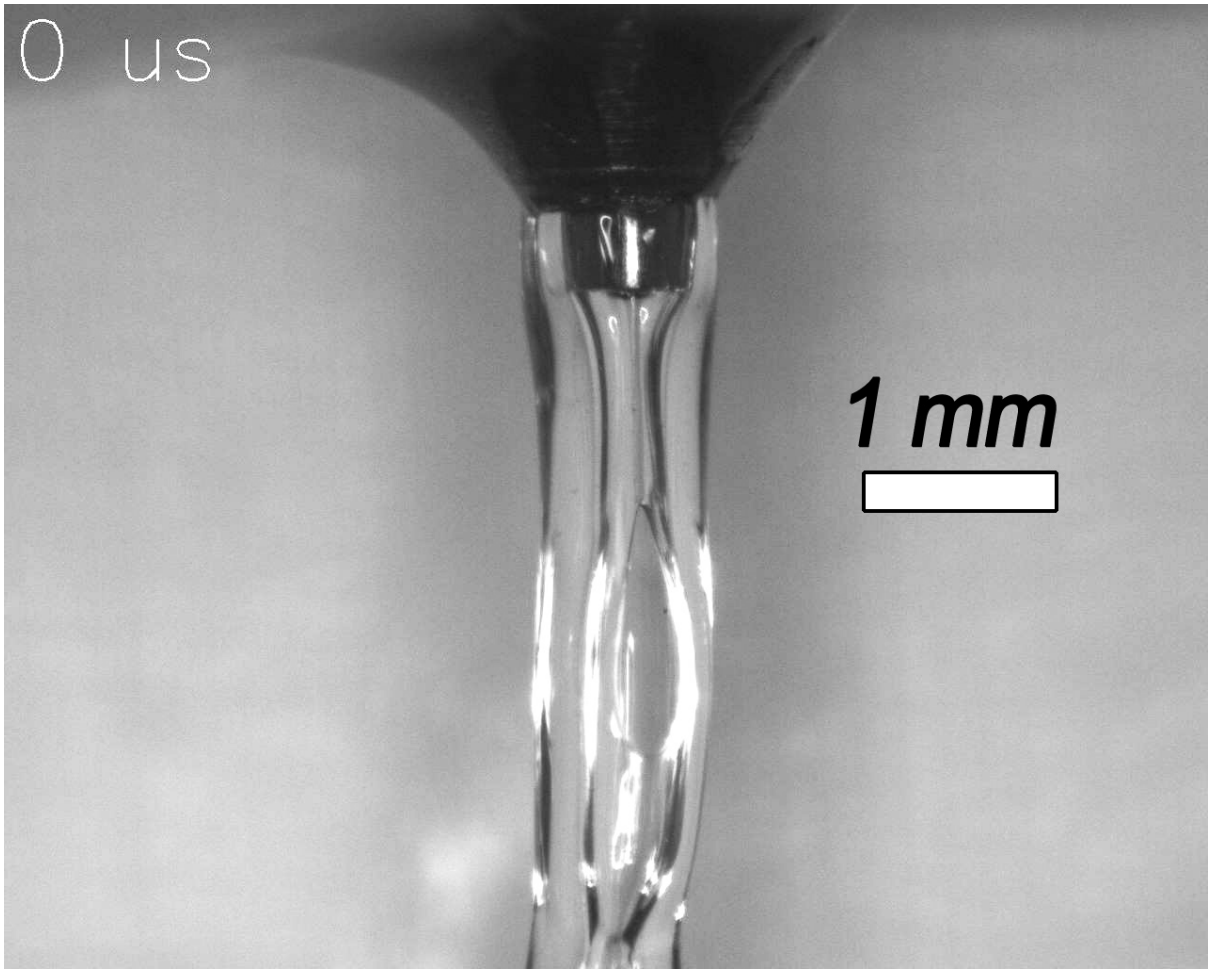

Figure 9.30: PLA stream coming out of the hot nozzle at 230°C. Notice the inner air cavity created by the air ejected from the needle, and how the outer wall is split to create the air escape opening on the front.

Figure 9.31: Wax flowing upwards on the brass nozzle and forming bubbles.

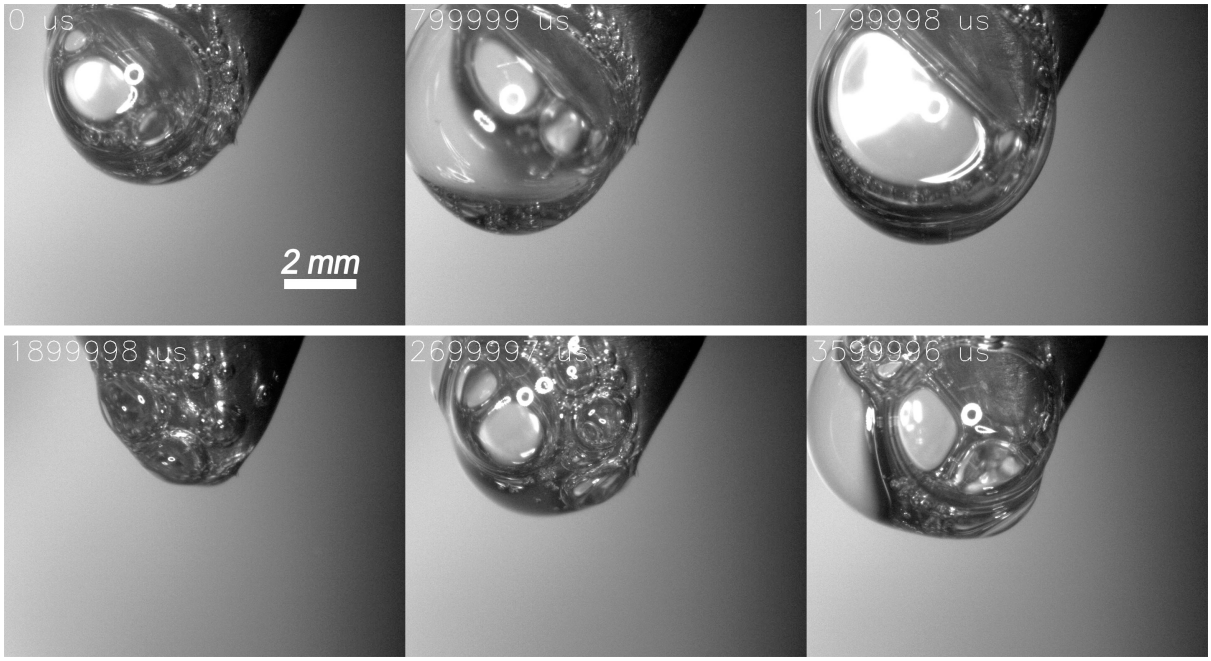

Figure 9.32: Another view similar to figure (9.31).

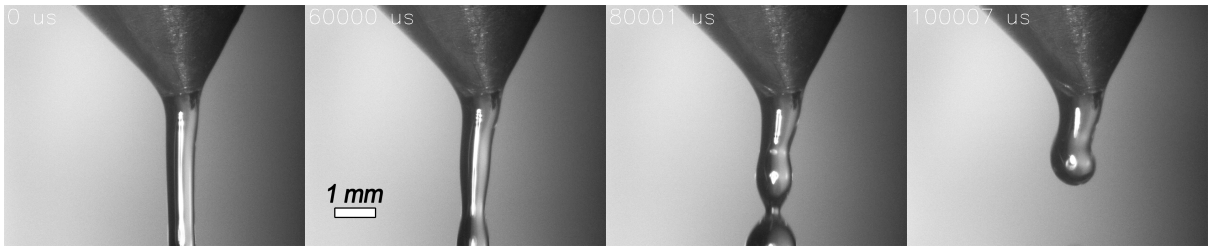

Figure 9.33: Continuous laminar stream of wax flowing straight down from the nozzle exit.

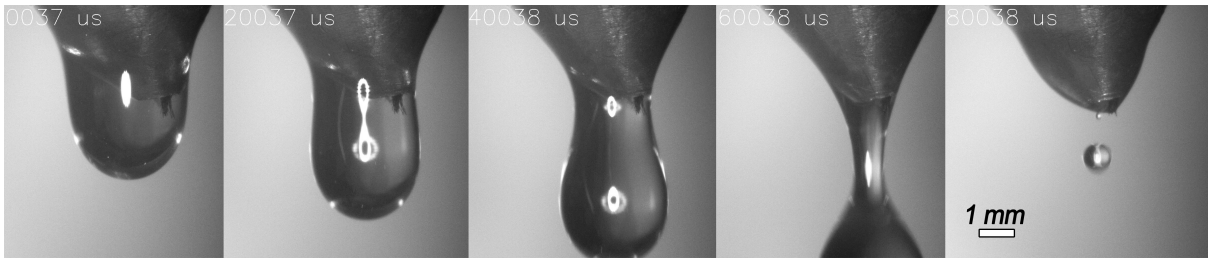

Figure 9.34: Wax dripping regime. Notice how the wax builds up on the brass nozzle above the nozzle exit - the drops are much larger than the 1.0 mm hole.

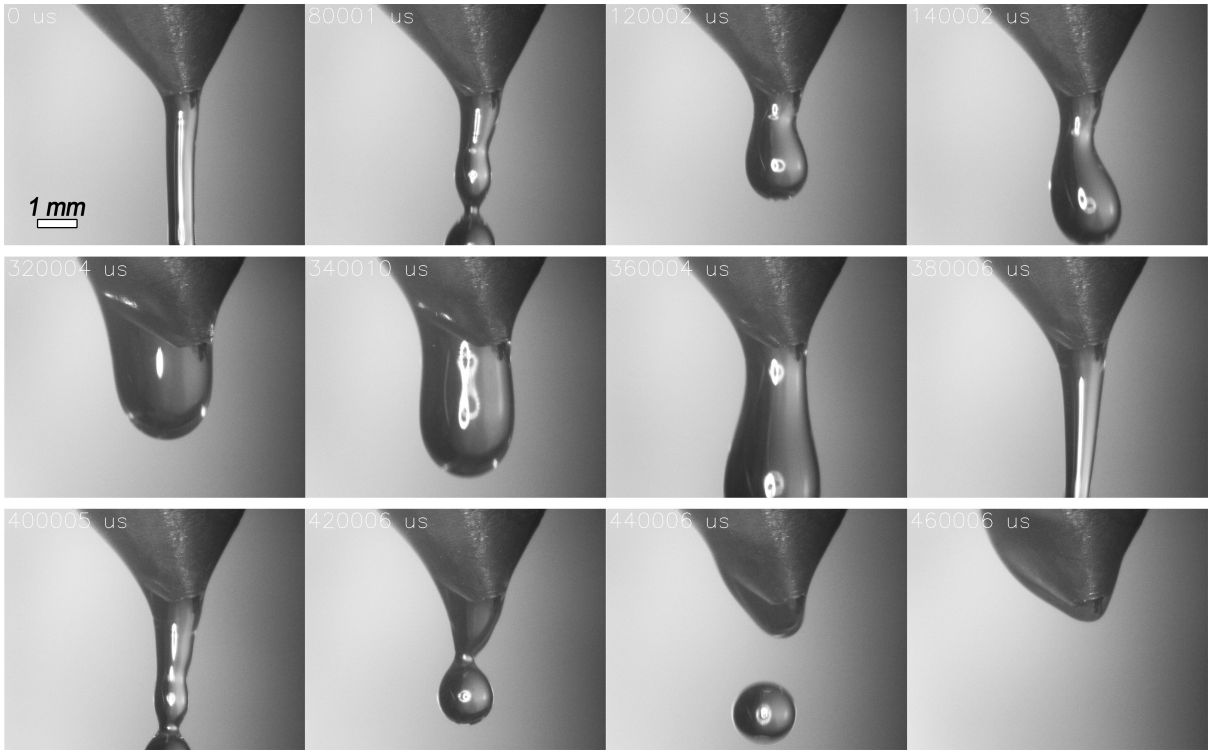

Figure 9.35: Another view of wax flowing out of Nozzle Design 4 at 90°C. Notice the periodic change in wax shape.

9.8 Nozzle Design 5

The next design iteration featured 3D printed components made of brass and epoxy resin. The brass models are made by first 3D printing in wax, and then casting it in plaster. These models come with a smooth finish, however the resolution is limited and the minimum feature dimension is 1 mm (e.g. holes, wall thicknesses, etc.).

The resin models are made using stereolithography (SLA), and the material is claimed by the vendor to be heat resistant to 120°C. This process allows 0.5 mm features and much smaller holes and detail. Both models were printed by Shapeways (www.shapeways.com) and took approximately 2 weeks to deliver.

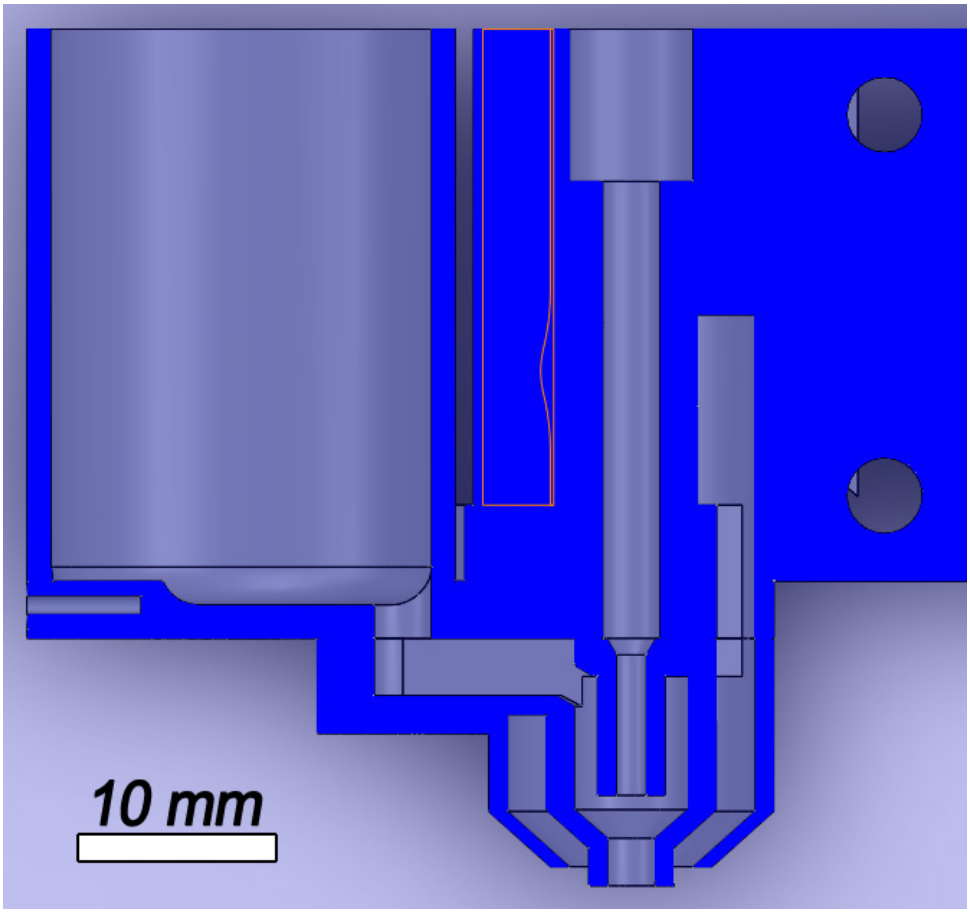

Figure 9.36: Cross section view of the 3D printed brass Nozzle 5 CAD model.

The brass nozzle has a large diameter of 5 mm and was designed for use with PLA. The main force leading to droplet detachment is gravity, which increases proportional to r^3, while the main opposing force is surface tension, which increases as r^2. The idea was therefore to increase the nozzle diameter and see if it's sufficient to cause droplet detachment. Unfortunately, even with the increased nozzle size, the PLA flow rate was far too slow even at 260°C and 8 psi of liquid pressure.

Next, the resin nozzles were tested, using paraffin wax. Several prototypes, with different needle diameters between 0.50 mm and 1.2 mm were used, as well as nozzle exit holes between 1.0 mm and 3.0 mm. The brass heated reservoir was set at 140°C to raise the wax far past its

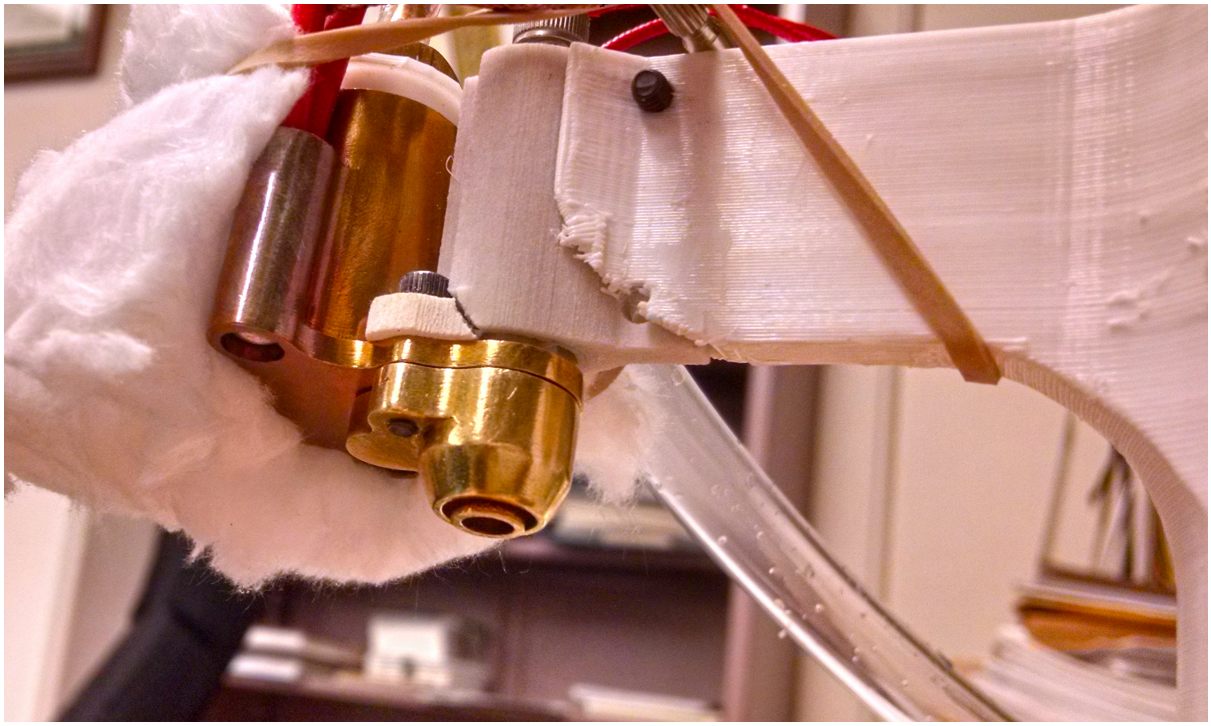

Figure 9.37: Photograph of the brass Nozzle 5 mounted.

melting point. A PVC tube was used to connect the resin nozzle to the outlet of the heated reservoir. However, the wax cooled and solidified before exiting the nozzle, in effect clogging it.

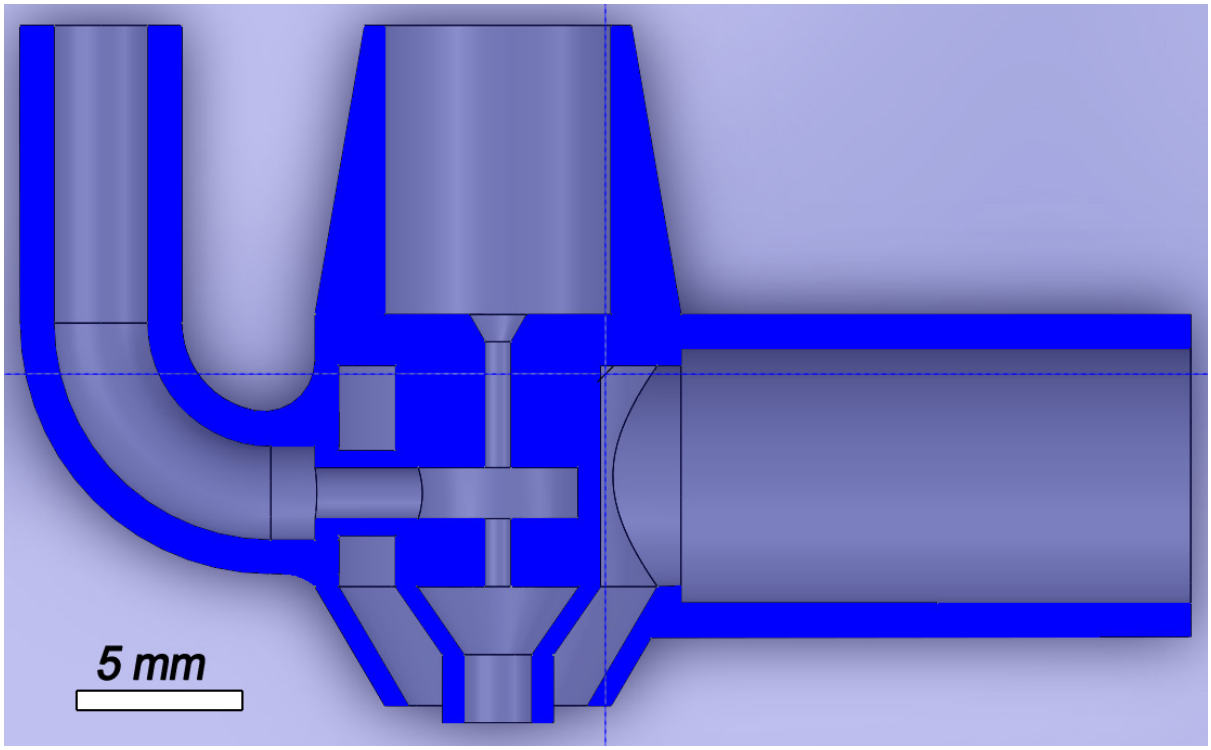

Figure 9.38: Cross section view of the resin Nozzle 5 CAD model. The heated reservoir is the same part used in the brass nozzle. A tube then transfers the melted material (e.g. wax) to the resin nozzle.

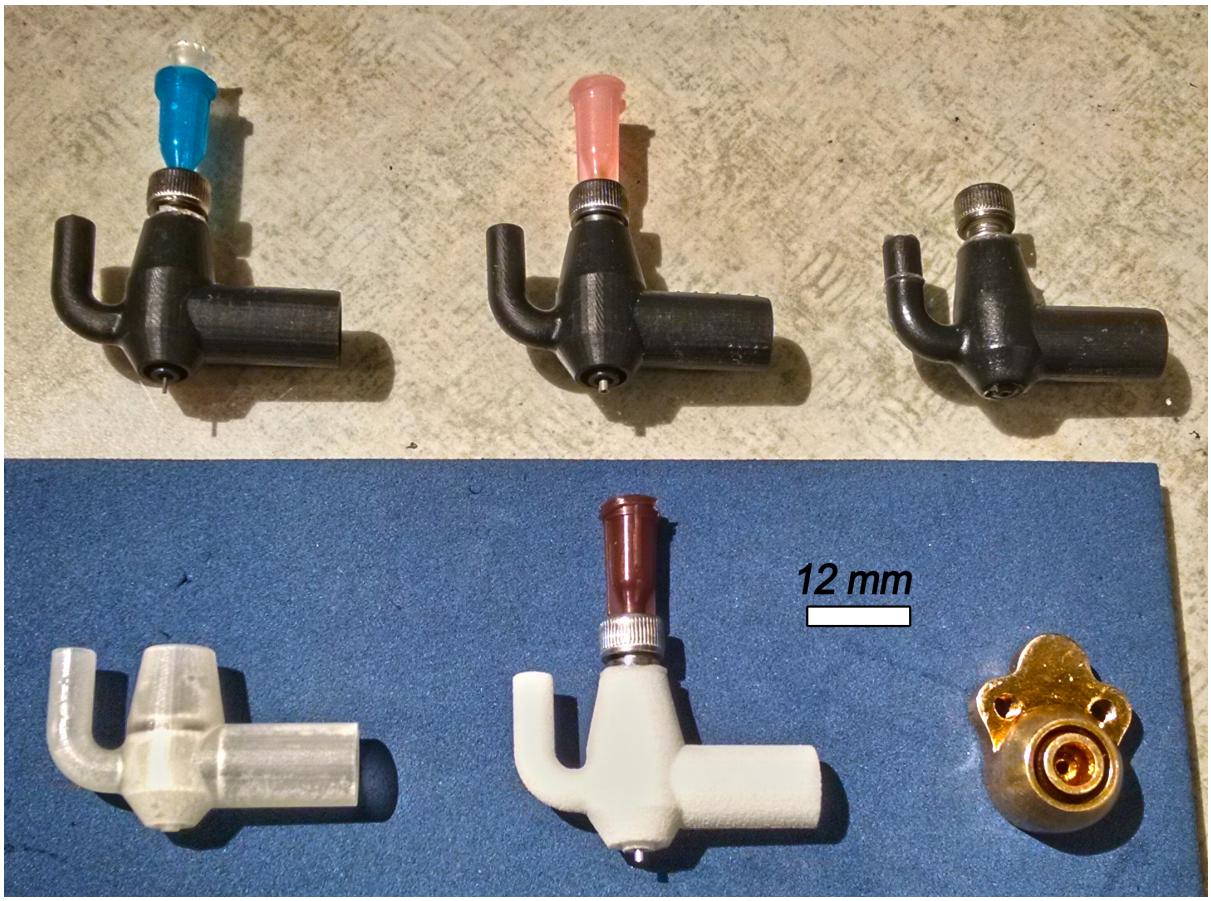

Figure 9.39: Photograph of the resin Nozzle 5 designs next to the brass nozzle in the lower right. The needle diameter and fluid outlet diameter vary among the prototypes.

9.9 Nozzle Design 6

Learning from Nozzle Design 5, PLA was abandoned and the focus shifted exclusively towards paraffin wax. A new design was created where the heated reservoir extended down into the nozzle as close as possible to the nozzle exit to ensure that hot wax was delivered right at the tip. Copper was also used this time due to its higher thermal conductivity. Since Shapeways did not offer copper prints at the time, a vendor named i.materialise (www.i.materialise.com) was used instead. The nozzle itself was printed by Shapeways using SLA/resin as before. It featured a 2.0 mm nozzle exit hole with the same 0.50 (25 gauge) mm needle as before.

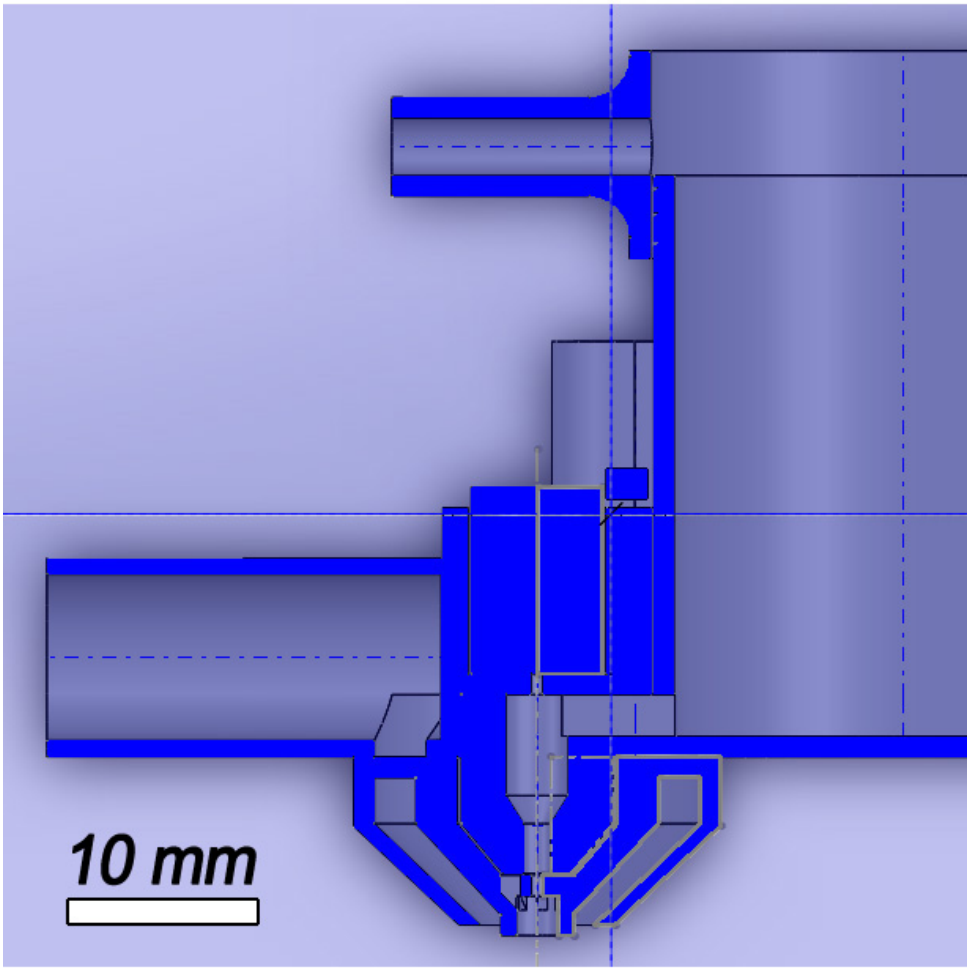

Figure 9.40: Cross section view of Nozzle Design 6 CAD model.

Again, different flow regimes were achieved depending on the system parameters. The most common was the one shown in figure (9.42), where the wax would form a small pendant drop, and, at a critical point, would turn into a turbulent stream for a short time, removing the material from the nozzle and starting the process again.

With more tweaking of the parameters, the nozzle was able to produce bubbles. Shown in figure (9.44), air was injected into the pendant drop to form a bubble, however this popped before detaching from the nozzle.

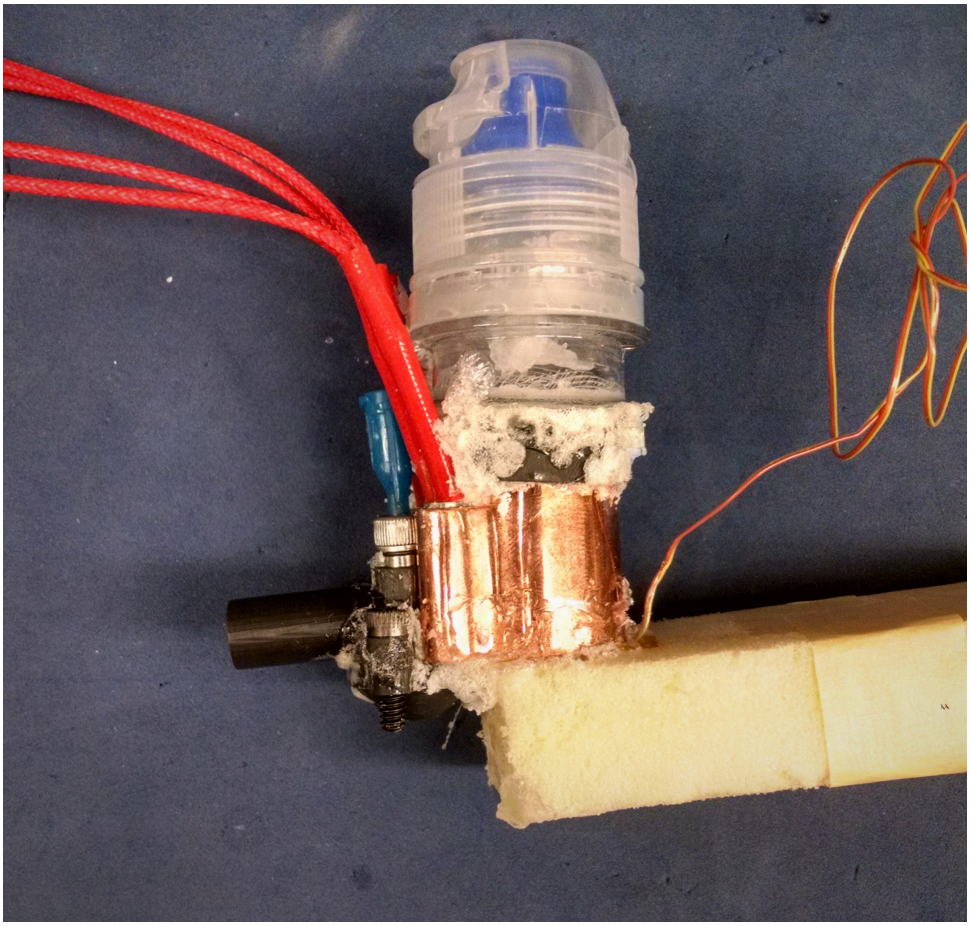

Figure 9.41: Photograph of assembled Nozzle Design 6.

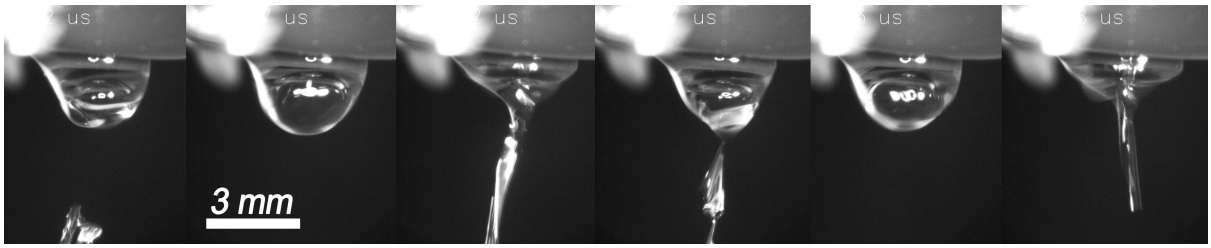

Figure 9.42: Wax forming a pendant drop followed by a quick ejection via a turbulent stream.

Perhaps the diameter to wall - thickness ratio of the bubble is too great. The next step is to reduce the nozzle exit diameter and try again.

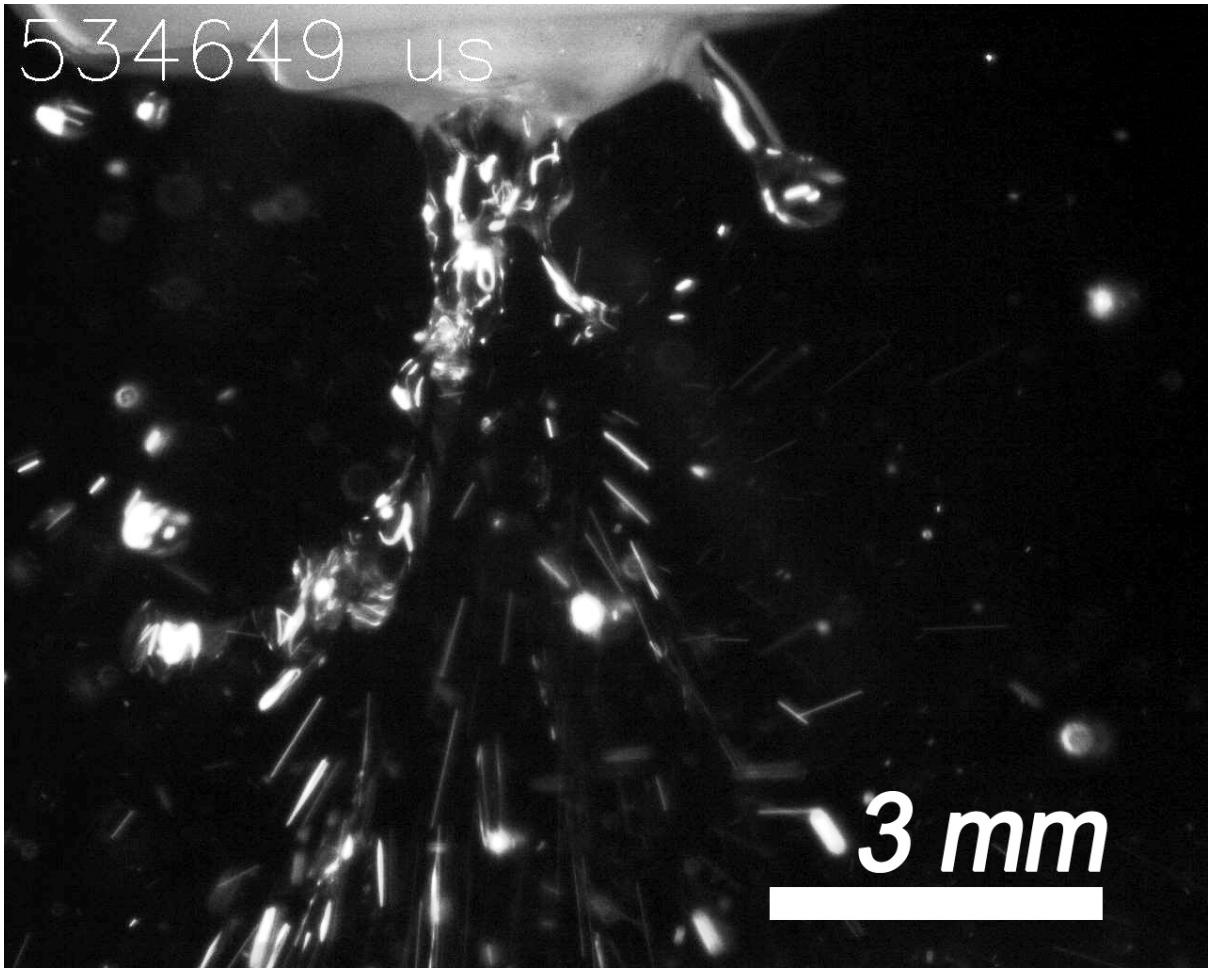

Figure 9.43: At high needle air pressures, the wax was sprayed chaotically.

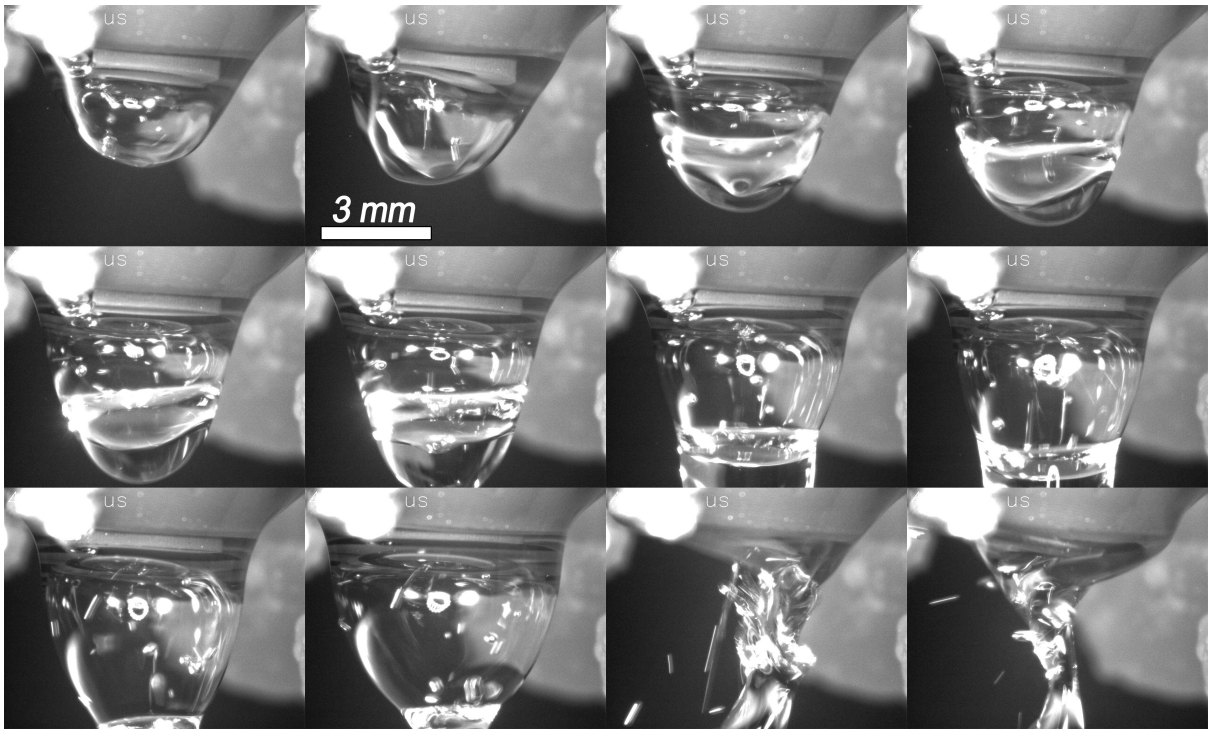

Figure 9.44: Nozzle design 6 producing wax bubbles that popped before detaching from the nozzle.

9.10 Nozzle Design 7

This design improved on the previous by eliminating the number of mating interfaces. This was done by removing the needle and instead 3D printing the geometry that delivered air to the innermost part of the nozzle. This allowed the heated reservoir to mate directly to the nozzle. An O-ring was used to further seal the liquid.

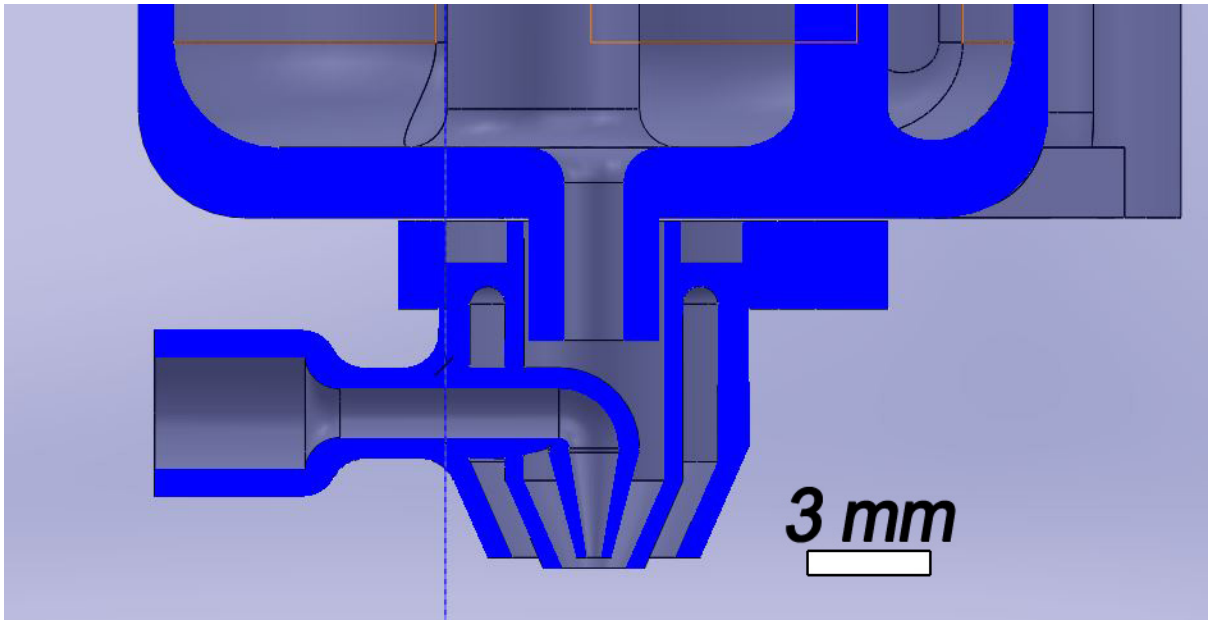

Figure 9.45: Cross section of nozzle design 7 - a simplified design by eliminating the adjustable needle and integrating the inner air channel into the 3D printed part.

This nozzle was printed by Shapeways out of epoxy resin, with a brass heated reservoir. For this experiment, milk butter butter was used as the working material since the long-chain fatty acids were hypothesized to allow stretching of a bubble wall, although not string like the high molecular weight polymers. The reservoir temperature for these experiments was 80°C unless otherwise noted in a particular figure. Large bubbles could be created in butter by lowering the liquid and air pressures.

The butter bubbles were very fragile, possibly due the the butter melt liquid being a colloid.

Since the bubbles were in a room temperature air environment, heat transfer from the thin walls would cause local freezing, leading to asymmetric mechanical properties. Once a certain part of the bubble wall becomes weaker than the rest, it would fail locally and the hole would grow until the entire bubble bursts.

Similar to the linked bubbles in soap water, butter also had a flow regime where a string of connected bubbles would continuously fall from the nozzle. However, these were less round and more elongated than their soap water counterparts.

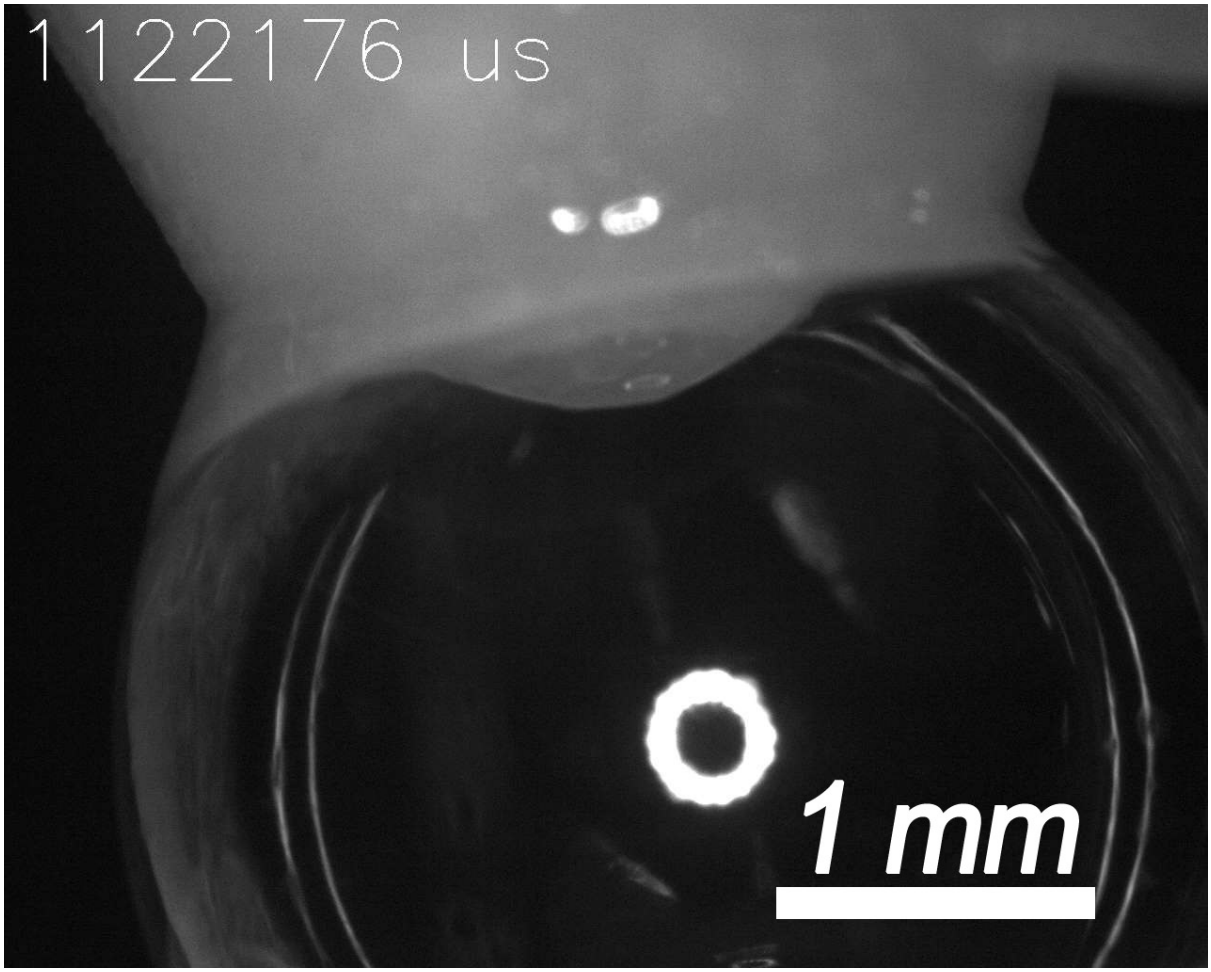

Figure 9.46: Large butter bubble by nozzle design 7. This type of bubble never detached from the nozzle.

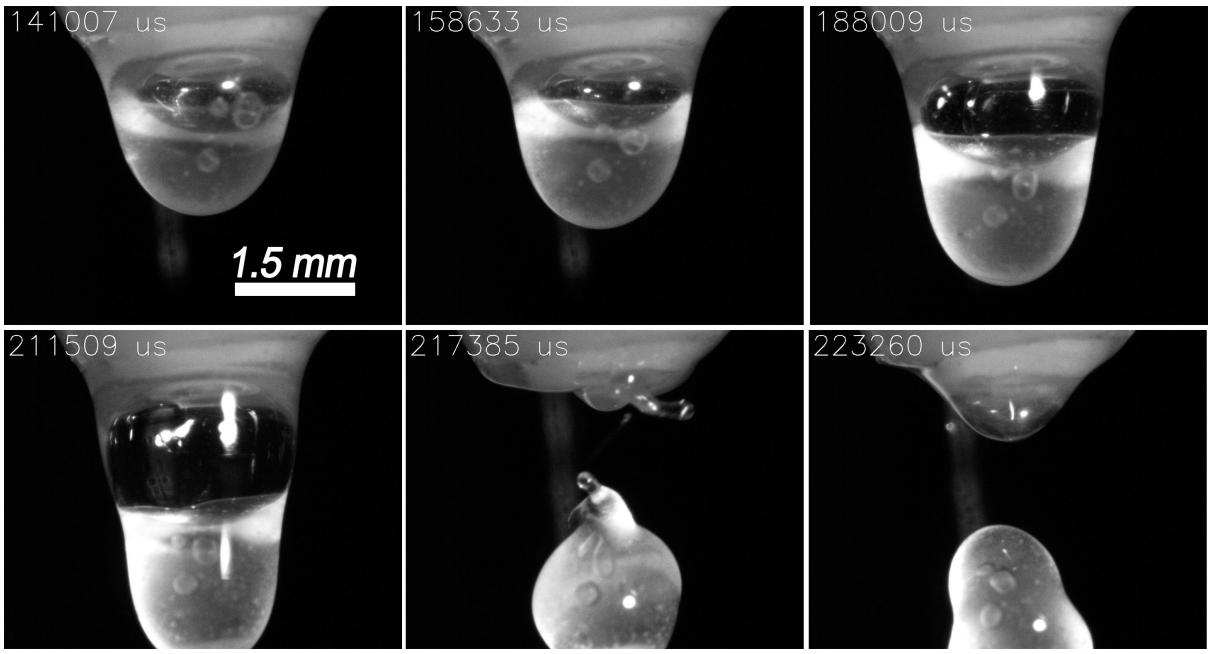

Figure 9.47: Pendant drop forming below a butter bubble and falling from the force of its weight.

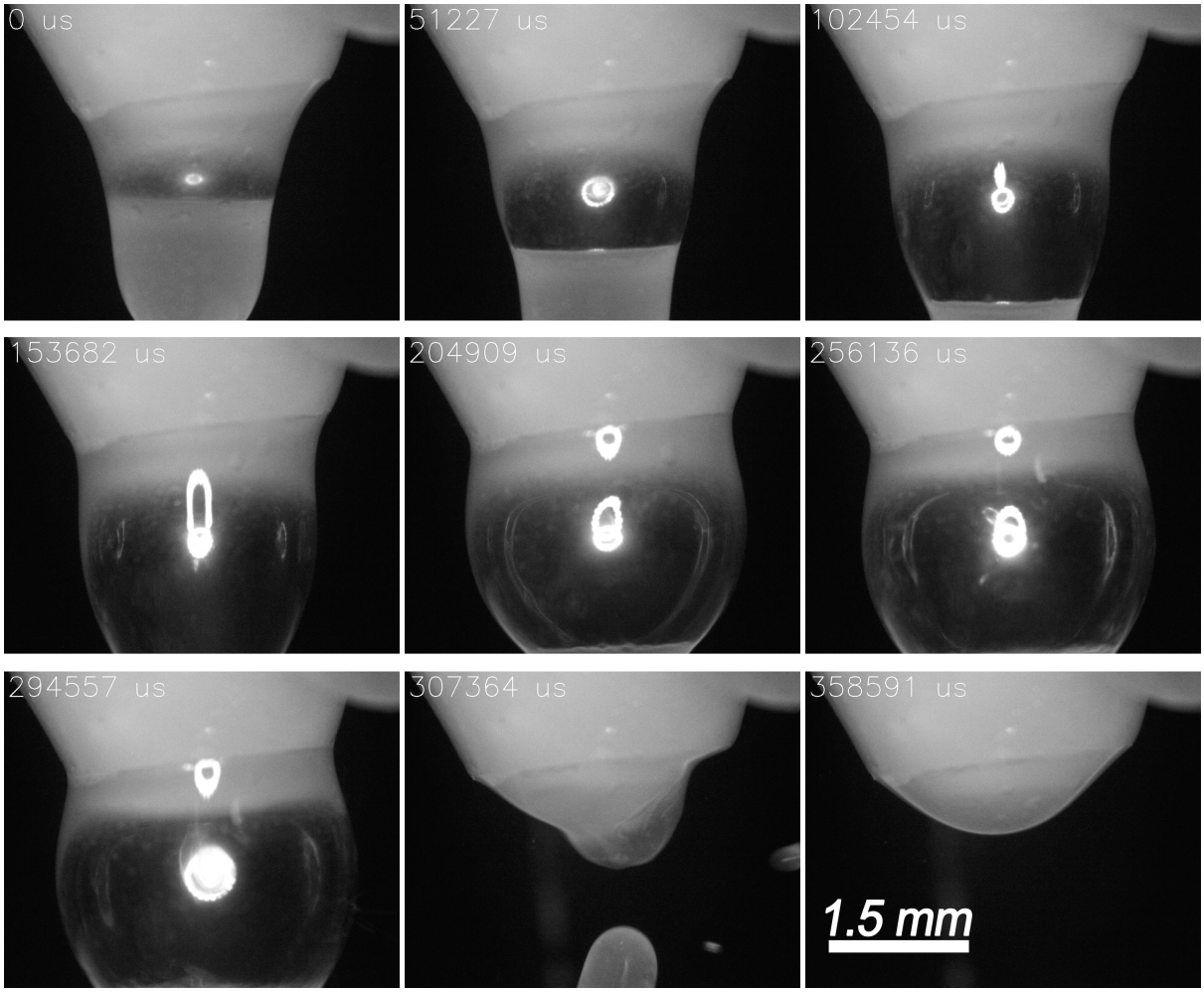

Figure 9.48: Butter bubble bursting under tension from its own weight.

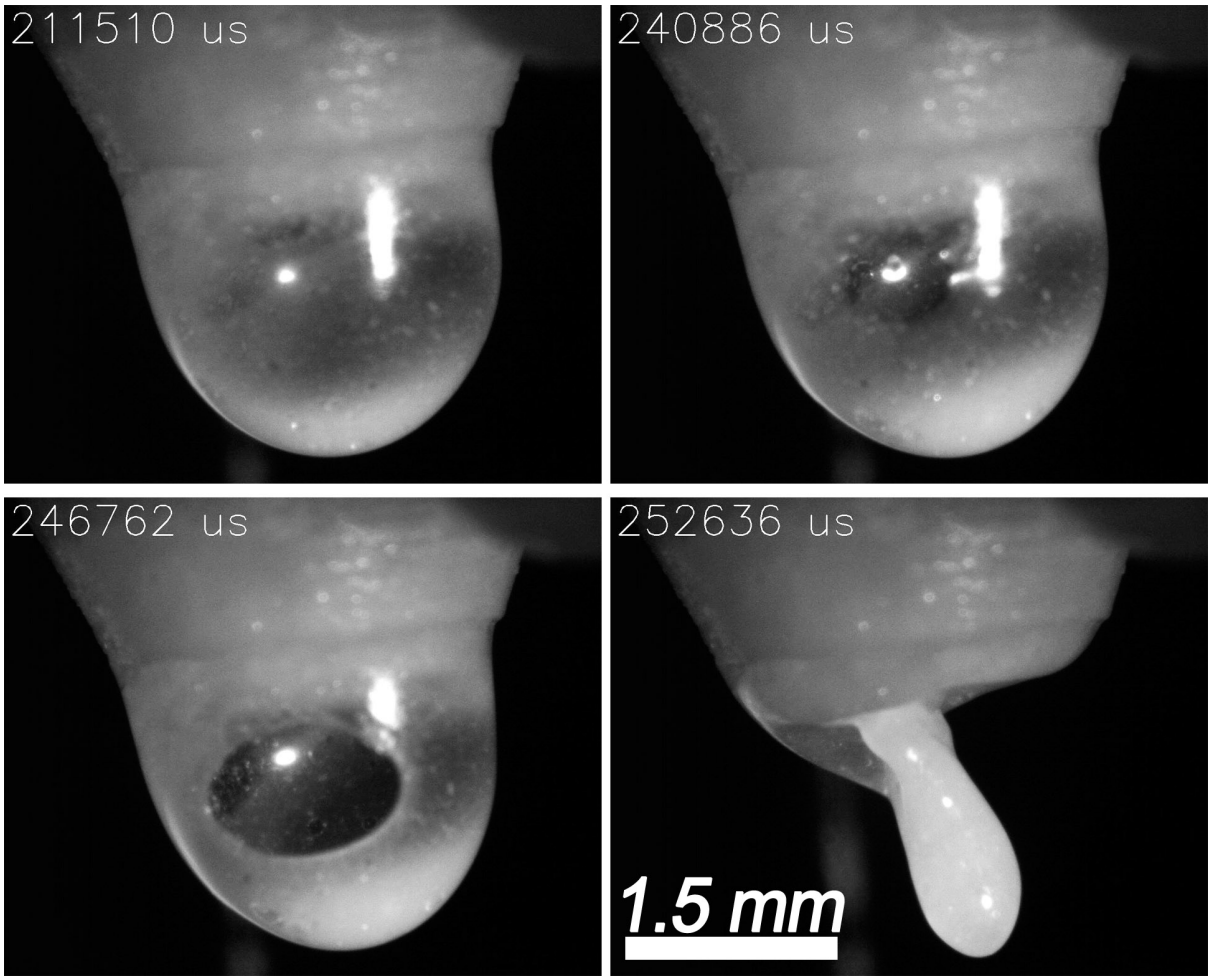

Figure 9.49: Butter bubble wall thinning leading to burst during inflation.

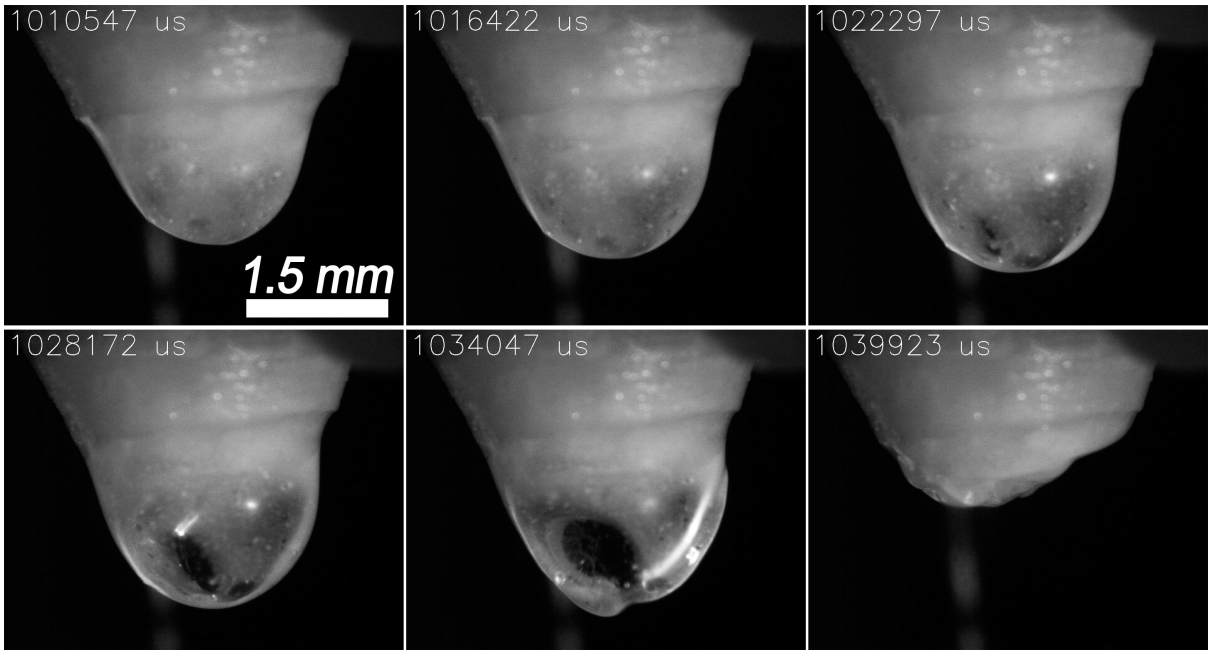

Figure 9.50: Another wall-thinning example causing bubble rupture.

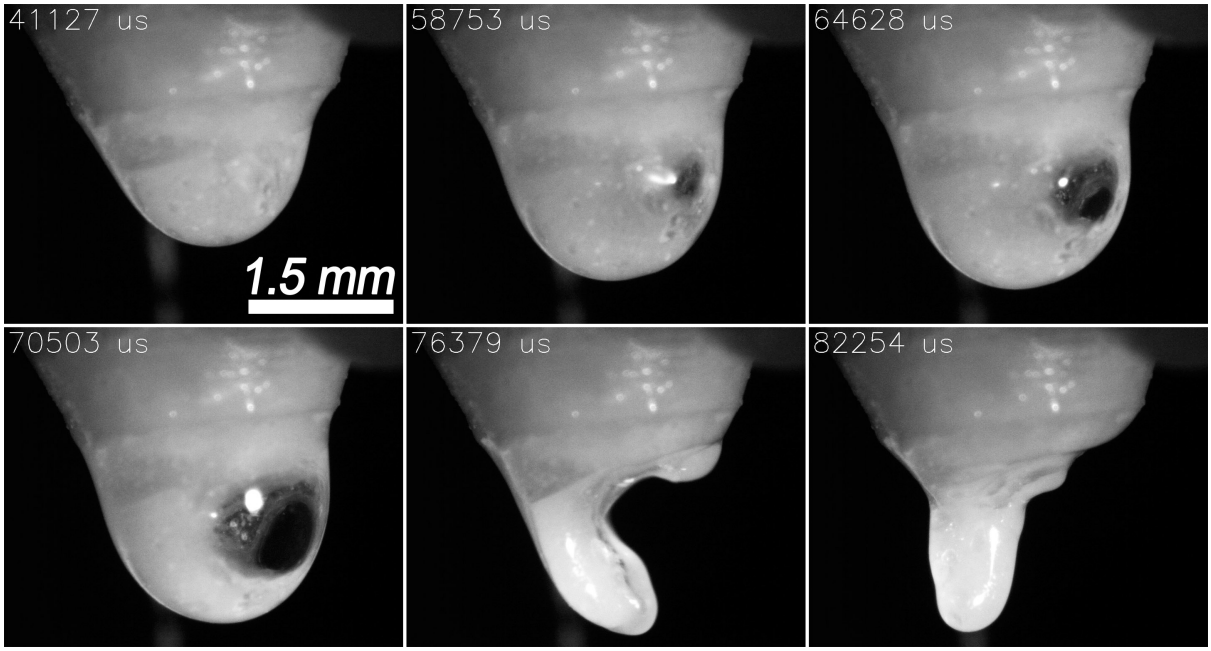

Figure 9.51: Partially frozen bubble (as evident from opacity) showing the growth of thin-wall region until bubble bursts. The reservoir temperature in this run was 60°C.

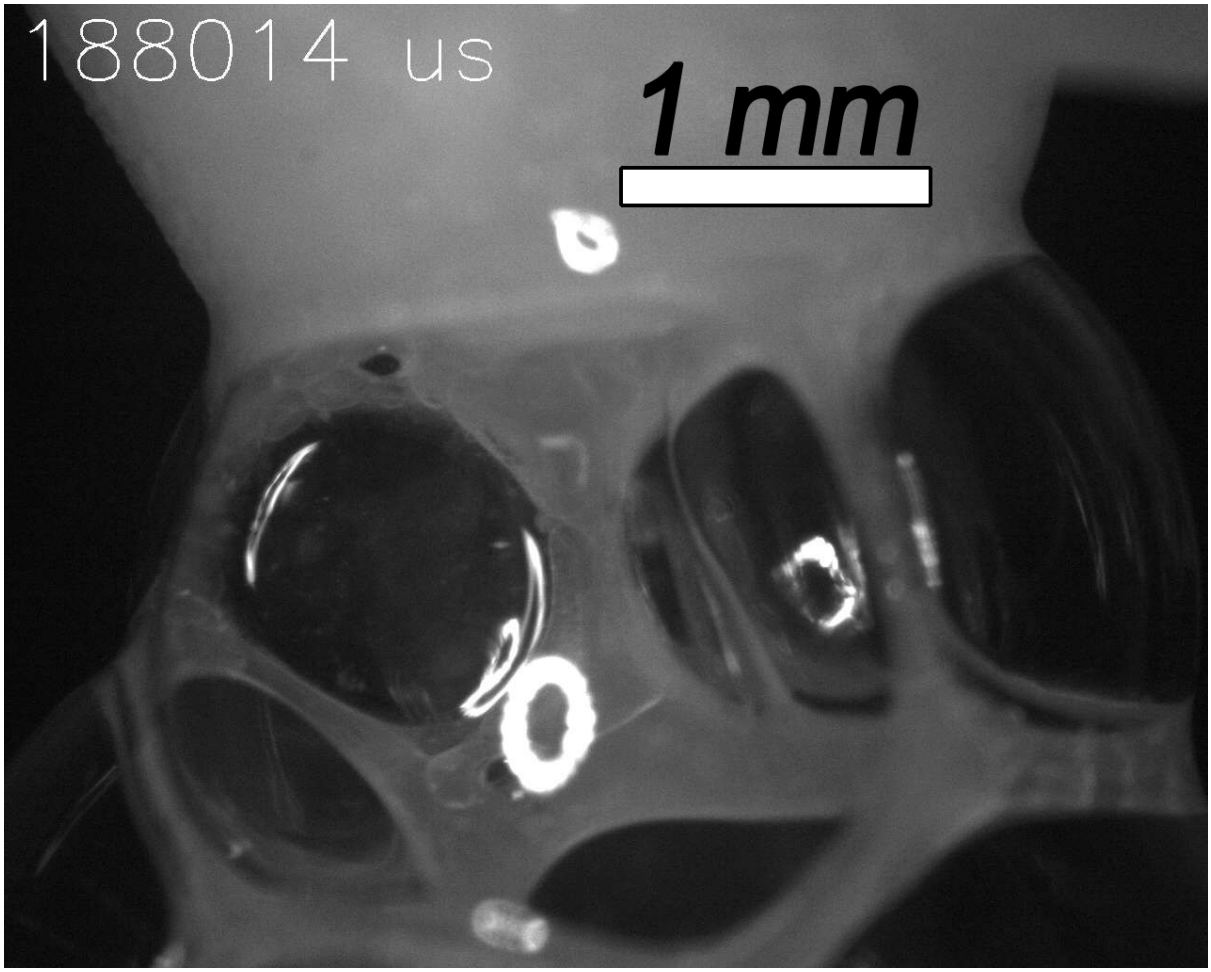

Figure 9.52: Coalescence of several butter bubbles.

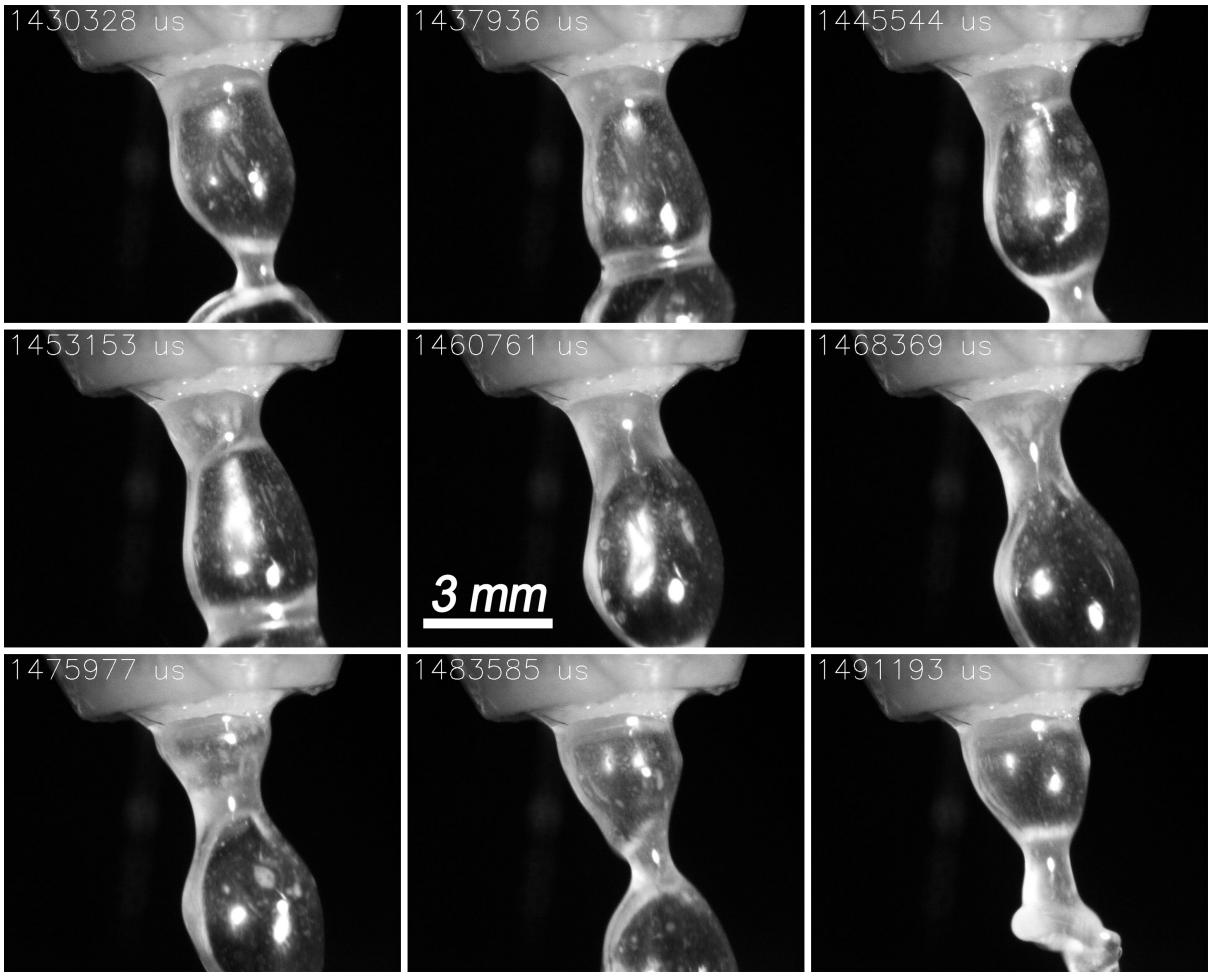

Figure 9.53: Linked butter bubbles formed at the nozzle. These are formed at a fast rate, however they burst further downstream.

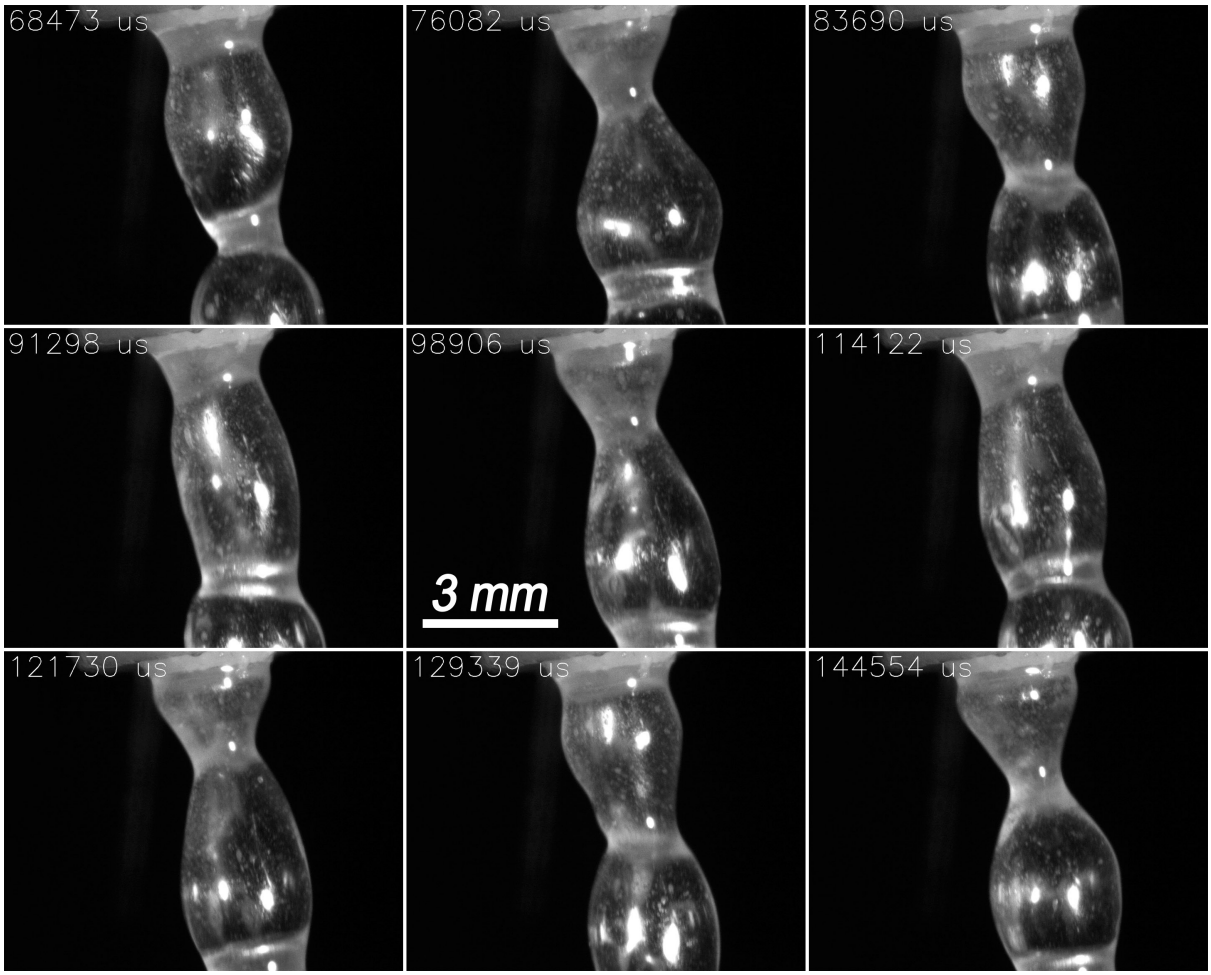

Figure 9.54: Another snapshot of linked butter bubbles ejected continuously from the nozzle.

9.11 Summary

The project concluded with nozzle design 7 running melted butter. Although individual bubbles were not created and deposited using butter, it was demonstrated that a bubble could be blown in butter up to a few millimeters in diameter. Over the course of the entire project, several observations were made. These are summarized as follows:

- High molecular weight polymers such as PLA, PET, PP, PE, PC, PMMA will not detach to create individual bubbles. Instead, the polymer will flow in a continuous string.
- Bubble walls are prone to a thickness instability during inflation - the thinnest part will stretch further, leading to rupture.
- Low molecular weight liquids such as paraffin wax, butter, and liquid latex can all form bubbles.
- Colloids tend to have more fragile bubble walls.
- Bubbles may string together (as shown in soap water and butter).
- Bubbles often burst upon ejection from the nozzle (before or after impact with a surface). This is problematic in polymers that have a long cure time - the bubbles pop before they will cure.

Overall, the project goals were fulfilled. First, it was shown that bubbles can be made in various materials, and they can be stacked to form a foam structure. However, no material was found that would maintain the bubble structure upon curing or freezing. However, this work transformed the bubble 3D printing challenge into one of finding the correct working material, having solved the mechanical engineering aspects of the nozzle and driving system. The top mechanical takeaways are as follows:

- The pressure driving the liquid through the nozzle should be small - about 0.2 psig.
- The pressure of the inner air supply was always below 2 psi. Higher pressures led to a spray flow pattern rather than bubble formation.
- The outer layer of air is critical - this must be low pressure, high flow, so a fan is ideal. This air flow helps bubbles detach from the nozzle.
- The needle position inside the nozzle is critical and can produce differing flow patterns. For bubble formation, it should be slightly recessed into the nozzle.
- Working pressures and fan speeds control the bubble size and ejection frequency.

9.11.1 Ideas for Further Work

The project left at a stage where it can be easily continued. Some ideas for future work are as follows:

- Try mixing solid particles or fibers into liquid latex to enhance the bubble wall stability.

- Use additives to decrease the viscosity of polymers.
- Design a high-precision metal nozzle to run thermoplastics - ensure that the needle and outer nozzle are concentric to tight tolerance.
- Use a secondary liquid rather than air as the inner phase.
- Make bubbles out of cross-linking polymers such as epoxy resins or UV cured resins, curing the bubble on its flight down from the nozzle.
- Try to run melted cane sugar through a 1 mm nozzle.

These are just a few of the many directions that the project can go.

Chapter 10

Conclusions and Recommendations for Future Work

10.1 Summary

Each chapter in the dissertation is concisely outlined here for reference.

10.1.1 Chapter 1 - Nanoporous Polyetherimide

In this chapter, I introduced nanoporous polyetherimide (PEI), its open-porous structure, and solid skin along with some background and prior work on the solid-state foaming process and the research trend of reducing cell size down to the nanometer scale. I then established the mathematical foundation for analyzing a fluid permeable nanoprous structure. The assumptions (e.g. isotropic core, continuum treatment of fluid in the pores, etc.) are established, as well as methods for estimating the permeability, tortuosity, and characteristic pore diameters of such structures. Further, an experimentally convenient result on reversing the sample position in a testing apparatus is mathematically proved.

10.1.2 Chapter 2 - Experimental Techniques

A complete process is developed that extends the conventional solid-state process to produce flat, uniform, PEI samples with low variance. Surface blistering was eliminated by using thinner (0.5 mm vs $>$1.0 mm) raw PEI sheets. However, this introduced a desorption problem since the processing window was reduced to around 5 minutes before the skin/transition regions grew (by virtue of diffusion) to undesirable lengths. This was solved by utilizing liquid nitrogen after samples' removal from the pressure vessel to slow desorption and extend the processing time to over 3 hours. Thus, a large batch of samples could be produced from a single vessel pressure cycle.

Techniques for removing the solid skin were also presented in this chapter, namely machining and needle piercing. Extensive studies were performed on both, however the needle-piercing was presented as the go-to method in this research due to its speed and consistency.

The apparatuses for permeability, diffusion, and intrusion pressure measurements are discussed, with both development and use are outlined. Lastly, image processing techniques for

obtaining necessary information on sample geometry are presented, with the MATLAB code referenced in the appendix.

The techniques developed here serve as the platform for future researchers to continue work in this field, such as in the specific areas outlined in the future work section below.

10.1.3 Chapter 3 - Fluid Flow Through a Cut Skin

Since the skin-removal techniques developed in chapter 2 selectively remove portions of the skin, obtaining permeability from flow data is nontrivial. The partial differential equation (with Dirichlet and Neumann mixed boundary conditions) was solved to correctly model the flow and avoid the large errors incurred if a 1D solution is assumed (such as in ASTM D6539-13). It was found that, by assuming 1D flow, the computed permeability would be 3 times higher than actual (error of 200%).

A complete algorithm was written in MATLAB which takes a photograph of a machined-skin specimen and correctly computes the permeability by solving the 2D PDE, with length parameters passed in by another image processing algorithm which analyzes the sample geometry.

This model was verified experimentally by creating 3 independent samples (from the same production batch) machined by different techniques and in different patterns, and the algorithm computed the same permeability (with low variance) for all three. This model and software code is critical for computing permeability from experimental data and is available in the appendix.

10.1.4 Chapter 4 - Permeability and Diffusivity Data

This chapter uses the tools developed above to obtain data on permeability and diffusivity of PEI nanofoams. Using the data on diffusivity and porosity, tortuosity was also computed and found to be between 20-40. Permeability and diffusivity were found to be comparable to volcanic rock, limestone, and sandstone. Calculations on the uncertainty in the measurements are presented, showing an approximately 13% total error due to uncertainties in measurements of flow rate, hole size, pressure, and thickness.

10.1.5 Chapter 5 - Transparent Foams

Perhaps one of the greatest discoveries in my Ph.D, I was able to make PEI nanofoams with cells so small that light scattering is diminished. The available SEM was not able to clearly image the structure, however we did see voids in the 10 nm range. These nanostructures are also open-porous, so permeability and diffusivity data was obtained as before. Interestingly, the data on these transparent nanostructures do not follow the continuum models that explain the quadratic relationship between permeability/diffusivity and porosity - rather, the trends seem erratic. With voids approaching the molecular scale, it is not a surprise that the continuum and Newtonian assumptions are not valid.

Light transmission data was obtained in the UV-IR range. It is shown that, for some wavelengths in the infrared, the foams are more transmissive than raw PEI. The hypothesis is that the yellow-amber tint is dispersed due to the polymer's expansion (since these samples are more than 50% void).

Transparent foams have been a fantasy since the early days of the microcellular process at MIT when my Ph.D advisor was researching it under professor Nam Suh. Now, it has been realized. Although these foams are not optically clear as PMMA or glass (and neither is raw PEI), this milestone is a breakthrough in transparent polymer foams. This is similar to Aerogel, a nanostructure that is also open-cell and transparent. However, Aerogel is silica based, while PEI is a thermoplastic polymer, and the first to be transparent at 50% relative density according to our literature search.

10.1.6 Chapter 6 - Thermal Conductivity

This chapter begins with the construction of a guarded-hotplate apparatus for measuring thermal conductivity. The dynamic system modeling and control algorithms are presented. Thermal conductivity of both transparent and opaque PEI foam samples are measured to be roughly half of raw PEI (which is 0.22 W/m-K). Counterintuitively, the thermal conductivity of the transparent foams (with much smaller cell size) is larger than that of an opaque foam of similar density.

A significant result of the work from this chapter is the design of an accurate thermal conductivity apparatus for about $200 in parts. Commercial systems were quoted at over $20,000 and up to $60,000, which is the reason that compelled me to construct my own. I then realized that there may be a demand for lower-cost, less sophisticated systems. Thus, I may design and sell thermal conductivity systems as a hobby side-project after graduation.

10.1.7 Chapter 7 - Applications

Some exploratory work was performed on possible applications of the nanoporous PEI structure. It was found that water would not penetrate the nanopores under 800 psi water pressure. This is due to PEI being hydrophobic and the pore size very small, a similar effect exploited by commercial membrane GoreTEX made by company W.L. Gore. Commercially available GoreTEX (which is based on a porous PTFE membrane) is waterproof to less than 3 atmospheres (according to their data sheet); however, it has higher permeability than nanoporous PEI. For this reason, we looked into underwater venting applications where the pressures were much higher (up to 500 meters underwater) and the flow rates required were smaller.

There was also work on thermoforming PEI foams - it was found that even with tool temperatures at 230°C, the nanoporous structure was preserved after thermoforming. This is a useful result for applications that require geometries other than flat sheets.

Since the permeability range (found in Chapter 4) was rather narrow, efforts were made to expand it by using a two-step foaming process. First, samples were saturated at low pressure (1 MPa) and foamed to yield large, closed cell bubbles. These samples were then saturated at higher pressure (5 MPa) and foamed at the usual temperatures 170-200°C with the intent of connecting the large closed cells into a more permeable open porous structure. It was found that this method produced open-porous structures with lower permeability than before. However, a variety of very interesting micro and nano-structures were produced in this two-step process. Perhaps with future work, these structures can be tuned to yield higher permeability foams, or they could be tailored for other niche applications.

10.1.8 Chapter 8 - Additional Experiments

This section is a collection of work I did at the beginning of my Ph.D while exploring possible research projects. The subjects are too numerous and disjoint to summarize here, but the ideas are as follows:

- Time Dependent Sorption - I built a pressure vessel system where the pressure could be varied as a function of time such that the concentration profile (as a function of distance in the thickness direction) could be controlled to a desired function. The necessary boundary condition (gas pressure) was computed by an algorithm that iteratively solved the diffusion differential equation in time. Further, a map from concentration to bubble size attempted to be computed experimentally using the model and SEM data.

- High Impact Strength Foams - Filling closed-cell bubbles with water would make the foam much stronger in compression due to the liquid pressure resisting the load. In fact, some cell walls could be in tension while the foam is compressed.

- Laser-induced Foaming - This idea was simple: use a laser beam as the heating source for foaming saturated PEI samples. This method produced visually spectacular and puzzling results, which are still unexplained.

10.1.9 Chapter 9 - 3D Printing Bubbles

This project may seem disjoint from the rest of the research in this thesis - the reason is that it was a 9 month grant sponsored by Amazon on the unique idea I proposed: to 3D print bubbles. The vision was a nozzle that jets small (around 1 mm) polymer liquid bubbles which are internally filled with air. These bubbles were to be deposited to form a 3D object. Since only the surface skin is polymer and the rest is air, the objects produced with this technique would be very lightweight, highly collapsible, and soft. While there is still much work that could be done in terms of polymer selection and parameter tuning, the feasibility of this idea was realized.

10.2 Topics of Future Investigation

10.2.1 Explore Nanostructures in Other Polymers

Since the solid-state process extends to many thermoplastics, further research can be extended to these materials, based on the same techniques I used on PEI. Polyimide (PI) is similar to PEI and especially of interest in the electronics industry due to its high glass transition temperature of 385°C.

10.2.2 Explore Nanoparticle Transport Through the Pores

For some applications such as filtration or composites, it will be necessary to load nanoparticles into the porous structure. Since the tortuosity is high and the pore sizes (although not clearly defined) small, it is not known what size particles can or cannot pass through.

10.2.3 Try Coating the Cell Walls with Metals

High surface area conductors are of interest for applications such as battery electrodes and supercapacitors. Coating the cell walls with a conductor using electroplating or electroless plating is a worthwhile effort.

10.2.4 Carburizing PEI Nanofoams

An approach to turn polymers into carbon is by quickly heating to very high temperatures in an oxygen-free environment. Thus, PEI nanofoams could be a turned into conductive carbon, hypothetically with a similar high surface area pore structure. This material would make excellent electrodes for high discharge rate batteries as well as many other applications [14].

10.2.5 Scale-up of PEI Nanofoam Production

The PEI foams I made were disks of approximately 1 inch diameter. Expanding the dimension to production-scale is nontrivial since effects such as friction increase nonlinearly with the characteristic length dimension. Other methods such as heating using infrared under tension, hot air ovens, or hot rollers could be explored.

10.2.6 Imaging Transparent PEI Structures

Since the current SEM techniques were inadequate in showing the porous structure, other imaging methods must be explored. TEM, AFM, SHIM, or more modern SEM techniques could be explored.

10.2.7 Exploring the Non-continuum Behavior in Transparent Nanofoams

Transport properties in the transparent nanoporous PEI samples were shown to not follow the classical fluid models that the opaque PEI foams did. For example, permeability and diffusivity are not quadratic functions of porosity for the transparent samples. These trends, now unexplained, could get investigated. Perhaps the pores are so small that molecular effects are significant. With the use of better imaging and understanding of the porous structure, these trends may be explained.

10.2.8 Further Exploring Applications

Since a lot of work has been done to show potential applications, it will be worthwhile to further develop one or more of these ideas towards a real product. A few directions are: underwater enclosures that must be vented and waterproofed (utilizing the waterproof-breathable property or nanoporous PEI), slow release systems (utilizing the diffusivity properties), thermoforming into shapes such as a mobile phone case, and further applications in energy storage.

10.2.9 Laser-Induced Foaming

Carrying on the preliminary experiments in laser beam induced foaming, this problem is interesting from both theoretical and practical standpoints. First, the new bubbles produced by laser

pulses have shapes that were never before seen in the solid-state foaming process. The cause of this new structure should be understood. Also, creating porous channels inside a PEI sheet using a laser could be useful for microfluidics and biochemical applications.

10.2.10 3D Printing

Perhaps one of the most exciting areas of my work, this technology has a lot of room for future work. Although a detailed list of suggestions was already given in the previous chapter, the recommendation here is to explore different application areas of this technology.

10.3 Remarks

I spent almost 10 years continuously as a full time student at the University of Washington, obtaining a Bachelor of Science in Mechanical Engineering with a minor in Mathematics, a Master of Science in Mechanical Engineering with thesis, and now a Ph.D all back-to-back. This time went by rather quickly and certainly doesn't feel like a decade - I still vividly remember my first quarter at the university, excited and disoriented by the vast size of the university. Even before turning 10 years old, I wanted to make cars, and in my teenage years I dreamed about designing my own rotary combustion engine. I came up with dozens of designs, mostly terrible ones, and knew I must study thermodynamics and kinematics in order to improve. This is why I chose Mechanical Engineering, and choosing the doctoral major area as Energy & Fluids rather than Mechanics & Materials as the other students in the lab. Graduate school was fun, interesting, and the best years of my life.

Bibliography

[1] V. Kumar and J. E. Weller. Production of microcellular polycarbonate using carbon dioxide for bubble nucleation. *Journal of Manufacturing Science and Engineering*, 116(4):413–420, 1994.

[2] Vipin Kumar and Nam P Suh. A process for making microcellular thermoplastic parts. *Polymer Engineering & Science*, 30(20):1323–1329, 1990.

[3] Vipin Kumar, John E Weller, and Romano Montecillo. Microcellular pvc. *Journal of Vinyl Technology*, 14(4):191–197, 1992.

[4] Vipin Kumar. Microcellular polymers: novel materials for the 21st century. *Cellular Polymers*, 12(3):207–223, 1993.

[5] J. S. Colton and N. P. Suh. The nucleation of microcellular thermoplastic foam with additives: Part i: Theoretical considerations. *Polymer Engineering & Science*, 27(7):485–492, 1987.

[6] John Weller. *The Effects of Processing and Microstructure on the Tensile Behavior of Microcellular Foams*. Dissertation, University of Washington, 1996.

[7] B. Krause, H. J. P. Sijbesma, P. Mnkl, N. F. A. van der Vegt, and M. Wessling. Bicontinuous nanoporous polymers by carbon dioxide foaming. *Macromolecules*, 34(25):8792–8801, 2001.

[8] B. Krause, K. Diekmann, N. F. A. van der Vegt, and M. Wessling. Open nanoporous morphologies from polymeric blends by carbon dioxide foaming. *Macromolecules*, 35(5):1738–1745, 2002.

[9] Dustin Miller. *Characterization of polyetherimide carbon dioxide system and mechanical properties of high relative density polyetherimide nanofoams*. PhD thesis, University of Washington, 2007.

[10] Dustin Miller, Pavee Chatchaisucha, and Vipin Kumar. Microcellular and nanocellular solid-state polyetherimide (pei) foams using sub-critical carbon dioxide i. processing and structure. *Polymer*, 50(23):5576–5584, 2009.

[11] Huimin Guo and Vipin Kumar. Solid-state poly(methyl methacrylate) (pmma) nanofoams. part i: Low-temperature co2 sorption, diffusion, and the depression in pmma glass transition. *Polymer*, 57:157–163, 2015.

[12] Huimin Guo and Vipin Kumar. Some thermodynamic and kinetic low-temperature properties of the pc-co2 system and morphological characteristics of solid-state pc nanofoams produced with liquid co2. *Polymer*, 56:46–56, 2015.

[13] J. W. Labadie, J. L. Hedrick, V. Wakharkar, D. C. Hofer, and T. P. Russell. Nanopore foams of high temperature polymers. *IEEE Transactions on Components, Hybrids, and Manufacturing Technology*, 15(6):925–930, 1992.

[14] BryceC Tappan, StephenA Steiner, and ErikP Luther. Nanoporous metal foams. *Angewandte Chemie International Edition*, 49(27):4544–4565, 2010.

[15] B Krause, ME Boerrigter, NFA Van der Vegt, H Strathmann, and M Wessling. Novel open-cellular polysulfone morphologies produced with trace concentrations of solvents as pore opener. *Journal of Membrane Science*, 187(1):181–192, 2001.

[16] J. E. Martini. *The production and analysis of microcellular foam*. PhD thesis, MIT, 1981.

[17] Brian Aher. Towards battery separator films: Production of solid-state pei nanofoams using supercritical co2. *University of Washington*, Master's Thesis, 2012.

[18] H. Bruus. *Theoretical Microfluidics*. OUP Oxford, 2008.

[19] P. Chadwick. *Continuum Mechanics: Concise Theory and Problems (Dover Books on Physics)*. Dover Publications, 2012.

[20] J. Bear. *Dynamics of Fluids in Porous Media*. Dover, 1972.

[21] S. P. Neuman. Theoretical derivation of darcy's law. *Acta Mechanica*, 25(3-4):153–170, sep 1977.

[22] A. C. Liakopoulos. Darcy's coefficient of permeability as symmetric tensor of second rank. *International Association of Scientific Hydrology. Bulletin*, 10(3):41–48, 1965.

[23] Vipin Kumar Andrei Nicolae. Guidelines for machining the skin of microcelullar and nanocellular foams to access the core. In *Conference Proceedings*, Seattle WA, September 2016. SPE FOAMS.

[24] J DESPOIS and A MORTENSEN. Permeability of open-pore microcellular materials. *Acta Materialia*, 53(5):1381–1388, mar 2005.

[25] zer Bağcı and Nihad Dukhan. Experimental hydrodynamics of high-porosity metal foam: Effect of pore density. *International Journal of Heat and Mass Transfer*, 103:879–885, dec 2016.

[26] Alexander D Poularikas. *Transforms and applications handbook*. CRC press, 2010.

[27] A.E. Scheidegger. *The physics of flow through porous media*. University of Toronto Press, 1974.

[28] D.V. Schroeder. *An Introduction to Thermal Physics*. Addison Wesley, 2000.

[29] Y.A. Cengel. *Thermodynamics: An Engineering Approach*. McGraw-Hill College, 1998.

[30] J. Crank. *The Mathematics of Diffusion*. Oxford Science Publications, 1975.

[31] Paolo Fornasini. *The Uncertainty in Physical Measurements: An Introduction to Data Analysis in the Physics Laboratory*. Springer, 2008.

[32] W. F. Brace. Permeability of crystalline and argillaceous rocks. *International Journal of Rock Mechanics and Mining Sciences & Geomechanics Abstracts*, 17(5):241–251, 1980.

[33] Thomas B. Boving and Peter Grathwohl. Tracer diffusion coefficients in sedimentary rocks: correlation to porosity and hydraulic conductivity. *Journal of Contaminant Hydrology*, 53(12):85–100, 2001.

[34] Sheng Peng, Qinhong Hu, and Shoichiro Hamamoto. Diffusivity of rocks: Gas diffusion measurements and correlation to porosity and pore size distribution. *Water Resources Research*, 48(2):n/a–n/a, 2012.

[35] E.M. Lifshitz and L.P. Pitaevski. *Physical kinetics*. Butterworth-Heinemann, 1981.

[36] T. R. Marrero and E. A. Mason. Gaseous diffusion coefficients. *Journal of Physical and Chemical Reference Data*, 1(1):3–118, 1972.

[37] J Kestin, K Knierim, EA Mason, B Najafi, ST Ro, and M Waldman. Equilibrium and transport properties of the noble gases and their mixtures at low density. *Journal of Physical and Chemical Reference Data*, 13(1):229–303, 1984.

[38] Amir Polak, Ronit Nativ, and Rony Wallach. Matrix diffusion in northern negev fractured chalk and its correlation to porosity. *Journal of Hydrology*, 268(14):203–213, 2002.

[39] Ronald R. Schnabel Gary J. Jellick. Field determination of gas diffusion coefficients in surface solids. *Soil Science Society of America*, 1985.

[40] Cyril Isenberg. *Science of Soap Films and Soap Bubbles*. Tieto Ltd, 1978.

[41] C. V. Boys. *Soap Bubbles*. DOVER PUBN INC, 2012.

[42] Alfonso M. Gañán-Calvo and José M. Gordillo. Perfectly monodisperse microbubbling by capillary flow focusing. *Physical Review Letters*, 87(27), dec 2001.

[43] Shu-Che Peng. Microfluidic drop formation with polymer plastics. Master's thesis, National Cheng Kung University, 2008.

Appendices

Appendix A

Experimental Techniques

A.1 Microcontroller Code for Parallel Plate Heater

```
#include <SPI.h>
#include "Adafruit_MAX31855.h"

#define DO1 2
#define CS1 3
#define CLK1 4
Adafruit_MAX31855 TC1(CLK1, CS1, DO1);

#define DO2 5
#define CS2 6
#define CLK2 7
Adafruit_MAX31855 TC2(CLK2, CS2, DO2);

#define heat1 8
#define heat2 9
#define overheat 225

int temp1;
int temp2;
double temp1double;
double temp2double;
int set1 = 0;
int set2 = 0;
int tol = 2; // gets to within this in while loop, then PWM.
long time0 = 0;

//declare functions
void safetyandread();
int readTC(int x);
void printtemp();

void setup()
{
```

```
 pinMode(heat1,OUTPUT);
 pinMode(heat2,OUTPUT);
 digitalWrite(heat1,LOW);
 digitalWrite(heat2,LOW); // for safety

 Serial.begin(9600);
 while(!Serial)
 Serial.setTimeout(10000000);
 if(Serial) Serial.println("Connected");
 Serial.println("Enter Temperature 1.");
 while(set1 < 10){set1 = Serial.parseInt();}
 Serial.println(set1);
 Serial.println("Got it. Now enter Temperature 2.");
 while(set2 < 10){set2 = Serial.parseInt();}
 Serial.println(set2);
 Serial.println("Got it. Control started.");
 } // end setup

void loop()
{
 safetyandread();

 while(temp1 < set1-tol && temp2 < set2-tol)
 {
  safetyandread();
  digitalWrite(heat1,HIGH);
  digitalWrite(heat2,HIGH);
  printtemp();
 }// end while loop

 digitalWrite(heat1,LOW);
 digitalWrite(heat2,LOW);


 safetyandread();
 if(temp1<set1){digitalWrite(heat1,HIGH);
      delay(200); digitalWrite(heat1,LOW);}
 if(temp2<set2){digitalWrite(heat2,HIGH);
      delay(200); digitalWrite(heat2,LOW);}

  printtemp();

}//loop end


// functions:
int readTC(int x)
{
 int Tout;
```

```
 if(x == 1) Tout=0.8188*TC1.readCelsius()+6.1452;
 if(x == 2) Tout=0.8188*TC2.readCelsius()+6.1452;
 return Tout;
}

// checks for overheat, and updates temp1 and temp2 ints
void safetyandread()
{
 temp1double = TC1.readCelsius();
 temp2double = TC2.readCelsius();
 temp1 = readTC(1);
 temp2 = readTC(2);

 if(isnan(temp1double) || isnan(temp2double))
   {
    digitalWrite(heat1,LOW); digitalWrite(heat2,LOW);
    Serial.println("ERROR! CHECK THERMOCOUPLES AND RESET ARDUINO.");
    while(1) {digitalWrite(heat1,LOW);
      digitalWrite(heat2,LOW); delay(100);}
   }


 if(temp1 > overheat || temp2>overheat)
   {
    digitalWrite(heat1,LOW); digitalWrite(heat2,LOW);
    Serial.println("OVERHEATED!");
    Serial.print("Temp 1: ");
    Serial.println(temp1);
    Serial.print("Temp 2: ");
    Serial.println(temp2);
    delay(10000);
   }
} // end safetyandread function


void printtemp()
{

    if(millis()-time0 > 1500)
    {
  Serial.print("Temp 1: ");
  Serial.println(temp1);
  Serial.print("Temp 2: ");
  Serial.println(temp2);
  Serial.println(" ");
  time0 = millis();
    }
}
```

A.2 Image Processing MATLAB code

```
clear all; clc; close all;
treshold = 150; %set the brightness
pixelsPerMm = 238; %get from photoshop

a = rgb2gray(imread('needleCounts.jpg'));
b = (a < treshold); %make it logical black and white

blob1=regionprops(b,'Area');
AreasCell=struct2cell(blob1)'; %switch data type
AreasVector=cell2mat(AreasCell(:,1)); %area of eac h blob

logicFilter=AreasVector > 10; % areas greater than 10 square pixels
m=find(logicFilter); % used later in drawing boundaries

AreasVector = AreasVector(logicFilter) / pixelsPerMm^2;
figure(1)
hist(AreasVector,30) % make histogram
xlabel('Hole Area [mm^2]')
TotalArea = sum(AreasVector) %find total area (useful for
   permeability)

DiametersVector = 2*sqrt(AreasVector/pi);
figure(2)
hist(DiametersVector, 30)
xlabel('Hole Diameter [mm]')

NumberOfHoles = length(AreasVector)

meanDiameter = mean(DiametersVector)

totalArea = sum(AreasVector)

% now show the image with holes encircled
figure
imagesc(a)%show the original image to draw blob boundaries on
colormap gray

%%%%%get boundaries %%%%
blobboundaries = bwboundaries(b,'noholes'); %gives cell with each
   region's boundary points
M=length(AreasVector);
cellboundaries=cell(M,1); %preallocate cell in memory
hold on

for i=1:M
   cellboundaries{i}=blobboundaries{m(i)};
   thisBoundary = cellboundaries{i};
```

```
   plot(thisBoundary(:,2), thisBoundary(:,1), 'Color', [1 0 0], '
      LineWidth', 1.5);
end
hold off
```

Appendix B

Permeability and Diffusivity Data

B.1 Script to Calculate Effective Pore Size

B.1.1 Main File

```
clear all; clc; close all;
treshold = 185; %set the brightness
pixelsPerMm = 93; %get from photoshop
NumImages = 7;
thickness = 0.58; %mm, of sample core
s = (45 / pixelsPerMm) / thickness;
%490 microns typical of needle fixture.

tic
%first area, then effective area
dataForH200Samples = zeros(NumImages,2);
for q = 1:NumImages

grayImage = rgb2gray(imread( strcat(num2str(q),'.jpg')));

%make it logical black and white
binaryImage = (grayImage > treshold);

blob1 = regionprops(binaryImage,'Area');

%switch data type
AreasCell = struct2cell(blob1)';

%area of each blob
AreasVector = cell2mat(AreasCell(:,1));

% areas greater than 10 square pixels
logicFilter = AreasVector > 10 & AreasVector < 10000;
m = find(logicFilter); % used later in drawing boundaries

AreasVector = AreasVector(logicFilter) / pixelsPerMm^2;
```

```
DiametersVector = 2*sqrt(AreasVector/pi);

% %makes histograms
% figure(1)
% hist(AreasVector,30) % make histogram
% xlabel('Hole Area [mm^2]')
% %find total area (useful for permeability)
% TotalArea = sum(AreasVector)

% figure(2)
% hist(DiametersVector, 30)
% xlabel('Hole Diameter [mm]')

numberOfHoles = length(AreasVector);
meanDiameter = mean(DiametersVector);
a = meanDiameter / thickness;
aEff = aEffective(a,s);
effectiveDiameter = aEff * thickness;

dataForH200Samples(q,1) = sum(AreasVector); % total area

% effective area
dataForH200Samples(q,2) = pi*effectiveDiameter^2 /4*numberOfHoles;

end

csvwrite('dataForH200Samples.txt',dataForH200Samples);
compTime = toc

% % now show the image with holes encircled
% figure
%show the original image to draw blob boundaries on
% imagesc(grayImage)
% colormap gray
%
% %%%%%get boundaries %%%%

% gives cell with each region's boundary points
% blobboundaries = bwboundaries(binaryImage,'noholes');
% M=length(AreasVector);
% cellboundaries=cell(M,1); %preallocate cell in memory
% hold on
%
% for i=1:M
%    cellboundaries{i}=blobboundaries{m(i)};
%    thisBoundary = cellboundaries{i};
%    plot(thisBoundary(:,2), thisBoundary(:,1),...
            'Color', [1 0 0], 'LineWidth', 1.5);
% end
```

```
% hold off
```

B.1.2 Function Files

```
function aEff = aEffective(a,s)
% this function takes (a,s) and returns a_eff for needle

alphaPoints = 10;
c = @(r, alpha) 2*r*acos( (2*r^2 + 2*alpha*r + alpha^2 - a^2)...
             / (2*r*(r + alpha)) );
pcAlpha = zeros(alphaPoints,1); %temp storage for p at each alpha
rAndHoles = rAndCounts(s,a); %generate for each s

for hole = 1:length(rAndHoles(:,1)) % do each hole cluster first
   r = rAndHoles(hole,1);
   if r < a %central hole different
      % do the single hole case
      Alpha = linspace(0, a, alphaPoints);
      for index = 1:alphaPoints
         alpha = Alpha(index); % is like r, but use alpha variable
         pcAlpha(index) = alpha*solveHankel(alpha,a);
      end
      pBar = mean(pcAlpha)*2/a; % just center hole here
   else
      % do the arc length stuff
       %at +-a, c=0 so it's pointless
      Alpha = linspace(-a*.9, a*.9, alphaPoints);
      for index = 1:alphaPoints
         alpha = Alpha(index);
         pcAlpha(index) = c(r, alpha)*solveHankel(r + alpha, a);
      end
      pBar = pBar + rAndHoles(hole,2)*mean(pcAlpha)*2/pi/a;
   end

end

aEff = sqrt(a^2./pBar);



function rAndCounts = rAndCounts(s,a)
%this function makes the new hex pattern for each s
r0=15;%10*(a + s);
tol = a/10; % tolerance for comparing r values
   b = 2*a + s;
   N = ceil(r0/b);
   M = ceil(r0/(2*b*sqrt(3)));
   k = 0; % initialize counter
   r = zeros(N*M,1);
```

```
   for n = -N:N
      for m = -M:M

         k = k + 1;
         x = n*b;
         y = m*b*sqrt(3);
         r(k) = sqrt(x*x + y*y);
%          scatter(x, y, [], [0 0 0])

         k = k + 1;
         x = (n + .5)*b;
         y = (m + .5)*b*sqrt(3);
         r(k) = sqrt(x*x + y*y);
%          scatter(x, y, [], [0 0 0])
      end
   end

   % figure(2)
   % hist(r,100)
   r = r( r < r0 ); % makes the circular pattern

   uniqueR = uniquetol(r, tol, 'DataScale', 1);
   rAndCounts2 = zeros(length(uniqueR), 2);
   rAndCounts2(:, 1) = uniqueR;

   for k = 1:length(uniqueR);
   % counts how many times each uniqueR is found in raw r array
      rAndCounts2(k, 2) = sum(ismembertol(r, ...
                  rAndCounts2(k,1), tol, 'DataScale', 1) );
   end
   rAndCounts = rAndCounts2;
   %bar(rAndCounts(:, 1),rAndCounts(:, 2))



% solves Laplace's eqn with unit pulse flux using Hanekl transform
function u=solveHankel(r,a)
warning('off','MATLAB:quadgk:MaxIntervalCountReached');
%U=@(q) (a/q) * besselJ(1,a*q) * besselJ(0,q*r) * tanh(q);
u=quadgk(@(q) (a./q) .* besselj(1,a.*q) .* besselj(0,q.*r)...
       .* tanh(q),0,Inf,'MaxIntervalCount',5000,...
            'AbsTol',1.e-16,'RelTol',1.e-10);
```

Appendix C

Thermal Conductivity Apparatus

C.1 Thermal Conductivity Apparatus Codes

C.1.1 PID Control Simulation in MATLAB

```
conductance = 0.05 / 0.001; % conductance, k/h
area = (25.4/1000)^2 * pi/4; % area of 1 inch disk
p = 3.7; % power of heater
c0 = 0.0138;
c1 = 0.3522; % from experimental data fit
stepAmplitude = 30; % degrees (C) to heat to
kp = 300/stepAmplitude;
kd = 0;
ki = .07;

s = zpk(0,[],1);
A = 100;
A = kp + kd * s + ki / s; % PID controller (MCU)
%A = kp + kd * (s + .1) / (s + .01);
B = p / 255; % heater power (W) / pwm (bytes)
C = (c0*s + c1) / (p*s);
L = area * conductance; % heat leak through sample (W) / temp
   difference (T)

sys = A*B*C / (1 + A*B*C + C*L);

opt = stepDataOptions;
opt.StepAmplitude = stepAmplitude;
figure(1)
step(sys,opt)
figure(2)
step( (1 - sys) * A, opt)
```

C.1.2 System Simulation

```
area = (25.4/1000)^2 * pi/4; % area of 1 inch disk
```

```
p = 3.7; % power of heater
c0 = 0.0138;
c1 = 0.3522; % from experimental data fit
stepAmplitude = 255; %byte to step at beginning during transient
tFinal = 90; %seconds to allow until k is esimated


s = zpk(0,[],1);
B = p / 255; % heater power (W) / pwm (bytes)
C = (c0*s + c1) / (p*s);

options = stepDataOptions;
options.StepAmplitude = stepAmplitude;

% change conductance range of sample
points = linspace(0, 1000, 30)';

tempFinal = zeros(length(points),1);
for k = 1: length(points)
   conductance = points(k);
   L = area * conductance; % heat leak through sample (W) / temp
      difference (T)
   sys = B*C / (1 + L*C);
   [y,t] = step(sys, tFinal, options);
   tempFinal(k) = y(end);
end

figure(1)
hold off
plot(tempFinal, points,'ko')
polyFit = polyfit(tempFinal, points, 4);
hold on
plot(tempFinal, polyval(polyFit, tempFinal), 'LineWidth', 2)
xlabel('T (K)');
ylabel('Conductance (W/m^2-K)')
```

C.1.3 Microcontroller Code

```
#include <SPI.h>
#include "Adafruit_MAX31855.h"
// define software SPI pins for MAX31855
#define DO 12
#define CS 9
#define CLK 13
#define gatePin 3
#define gatePinGuard 10
// define pins for each thermocouple
#define cC 3
#define cE 4
```

```
#define hC 1
#define hE 2
#define guardT 5
#define KpGuard 10
// misc
#define resistorAnalogPin 3
#define resistance 2.9 // resistor value [Ohm]
#define voltage 11.95 // voltage from regulator
#define averageNum 10 // number of readings to use in moving average

Adafruit_MAX31855 TC(CLK, CS, DO);

// global variables
float TCdata[5] = {};
float hot, cold, T; // hot/cold averaged temps, T = hot - cold (
   delta temp);
float h, Tset; // h is sample thickness in mm
long time0;
byte powerByte; // what is written to analogWrite to control gate
   PWM
float area; // = 0.00050670747; // area of 1 inch disk in [m^2]
const float tol = 1.0 / 30.0; // tolerance for derivative to be
   considered 0
const float power = 3.7; // estimated power [W] delivered to coil at
    duty cycle of 1
float kArray[10] = {}; // C++ initializes it to 0

// function prototypes
//void getTemps();
//void setThermocoupleNumber(int);
//float estimateConductance(float T);
//void controlGuard();
//float Tderivative();
//void printTime();
//void printDeltaT();
//void printPower(float power);
//void printHotCold();

void setup() {
  // TC multiplexer stuff
  pinMode(9,OUTPUT);
  pinMode(4,OUTPUT);
  pinMode(5,OUTPUT);
  pinMode(6,OUTPUT);
  pinMode(7,OUTPUT);
  pinMode(13,OUTPUT);
  digitalWrite(7, HIGH); // multiplexer enable
  digitalWrite(13,LOW);
```

```
analogReference(INTERNAL); // sets 1.1V=1023
Serial.begin(9600);

// MOSFET gates controlling heater coils
pinMode(gatePin, OUTPUT);
pinMode(gatePinGuard, OUTPUT);
digitalWrite(gatePin, 0);
digitalWrite(gatePinGuard, 0);
// set timer 2 divisor to 256 for PWM frequency of 122.55 Hz
TCCR2B = TCCR2B & B11111000 | B00000110;

// gather inputs from user
while (!Serial.available()) {
  delay(20); // wait for user input to start
}
while (Serial.available()) {
   Serial.read(); // flush out any other bites in serial buffer
}
Serial.print("Enter sample thickness (mm)@");
while (!Serial.available()) ; // waits for input
h = Serial.parseFloat() / 1000.f;
Serial.print("@Thickness is: ");
Serial.print(h * 1000.f, 2); Serial.print(" mm@");
delay(1000); // chance to read

Serial.print("Enter sample area (mm^2)@");
while (!Serial.available()) ; // waits for input
area = Serial.parseFloat() / 1000000.f;
Serial.print("@Area is: ");
Serial.print(area * 1000000.f, 2); Serial.print(" mm^2@");
delay(1000); // chance to read

// verify that T = 0 before beginning
getTemps();
while (T > 1.01) {
  getTemps();
  Serial.print("@Temperature difference detected!@");
  printDeltaT(); printHotCold();
  delay(2000);
}

// get the system to approximately steady state.
// first, step power to full 255 for 90 seconds
Serial.print("@Heating for 90 seconds.@");
time0 = millis(); // set time datum
while (millis() - time0 < 90000) { // 90 seconds step
  analogWrite(gatePin, 255);
  delay(50);
  getTemps();
```

```
  printDeltaT(); printTime(); printHotCold();
}
// now get the temp it reached, and estimate conductance
getTemps();
Tset = T;
float conductanceEstimate = estimateConductance(T);
// print out the conductivity estimate:
Serial.print("@Conductivity estimate: ");
Serial.print(conductanceEstimate * h, 4); Serial.print("@");
// estimate the required powerByte for SS at Tset
int powerLevel = 255.0 * conductanceEstimate * area * Tset / power;

// make sure there is no overflow
if (powerLevel > 255) {
  powerByte = 255;
} else {
  powerByte = (byte) powerLevel;
}
printPower(power * powerByte / 255.f);
// now wait to reach Tset. gate duty cycle should still be 1 (255)
while (T < Tset) {
  getTemps();
  printDeltaT(); printTime();
}

analogWrite(gatePin, powerByte);
// compute derivatives to get optimization started
float d1 = Tderivative();
float Q2;
float d0 = d1;
float Q1 = powerByte;
if (abs(d1) < tol) return;
if (d1 > 0) {
  Q2 = 0.8 * Q1;
} else {
  Q2 = 1.2 * Q1;
}
analogWrite(gatePin, Q2);
delay(100); // give it a chance for derivative to settle
float d2 = Tderivative();
// now start loop to converge on zero derivative
Serial.print("@Loop to make derivative zero@");
while (abs(d0) > tol) {
  powerByte = Q1 - d1 * (Q1 - Q2) / (d1 - d2);
  analogWrite(gatePin, powerByte);
  d0 = Tderivative();
  Q1 = Q2;
  Q2 = powerByte;
  d1 = d2;
```

```
    d2 = d0;
    printDeltaT(); printTime(); printHotCold();
  }
  Serial.print("@Running...@");
} // end setup

//*************************************************************
void loop() {
  getTemps();

  // compute the accurate power to coil
  // surround with analogWrite(gatePin, 255) to avoid reading 0.0V
  analogWrite(gatePin, 255);
  float Vresistor = analogRead(resistorAnalogPin) * 1.1f / 1023.f;
  analogWrite(gatePin, powerByte);

  float measuredPower =
       ((powerByte / 255.f) * (voltage - Vresistor) * Vresistor /
          resistance) ;
  printPower(measuredPower);

  // compute conductivity using this measured power
  float conductivity = h * measuredPower / (T * area);

  // send data: conductivity, hot/cold temps, guard temp, time
     elapsed
  printDeltaT();
  Serial.print("@a");
  Serial.print(conductivityMovingAverage(conductivity), 4); Serial.
     print("@");
  printHotCold();
  printTime();
} // end main loop ****************************************

// proportional control to make guard temp match Al (Main) temp
void controlGuard() {
  if (TCdata[guardT - 1] < hot) {
    analogWrite(gatePinGuard, (byte) KpGuard * (hot - TCdata[guardT -
        1]));
  }
}

// updates all temps
void getTemps() {
  // reads TCs and updates TCdata array
  for (int k = 0; k < 5; k++) {
    setThermocoupleNumber(k);
    TC.readCelsius();
    delay(5);
```

```
    TC.readCelsius();
    TCdata[k] = TC.readCelsius();
    // make sure it doesn't return nan for temperature
    while (isnan(TCdata[k])) {
      TCdata[k] = TC.readCelsius();
    }
  }
  // average out the inner/outer thermocouples

  // stores them in hot/cold global variables
  //hot = (TCdata[hC - 1] + 2 * TCdata[hE - 1]) / 3;
  hot = TCdata[hC -1];
  //cold = (TCdata[cC - 1] + 2 * TCdata[cE - 1]) / 3;
  cold = TCdata[cC - 1];
  T = hot - cold;

  // check overheat, wait for cooling, maybe print message:
  while (hot > 80.f) {
    delay(1000);
    Serial.print("@OVERHEATED@");
    analogWrite(gatePin, 0);
    printHotCold();
  }

  // make guard temp mat ch hot temp
  controlGuard();
}

// set multiplexer to read certain TC
void setThermocoupleNumber(int x) {
  if ( x > 7 || x < 0 ) return;
  digitalWrite(6, bitRead(x, 2));
  digitalWrite(5, bitRead(x, 1));
  digitalWrite(4, bitRead(x, 0));
  delay(5);
}

// Estimated conductance: pass in the value of T after 90 seconds at
    255 step
// uses linear fit for system simulation in MATLAB
float estimateConductance(float T) {
  return 0.0056 * pow(T, 4) - 0.5195 * pow(T, 3) +
            18.5135 * pow(T, 2) - 321.4312 * T + 2525.0;
}

float Tderivative() {
  getTemps();
  long t1 = millis()/1000;
  float T1 = T;
```

```
  delay(3000); // to let system settle to new derivative
  getTemps();
  long t2 = millis()/1000;
  float T2 = T;
  return (T2 - T1) / (t2 - t1);
}

void printTime() {
 Serial.print("@e"); Serial.print((millis() - time0) / 1000); Serial.
    print("@");
}

void printDeltaT() {
 Serial.print("@f"); Serial.print(T, 2); Serial.print("@");
}

void printPower(float power) {
  Serial.print("@b"); Serial.print(power, 2); Serial.print("@");
}

void printHotCold() {
  Serial.print("@c"); Serial.print(hot, 2); Serial.print("@");
  Serial.print("@d"); Serial.print(cold, 2); Serial.print("@");
}

float conductivityMovingAverage(float k) {
  static int i = 0;
  if (i > averageNum) i = 0;
  kArray[i] = k;
  i++;
  // now average the entire array
  float sum = 0.f;
  for (int j = 0; j < averageNum; j++) {
    if (kArray[j] < 0.0001f) return k;
    sum += kArray[j];
  }
  return sum / averageNum;
}
```

C.1.4 Android Application

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/
   android"
   xmlns:tools="http://schemas.android.com/tools"
   android:layout_width="match_parent"
   android:layout_height="match_parent"
   android:orientation="vertical"
   tools:context="com.example.android.thermalconductivity.
```

```
   MainActivity">

<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:orientation="horizontal">
   <Button
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="4"
      android:text="Connect"
      android:id="@+id/ConnectButton"
      android:onClick="onClickConnect"/>
   <View
      android:layout_width="0dp"
      android:layout_height="30dp"
      android:layout_weight="1"
      android:layout_gravity="center_vertical"
      android:id="@+id/StatusView"/>
</LinearLayout>

<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content">
   <EditText
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:inputType="numberDecimal"
      android:layout_weight="5"
      android:id="@+id/editBox"/>
   <Button
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="2"
      android:text="Send"
      android:id="@+id/SendButton"
      android:onClick="onClickSend"/>
</LinearLayout>

<TextView
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:text="Serial Messages"
   android:id="@+id/SerialMessages"
   android:textSize="20sp"
   android:padding="10dp"/>

<LinearLayout
   android:layout_width="match_parent"
```

```
   android:layout_height="wrap_content"
   android:padding="5dp">
   <TextView
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:gravity="center"
      android:id="@+id/K"
      android:textStyle="bold"
      android:text="Conductivity"/>
   <TextView
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:gravity="center"
      android:id="@+id/Power"
      android:text="Power"/>
</LinearLayout>

<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:padding="5dp">
   <TextView
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:gravity="center"
      android:id="@+id/Thot"
      android:text="Hot Temp"/>
   <TextView
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
      android:gravity="center"
      android:id="@+id/Tcold"
      android:text="Cold Temp"/>
</LinearLayout>

<LinearLayout
   android:layout_width="match_parent"
   android:layout_height="wrap_content"
   android:paddingBottom="10dp"
   android:padding="5dp">

   <TextView
      android:layout_width="0dp"
      android:layout_height="wrap_content"
      android:layout_weight="1"
```

```
            android:gravity="center"
            android:id="@+id/Time"
            android:text="Time"/>
        <TextView
            android:layout_width="0dp"
            android:layout_height="wrap_content"
            android:layout_weight="1"
            android:gravity="center"
            android:id="@+id/DeltaT"
            android:text="Delta T"/>
    </LinearLayout>

    <com.github.mikephil.charting.charts.LineChart
        android:id="@+id/chart1"
        android:layout_width="match_parent"
        android:layout_height="match_parent"/>

</LinearLayout>
```

```
package com.example.android.thermalconductivity;

import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import android.content.Intent;
import android.content.pm.ActivityInfo;
import android.graphics.Color;
import android.os.Bundle;
import android.os.Handler;
import android.support.v7.app.AppCompatActivity;
import android.view.View;
import android.view.WindowManager;
import android.widget.EditText;
import android.widget.TextView;
import android.widget.Toast;

import com.github.mikephil.charting.charts.LineChart;
import com.github.mikephil.charting.data.Entry;
import com.github.mikephil.charting.data.LineData;
import com.github.mikephil.charting.data.LineDataSet;

import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.util.ArrayList;
import java.util.List;
import java.util.Set;
```

```
import java.util.UUID;

public class MainActivity extends AppCompatActivity {
   // Display fields, thread classes
   Thread streamThread;
   UpdateUI updateUI;

   // chart pointers
   List<Entry> entries;
   LineDataSet dataSet;
   LineData lineData;
   LineChart chart;
   Handler handler;

   // bluetooth pointers/constants
   public static final String DEFAULT_DEVICE_NAME = "HC-06";
   public static final String UUID_STRING =
      "00001101-0000-1000-8000-00805F9B34FB";
   private final int REQUEST_ENABLE_BT = 1; //key for
      onActivityResult() from intent
   BluetoothAdapter btAdapter;
   BluetoothDevice btDevice;
   BluetoothSocket btSocket;
   InputStream inputStream;
   OutputStream outputStream;
   boolean bluetoothStarted;

   // view pointers
   TextView messages;
   TextView conductivity;
   TextView power;
   TextView hotTemp;
   TextView coldTemp;
   TextView deltaT;
   TextView time;
   View statusView;
   EditText editBox;

   @Override
   protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.activity_main);

      // disable screen rotation and screen sleep
      setRequestedOrientation(ActivityInfo.
         SCREEN_ORIENTATION_PORTRAIT);
      getWindow().addFlags(WindowManager.LayoutParams.
         FLAG_KEEP_SCREEN_ON);
```

```
    // constructors
    chart = (LineChart) findViewById(R.id.chart1);
    messages = (TextView) findViewById(R.id.SerialMessages);
    conductivity = (TextView) findViewById(R.id.K);
    power = (TextView) findViewById(R.id.Power);
    hotTemp = (TextView) findViewById(R.id.Thot);
    coldTemp = (TextView) findViewById(R.id.Tcold);
    deltaT = (TextView) findViewById(R.id.DeltaT);
    time = (TextView) findViewById(R.id.Time);
    statusView = findViewById(R.id.StatusView);
    editBox = (EditText) findViewById(R.id.editBox);

    // for handler, need to pass in the UI thread looper
    handler = new Handler(getApplicationContext().getMainLooper())
        ;
    updateUI = new UpdateUI();
    streamThread = new Thread(new BtRunnable());

    // by default, bluetooth is not connected. Show red,
        initialize boolean
    statusView.setBackgroundColor(Color.RED);
    bluetoothStarted = false;
}

public void onClickSend(View v) {
    byte[] bytesOut = editBox.getText().toString().getBytes();
    try {
        outputStream.write(bytesOut);
        editBox.setText(null);
    } catch (IOException e) {
        e.printStackTrace();
    } catch (NullPointerException p) {
        p.printStackTrace();
    }
}

// pre: bluetooth must be connected (gets called by
    startBluetooth(...)
// post: starts stream thread, initializes chart
public void onConnected() {
    messages.setText("Connected!");
    bluetoothStarted = true;
    // update to show it has been connected
    statusView.setBackgroundColor(Color.GREEN);
    // start thread for bluetooth stream listener
    streamThread.start();

    // start chart:
    // List of "entry" objects, each entry contains x and y
```

```
    entries = new ArrayList<Entry>();
    entries.add(new Entry(0, 0));

    // each list of entries is made into LineDataSet object,
       includes label for legend
    // LineDataSet extends DataSet which is chart-type specific
    dataSet = new LineDataSet(entries, "T ");

    // LineDataSet objects can be added to the LineData object (
       more than one)
    // e.g. lineData.addDataSet(); to make multiple plots on same
       chart
    lineData = new LineData(dataSet);

    // takes the LineData objects and plots each LineDataSet on
       the chart View
    chart.setData(lineData);
    chart.invalidate(); // refresh (not sure how it works)

    // send byte to trigger arduino
    byte[] b = {0};
    try {
       outputStream.write(b);
    } catch (IOException e) {
       e.printStackTrace();
    }
}

public void onClickConnect(View v) {
    if (bluetoothStarted){
       return;
    }
    // initialize bluetooth adapters, checks if bluetooth
       supported/enabled
    btAdapter = BluetoothAdapter.getDefaultAdapter();
    if (btAdapter == null) {
       // means BT is not supported
       Toast.makeText(getApplicationContext(),
                "Device doesn't support Bluetooth", Toast.
                   LENGTH_SHORT).show();
    } else if (!btAdapter.isEnabled()) {
       // if it's supported but not enabled, make intent to enable
           it
       Intent enableAdapter = new Intent(BluetoothAdapter.
          ACTION_REQUEST_ENABLE);
       startActivityForResult(enableAdapter, REQUEST_ENABLE_BT);
    } else {
       startBluetooth(DEFAULT_DEVICE_NAME);
    }
```

```
}

@Override
protected void onActivityResult(int requestCode, int resultCode,
   Intent data) {
   super.onActivityResult(requestCode, resultCode, data);
   if (requestCode == REQUEST_ENABLE_BT) {//got here from
      enableAdapter intent
      if (resultCode == RESULT_CANCELED) {
         messages.setText("Bluetooth could not be started :(");
      } else {
         startBluetooth(DEFAULT_DEVICE_NAME);
      }
   }
}

// pre: bluetooth must be enabled to call this method
// post: connects to device whose name is passed in
public void startBluetooth(String deviceName) {
   //check device list for paired devices
   Set<BluetoothDevice> bondedDevices = btAdapter.
      getBondedDevices();
   if (bondedDevices.isEmpty()) {
      Toast.makeText(getApplicationContext(),
               "No paired devices found :(",Toast.LENGTH_SHORT).
                  show();
   } else {
      // go through the set of bonded (paired) devices
      for (BluetoothDevice iterator : bondedDevices) {
         // find one with the correct name, and make it the
            bluetooth device
         if (iterator.getName().equals(deviceName)) {
            btDevice = iterator; // set that device
            Toast.makeText(getApplicationContext(),
                     "Device Found :)",Toast.LENGTH_SHORT).show()
                        ;
            break; // get out of the loop
         }
      }
   }

   // create socket to handle outgoing connection
   // Here a RFCOMM socket is used. RFCOMM--also known as Serial
      Port Profile
   // -- is essentially a Bluetooth protocol to emulate an RS232
      cable.
   try {
      btSocket = btDevice.createRfcommSocketToServiceRecord(
```

```
                UUID.fromString(UUID_STRING));
        btSocket.connect();
        outputStream = btSocket.getOutputStream();
        inputStream = btSocket.getInputStream();
        onConnected();
    } catch (Exception e) {
        messages.setText("Could not set up a stream :(");
        Toast.makeText(getApplicationContext(),
                e.getMessage(),Toast.LENGTH_LONG).show();
        bluetoothStarted = false;
        statusView.setBackgroundColor(Color.RED);
    }
}

// reads bytes from input stream, assuming it is created
public byte[] readBytes() {
    try {
        int bytesAvailable = inputStream.available();
        if (bytesAvailable > 0) {
            byte[] bytesInStream = new byte[bytesAvailable];
            inputStream.read(bytesInStream); // inputs them into
                bytesInStream
            return bytesInStream;
        }
    } catch (IOException e) {
        Toast.makeText(getApplicationContext(),
                e.getMessage(),Toast.LENGTH_LONG).show();
    }
    return "Stream is empty.".getBytes();
}

// this class is run by the bluetooth thread
public class BtRunnable implements Runnable {

    @Override
    public void run() {
        while (true) {
            // check stream, if available wait a few millis, then
                call handler
            try {
                Thread.sleep(50);
                if (inputStream.available() > 5) {
                    Thread.sleep(100);
                    handler.post(updateUI);
                }
            } catch (InterruptedException e1) {
                e1.printStackTrace();
            } catch (IOException e2) {
                e2.printStackTrace();
```

```
            }
        }
    }
}

// class that updates UI. Pass it to handler to run on UI thread.
public class UpdateUI implements Runnable {

    // this will run on the UI thread
    @Override
    public void run() {
        // get the byte array from InputStream and turn into string
        String streamString = (new String(readBytes())).trim();
        streamString = streamString.replaceAll("^[ @]+", "");
        // parse the string with spaces as delimiters
        String[] tokens = streamString.split("[@]+");
        float t =0.f; float T = 0.f;
        for (String token : tokens) {
            // trim leading delimiters to avoid empty string in
               string array
            // get the leading char which dictates what view it goes
                to
            char switchKey = token.charAt(0);
            switch (switchKey) {
                case 'a': conductivity.setText("k: " + token.
                    substring(1) + " W/m-K");
                    break;
                case 'b': power.setText("Power: " + token.substring
                    (1) + " W");
                    break;
                case 'c': hotTemp.setText("Hot: " + token.substring
                    (1) + " \u2103");
                    break;
                case 'd': coldTemp.setText("Cold: " + token.substring
                    (1) + " \u2103");
                    break;
                case 'e': time.setText("Time: " + token.substring(1)
                    + " s");
                    try {
                        t = Float.parseFloat(token.substring(1));
                    } catch (NumberFormatException e) {
                        messages.setText("FLOAT ERROR");
                    }
                    break;
                case 'f': deltaT.setText("\u0394T: " + token.
                    substring(1) + " \u2103");
                    try {
                        T = Float.parseFloat(token.substring(1));
                    } catch (NumberFormatException e) {
```

```
                    messages.setText("FLOAT ERROR");
                 }
                 break;
              default: messages.setText(token); break;
           }
        }
        // also update chart
        if (T > 0.f && t > 0.f) {
           lineData.addEntry(new Entry(t, T), 0);
           chart.notifyDataSetChanged();
           chart.invalidate();
        }
     }
  }
}
```

C.1.5 Areas by Image Processing

```
% need to put a US quarter on top left of image
clear; clc; close all;
quarterArea = (pi/4) * 24.08^2; % mm^2
threshold = 150; % brightness units cutoff
rawImage = rgb2gray(imread('200C.jpg'));
bwLogical = (rawImage > threshold); %make it logical black and white
% get areas and centroid (in px unit)
   % generate a filter to eliminate small areas (dust in photo etc.)
   cc = bwconncomp(bwLogical);
   statsUnfiltered = regionprops(cc, 'Centroid', 'Area');
   filter = find([statsUnfiltered.Area] > 1000);
   filteredBwLogical = ismember(labelmatrix(cc), filter);
stats = regionprops(filteredBwLogical, 'Centroid', 'Area');
unitsFactor = quarterArea / stats(1).Area; % to switch to mm^2 units

for k = 1 : length(stats)
  areas(k) = stats(k).Area * unitsFactor;
  x(k) = stats(k).Centroid(1);
  y(k) = stats(k).Centroid(2);
end

% now show the image with holes encircled
figure(1)
imagesc(rawImage)%show the original image to draw blob boundaries on
colormap gray

%%%%% plot boundaries %%%%
boundaryColor = [1 0 0]; % color of boundary (1 0 0) is red (RGB)
textColor = [0 0 0]/255; % color of text showing area
```

```
% get cell with each region's boundary points:
blobBoundaries = bwboundaries(filteredBwLogical,'noholes');
M = length(blobBoundaries);
cellboundaries = cell(M,1); %preallocate cell in memory
   hold on
   for i=1:M
      cellboundaries{i} = blobBoundaries{i};
      thisBoundary = cellboundaries{i};
      plot(thisBoundary(:,2), thisBoundary(:,1), 'Color',
         boundaryColor, 'LineWidth', 1.5);
      text(x(i), y(i), num2str(areas(i)), 'Color', textColor, '
         FontWeight', 'bold');
   end
   hold off
```

Appendix D

Additional Experiments

D.1 Time Varying Concentration

D.1.1 ODE Solving MATLAB Scripts

Main File

```
clc; clear variables; close all;
matlabpool open %use all available CPU cores
L=1; % half of thickness
n=30;% discretizations in x
h=5;%mass transfer coeficient
x=linspace(0,L,n);
dx=x(5)-x(4);
t1=2;
tsteps=40;
tspan=linspace(0,t1,tsteps);

% construct F matrix
zerovec=zeros(1,n);
onevec=ones(n,1);
F=spdiags([onevec,onevec],[-1,1],n,n)/(2*dx);
F(1,:)=zerovec;
F(end,:)=zerovec;

% construct S matrix
S=spdiags([onevec,-2*onevec,onevec],[-1 0 1],n,n)/dx^2;
S(1,2)=2/dx^2;
S(end,:)=zerovec;

%initial guess
pn0=[1 1 1 1 1 1]*3;

u0=zerovec';%onevec;
tol=.001;
options=odeset('AbsTol',tol,'RelTol',tol);
```

```
% tic
%[T,C1]=ode15s(@(t,u) cartpdeback(t,u,pn0*2),tspan,u0,options);
% toc
% key=C1(21,:);
% save 'target.dat' key -ascii
%target=load('target.dat');

%target=4-cos(x*2*pi/(3*L)); %target function
targetc{1}=.25+sqrt(1.1^2-x.^2);
targetc{2}=targetc{1};
targetc{3}=cos(2*pi*x/6)+.2;
targetc{4}=.3*cos(2*pi*x/1.6)+1;

% starting guess pn0 larger makes the solution faster
% (after less T). Solution for cos() must start near zero.

target=targetc{1};
tic
[coeffs{1},eval1]=fminsearch(@errorfunc,pn0/10);
toc
target=targetc{2};
tic
[coeffs{2},eval2]=fminsearch(@errorfunc,pn0/5);
toc
target=targetc{3};
tic
[coeffs{3},eval3]=fminsearch(@errorfunc,pn0/10);
toc
target=targetc{4};
tic
[coeffs{4},eval4]=fminsearch(@errorfunc,pn0/10);
toc
[eval1 eval2 eval3 eval4]

%plot the solution using coefficients found by fminsearch
for tes=1:4

[~,C]=ode15s(@(t,u) cartpdeback(t,u,coeffs{tes}),tspan,u0);

figure(tes)
plot(x,targetc{tes},'ro')
hold on
for p=1:tsteps;
   plot(x,C(p,:))
   axis([0 1 -1 1.6])
   hold on
   pause(.1)
end
```

```
end
```

Error Function

```
function minerror=errorfunc(pn)

%options=evalin('base',options);
target=evalin('base','target');
tspan=evalin('base','tspan');
u0=evalin('base','u0');
tsteps=evalin('base','tsteps');
[~,C]=ode15s(@(t,u) cartpdeback(t,u,pn),tspan,u0);

error=zeros(tsteps,1);
for ro=1:tsteps
error(ro,1)=norm(C(ro,:)-target);
end
minerror=min(error);
```

ODE RHS Function

```
function dudt=cartpdeback(t,u,pn)

%pn=evalin('base','pn');
c=dot([1 t t^2 t^3 t^4 t^5],pn);
Dif=@(u) 1;
Difu=@(u) 0;
D=Dif(u);
Du=Difu(u);
S=evalin('base','S');
F=evalin('base','F');
dx=evalin('base','dx');
h=evalin('base','h');


d2udx2=S*u;
d2udx2(end)=(1/dx^2)*( (2*dx*h/D(end))*(c-u(end))+u(end-1)...
   -2*u(end) + u(end-1));

dudx=F*u;
dudx(end)=(h/D(end))*(c-u(end)); % clear up D(end)=Dl

dudt=(D.*d2udx2 + Du.*(dudx.^2));
```

D.1.2 Microcontroller Code for Vessel System

```
#include<SD.h>
#include <LiquidCrystal.h>
#include <MAX6675.h>
#include <Wire.h>
```

```
// TEMPERATURE
const int kT=2;
const int Ttol=20; // 10 = 1C
int tempe;
int tempeTarget;
int heatTime;
int Tsense;
int tempe2;
#define heaterpin 42
#define safetypin 43

MAX6675 tc1(34,35,33,1); // MAX6675 (new library)
MAX6675 tc2(31,32,30,1); // SCK, CS, SO - old one
// TC reads 3.25 at freezing, 101.25 at boiling

// PRESSURE
const int Ptol=6; // digital read units to tolerate
int ventTime=100; // milliseconds for solenoid valve open
int inletTime=100; // milliseconds for solenoid valve open
int Psense;
int pressure;
int pressureTarget;
#define ventpin 41
#define inletpin 40
#define pressurepin 1

// OTHER
#define beeppin 3

long int tyme; // what's assigned from millis();
long int tymeTarget; // Target is what's read from SD
long int tymedone; // millis when finished
long int tyme0;
long int tymenow;

LiquidCrystal lcd(27,26,25,24,23,22);
File targetFile;
File recordFile;

String towrite=""; // string created to write to SD CSV
int finish = 0; // check finish condition


void setup()
{
// set relay control pins
pinMode(ventpin,OUTPUT);
digitalWrite(ventpin,LOW);
```

```
pinMode(inletpin,OUTPUT);
digitalWrite(inletpin,LOW);
pinMode(safetypin,OUTPUT);
digitalWrite(safetypin,HIGH);
pinMode(heaterpin,OUTPUT);
digitalWrite(heaterpin,LOW);
pinMode(beeppin,OUTPUT);

// set up LCD
lcd.begin(20,4);
delay(200);
lcd.clear();
delay(200);
lcd.setCursor(0,0);

// set up SD
pinMode(53,OUTPUT); // check if this is correct
if ( !SD.begin(53) ){lcd.print("Error"); while(1){}}
targetFile = SD.open("targets.csv"); //open file for reading
if(targetFile) lcd.print("Success");

// initialize readings
tempe=10*tc1.read_temp();
pressure=analogRead(pressurepin);

// initialize targets
tymeTarget = targetFile.parseInt();
pressureTarget = targetFile.parseInt();
tempeTarget = targetFile.parseInt();

// set time reference
//delay(100);
beep(200);
tyme0=millis();
}

 void loop()
{
 resettargets(); // checks to see if it's on target
 finishfunction(); // checks to see if it reached end
 controlPressure();
 //controlTemperature();

}

void beep(int t) // beeps for t milliseconds
{
analogWrite(beeppin,100);
delay(t);
```

```
analogWrite(beeppin,0);
}

void controlPressure()
   {
    pressure=analogRead(pressurepin); //inlet

    if(pressure<(pressureTarget-Ptol))
    { // pressure low, open inlet
      while(pressure<(pressureTarget-Ptol))
      {
      digitalWrite(inletpin,HIGH);
      pressure=analogRead(pressurepin);
      }
      digitalWrite(inletpin,LOW);
      delay(400);
    }

    if(pressure>(pressureTarget+Ptol)) // vent
    {
      while(pressure>(pressureTarget+Ptol))
      {
      digitalWrite(ventpin,HIGH);
      pressure=analogRead(pressurepin);
      }
      digitalWrite(ventpin,LOW);
      delay(400);
    }
   }

// function that controls Temperature
void controlTemperature()
   {
   // checks overheating condition
    if(tc2.read_temp()>90 || tc1.read_temp()>90)
    {
    digitalWrite(safetypin,LOW);
    digitalWrite(heaterpin,LOW);
    lcd.clear();
    lcd.setCursor(0,6);
    lcd.print("WARNING!");
    lcd.setCursor(1,3);
    lcd.print("RESISTIVE PAD");
    lcd.setCursor(2,5);
    lcd.print("OVERHEATED");
    lcd.setCursor(3,2);
    lcd.print("PAD TEMP:");
    lcd.setCursor(3,11);
    lcd.print(tc2.read_temp() );
```

```
    delay(5000);
    return;
    }

    if(tc2.read_temp()<10 || tc1.read_temp()<10
            || tc2.read_temp()>200 || tc1.read_temp()>200)
    {
     digitalWrite(heaterpin,LOW);
     digitalWrite(safetypin,LOW);
     digitalWrite(ventpin,LOW);
     digitalWrite(inletpin,LOW);
     lcd.clear();
     lcd.print("TC broke");
     while(1);
    }

    tempe=10*tc1.read_temp();
    if(tempe<(tempeTarget-Ttol))
     {
      heatTime=200+kT*(tempeTarget-tempe);
      digitalWrite(heaterpin,HIGH);
      delay(heatTime);
      digitalWrite(heaterpin,LOW);
     }
    digitalWrite(heaterpin,LOW);
   }

// finish function
void finishfunction()
{
 if(finish)
  {
   digitalWrite(safetypin,LOW); // turn everything off
   digitalWrite(heaterpin,LOW); // turn everything off
  tymedone=millis();
  lcd.clear();

  lcd.setCursor(3,0);
  lcd.print("***RUN OVER***");

  lcd.setCursor(0,1);
  lcd.print("T1:");

//  lcd.setCursor(3,1);
//  lcd.print(tc1.read_temp());


  lcd.setCursor(10,1);
  lcd.print("T2:");
```

```
//  lcd.setCursor(13,1);
//  lcd.print(tc2.read_temp());

  lcd.setCursor(0,2);
  lcd.print("P:");

  lcd.setCursor(9,2);
  lcd.print("Cool Time:");

  lcd.setCursor(0,3);
  lcd.print("Hr:");

  lcd.setCursor(9,3);
  lcd.print("Min:");

  int hours; // don't think they need to be long
  int minutes;
  const long int divisorH=3600000;
  const long int divisorM=60000;
  while(finish)
  //keeps printing time since finished (locked in this mode)
  {
   tymenow=millis();
   tyme=tymenow-tymedone;
   hours=tyme/divisorH;
   minutes=tyme/divisorM-60*hours;
   lcd.setCursor(3,3);
   lcd.print(hours);
   lcd.setCursor(13,3);
   lcd.print(minutes);

  lcd.setCursor(3,1);
  lcd.print(tc1.read_temp());

  lcd.setCursor(13,1);
  lcd.print(tc2.read_temp());

  lcd.setCursor(0,3);
  lcd.print(MPaPressure());

   delay(5000);
   beep(200);
  }

 }
}

 // write records to recordFile on SD card
```

```
void recordtoSD()
{
recordFile = SD.open("records.csv",FILE_WRITE);
  tymenow=millis();
  tyme=tymenow-tyme0;
  tempe2=10*tc2.read_temp();
  if(recordFile)
    {
    // records raw sensor data to better compare with target
    towrite=String(tyme);
    towrite+=",";
    towrite+=String(pressure);

    towrite+=",";
    towrite+=String(tempe);
    towrite+=",";
    towrite+=String( tempe2 );
    //towrite+="\n"; // puts extra space
    recordFile.println(towrite);
    }
   recordFile.close();
}

// function that sets targets - call more often than tstep
 void resettargets()
    {
     tymenow=millis();
     tyme=tymenow-tyme0;
     if(tyme>tymeTarget)
     {
      tymeTarget = targetFile.parseInt();
      pressureTarget = targetFile.parseInt();
      tempeTarget = targetFile.parseInt();

      if(tymeTarget==911) // check to see if it's done
        {
        finish=1; // will go to exit mode in the main loop
        targetFile.close();
        beep(1000);
        return;
        }

      recordtoSD(); // write current state to file

      // print current state to LCD:
      lcd.clear();

      lcd.setCursor(3,0);
      lcd.print("***Running***");
```

```
      lcd.setCursor(0,1);
      lcd.print("Set P:");

      lcd.setCursor(6,1);
      lcd.print(0.0054*pressureTarget+0.0834);

      lcd.setCursor(12,1);
      lcd.print(" T:");

      lcd.setCursor(14,1);
      lcd.print(tempeTarget/10);

      lcd.setCursor(0,2);
      lcd.print("T1");

      lcd.setCursor(3,2);
      lcd.print(tc1.read_temp());

      lcd.setCursor(10,2);
      lcd.print("T2:");

      lcd.setCursor(13,2);
      lcd.print(tc2.read_temp());

      lcd.setCursor(0,3);
      lcd.print("Pressure:");

      lcd.setCursor(11,3);
      lcd.print(MPaPressure());

//      lcd.setCursor(15,3);
//      lcd.print("MPa");

      }// end of if statement
    }
```

D.1.3 MATLAB Script for Generating CSV File

```
%linear function to go from MPa to digital y=185.61x-15.422;

tend=1*60; % total run time in seconds
tstep=2; % time step in seconds
t=0:tstep:tend; % time vector in seconds
t=t*1000; % time vector in millis (for Arduino)
p=(1+sin(t)); %pressure in MPa
%pressure in digital readout (from calibration)
p=round( 185.61*p-15.422 );
```

```
T=25+10*sin(t); % temperature in degrees C
% move decimal over (for Arduino to parse int correctly)
T=T*10;
M=[uint32(t)', uint32(p)', uint32(T)'];

csvwrite('targets.csv',M);
```

Appendix E

Code for 3D Printing Apparatus

E.1 3D Printing Bubbles

Here is the source code for the Windows application and the microcontroller.

E.1.1 MainWindow.xaml.cs

```
using System;
using System.Collections.Generic;
using System.IO.Ports;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using System.Windows;
using System.Windows.Controls;
using System.Windows.Data;
using System.Windows.Documents;
using System.Windows.Input;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Navigation;
using System.Windows.Shapes;

namespace CatalystGUI
{
   public partial class MainWindow : Window
   {
      const int stepsPerClick = 1000; // number of stepper steps per
          +/- click
      CameraStuff cameraStuff;
      ArduinoStuff arduinoStuff;

      public MainWindow()
      {
         InitializeComponent();
```

```
   PortSelector.ItemsSource = SerialPort.GetPortNames(); //
      populate combo box

   // gray out camera buttons until camera is started
   LiveButton.IsEnabled = false;
   StopButton.IsEnabled = false;
   CaptureButton.IsEnabled = false;

}


#region Buttons: Exposure, Frame Rate, Count
// when clicked, the value gets multiplied/divided by this
   amount:
private const double MULTIPLIER_HIGH = 1.5;
private const double MULTIPLIER_MED = 1.05;
private const double MULTIPLIER_LOW = 1.005;

private void ExposurePlus(object sender, RoutedEventArgs e)
{
   if (!StartCamButton.IsEnabled) cameraStuff.ExposureTime = (
      uint)(cameraStuff.ExposureTime * MULTIPLIER_HIGH);
}

private void ExposureMinus(object sender, RoutedEventArgs e)
{
   if (!StartCamButton.IsEnabled) cameraStuff.ExposureTime = (
      uint)(cameraStuff.ExposureTime / MULTIPLIER_HIGH);
}

// frame rates

   // ---
private void FrameRateMinus(object sender, RoutedEventArgs e)
{
   if (!StartCamButton.IsEnabled) cameraStuff.FrameRate /=
      MULTIPLIER_LOW;
}

private void FrameRateMinus2(object sender, RoutedEventArgs e)
{
   if (!StartCamButton.IsEnabled) cameraStuff.FrameRate /=
      MULTIPLIER_MED;
}

private void FrameRateMinus3(object sender, RoutedEventArgs e)
{
```

```
   if (!StartCamButton.IsEnabled) cameraStuff.FrameRate /=
      MULTIPLIER_HIGH;
}

   // +++
private void FrameRatePlus(object sender, RoutedEventArgs e)
{
   if (!StartCamButton.IsEnabled) cameraStuff.FrameRate *=
      MULTIPLIER_LOW;
}

private void FrameRatePlus2(object sender, RoutedEventArgs e)
{
   if (!StartCamButton.IsEnabled) cameraStuff.FrameRate *=
      MULTIPLIER_MED;
}

private void FrameRatePlus3(object sender, RoutedEventArgs e)
{
   if (!StartCamButton.IsEnabled) cameraStuff.FrameRate *=
      MULTIPLIER_HIGH;
}

// end frame rates

private void FrameCountMinus(object sender, RoutedEventArgs e)
{
   if (!StartCamButton.IsEnabled) cameraStuff.FrameCount -= 5;
}

private void FrameCountPlus(object sender, RoutedEventArgs e)
{
   if (!StartCamButton.IsEnabled) cameraStuff.FrameCount += 5;

}
#endregion

#region Buttons: Capture, Live, Stop, Start Cam
private void Capture_Click(object sender, RoutedEventArgs e)
{
   //CaptureButton.IsEnabled = false; // don't know how to re-
      enable through binding
   cameraStuff.Capture();
   // debug openCV
   //cameraStuff.GetImage();
}

private void Live_Click(object sender, RoutedEventArgs e)
{
```

```
    if (!cameraStuff.liveMode)
    {
        cameraStuff.Live();
    }
}

private void Stop_Click(object sender, RoutedEventArgs e)
{
    cameraStuff.liveMode = false;
}

// starts camera that way it won't happen right when UI starts
private void StartCam_Click(object sender, RoutedEventArgs e)
{
    if (cameraStuff == null)
    {
        cameraStuff = new CameraStuff(Dispatcher);
    }

    // once it initializes, it turns off Start Cam button
    if (cameraStuff.InitializeCamera())
    {
        CameraGrid.DataContext = cameraStuff; // every child in
            "Grid" gets properties from this object
        StartCamButton.IsEnabled = false;
        LiveButton.IsEnabled = true;
        StopButton.IsEnabled = true;
        CaptureButton.IsEnabled = true;
    }
}


#endregion

#region Arduino Control Buttons
private void ConnectUSB_Click(object sender, RoutedEventArgs e
    )
{
    if (arduinoStuff == null)
    {  // initialization of ArduinoStuff object
        this.arduinoStuff = new ArduinoStuff(Dispatcher);
        ArduinoGrid.DataContext = this.arduinoStuff;
        PressuresItemsControl.ItemsSource = this.arduinoStuff.
            Pressures; // cuz I can't figure out how to bind it
            in XAML

    }

    this.arduinoStuff?.Connect(PortSelector);
```

```
}

// FAN
private void FanMinus_Click(object sender, RoutedEventArgs e)
{
   if (this.arduinoStuff != null && arduinoStuff.Fan <=
      ArduinoStuff.FAN_TRESHOLD)
   {
      arduinoStuff.Fan = 0;
   }
   else
   {
      arduinoStuff.Fan -= 5;
   }
}

private void FanPlus_Click(object sender, RoutedEventArgs e)
{
   if (this.arduinoStuff != null) arduinoStuff.Fan += 5;
}

private void FanStop_Click(object sender, RoutedEventArgs e)
{
   if (this.arduinoStuff != null) this.arduinoStuff.Fan = 0;
}

// NEEDLE
private void NeedlePositionMinus_Click(object sender,
   RoutedEventArgs e)
{
   arduinoStuff?.MoveStepper("NeedlePositionMotor", -1000);
}

private void NeedlePositionPlus_Click(object sender,
   RoutedEventArgs e)
{
   arduinoStuff?.MoveStepper("NeedlePositionMotor", 1000);
}

private void NeedleStop_Click(object sender, RoutedEventArgs e
   )
{
   if (arduinoStuff != null)
   {
      this.arduinoStuff.controlNeedleBool = false;
      this.arduinoStuff.MoveStepper("NeedlePositionMotor", 0);
   }
}
```

```
// ITEMS CONTROL (PRESSURES)
private void PressurePlus_Click(object sender, RoutedEventArgs
    e)
{
   Button button = (Button)sender;
   // get button's data context so I can see which Pressure it
       refers to
   // by default it's "object" but I know it's AnalogValue cuz
       that's the context it inherits from ItemsControl (
      because I set it)
   AnalogValue context = (AnalogValue)button.DataContext;

   // move the appropriate motor to increase pressure
   arduinoStuff?.MoveStepper(context.DisplayName, -
      stepsPerClick); // clockwise (-) increases pressure
}

private void PressureMinus_Click(object sender,
   RoutedEventArgs e)
{  // see Plus method for comments
   Button button = (Button)sender;
   AnalogValue context = (AnalogValue)button.DataContext;
   this.arduinoStuff?.MoveStepper(context.DisplayName,
      stepsPerClick); // ccw(+) decreases pressure
}

// sets that motor to 0 steps - this is just for pressure
   regulator motors
private void RegulatorMotorStop_Click(object sender,
   RoutedEventArgs e)
{
   Button button = (Button)sender;
   AnalogValue context = (AnalogValue)button.DataContext;
   if (this.arduinoStuff != null)
   {
      context.controlPressureBool = false; // stop trying to
         control it
      this.arduinoStuff?.MoveStepper(context.DisplayName, 0);

   }
}

// Solenoids
private void Solenoid_Click(object sender, RoutedEventArgs e)
{
   Button button = (Button)sender;
   AnalogValue context = (AnalogValue)button.DataContext;

   // now toggle the solenoid
```

```
   if (this.arduinoStuff != null)
   {
      if (!this.arduinoStuff.DigitalRead(context.SolenoidPin))
      {  // solenoid is off
         this.arduinoStuff.DigitalWrite(context.SolenoidPin,
            1); // switch it on
         button.Content = "is ON"; // label button as "ON"
      }
      else
      {
         this.arduinoStuff.DigitalWrite(context.SolenoidPin,
            0); // switch it off
         button.Content = "is OFF"; // label button as "ON"
      }
   }
}


// TEMPERATURE
private void Temp1Plus_Click(object sender, RoutedEventArgs e)
{
   if (this.arduinoStuff != null) this.arduinoStuff.
      Temperature1Set += 5; // 5 deg C
}

private void Temp1Minus_Click(object sender, RoutedEventArgs e
   )
{
   if (this.arduinoStuff != null) this.arduinoStuff.
      Temperature1Set -= 5; // 5 deg C
}

private void Temp1Stop_Click(object sender, RoutedEventArgs e)
{
   if (this.arduinoStuff != null) this.arduinoStuff.
      Temperature1Set = 0;
}
#endregion


// COMMAND LINE =)
private void CommandLine_KeyDown(object sender, KeyEventArgs e
   )
{
   // when "enter" is hit
   if (Key.Return == e.Key)
   {
      // get whatever was typed in text box
      string command = CommandLine.Text;
```

```
// clear text box
CommandLine.Clear();

// capture n imagges once frame rate matches bubble drop
    frequency
if (command.ToLower().StartsWith("frame") && cameraStuff
    != null)
{
   string[] tokens = command.Split(' ');
   if (tokens.Length != 2) return;
   int n;
   if (!int.TryParse(tokens[1], out n)) return;
   this.cameraStuff.FrameRate = this.cameraStuff.
      FrameRate / (1.0 + 1.0 / n);
}


// saving images
if (command.ToLower().StartsWith("save") && cameraStuff
   != null)
{
   // make sure a directory name was entered
   string[] tokens = command.Split(' ');
   if (tokens.Length != 2) return;
   // tokens[1] is folder name
   cameraStuff.SaveImages(tokens[1]);

   string path = CameraStuff.DEFAULT_PATH_ROOT + tokens
      [1] + "\\Info.txt";
   System.IO.StreamWriter textFile = new System.IO.
      StreamWriter(path);
   textFile.WriteLine(GetState()); // current value of
      parameters to info file
   textFile.Close();
}


// toggle LED Ring
if (command.ToLower().Equals("led") && this.arduinoStuff
    != null)
{
   this.arduinoStuff.LEDring = !arduinoStuff.LEDring;
}

//// solenoid ON/OFF - no longer used since I made
   buttons
//if (command.ToLower().StartsWith("sol") && this.
   arduinoStuff != null)
//{
```

```
            //   if (command.Contains("on"))
            //   {  // have to explicitly turn it on
            //      this.arduinoStuff.Solenoid = true;
            //   }
            //   else
            //   {  // off by default (for safety)
            //      this.arduinoStuff.Solenoid = false;
            //   }
            //}

         }
      }

      // Gets current value of all parameters (except needle pos.)
      string GetState()
      {
         string s = "";
         s = DateTime.Now.ToString() + "\n";

         // pressure sensors
         foreach (var p in this.arduinoStuff.Pressures)
         {
            var name = p.DisplayName; // sensor name
            var value = p.Value; // pressure in whatever units it
               was written in
            s += name + ": " + value + "\n";
         }

         // fan
         s += "Fan Speed: " + arduinoStuff.Fan + "\n";

         // LED Ring
         s += "LED Ring on: " + arduinoStuff.LEDring + "\n";
         Console.Write(s);
         return s;
      }


   }
}
```

E.1.2 MainWindow.xaml

```
<Window x:Class="CatalystGUI.MainWindow"
      xmlns="http://schemas.microsoft.com/winfx/2006/xaml/
         presentation"
      xmlns:x="http://schemas.microsoft.com/winfx/2006/xaml"
      xmlns:d="http://schemas.microsoft.com/expression/blend/2008"
      xmlns:mc="http://schemas.openxmlformats.org/markup-
```

```
      compatibility/2006"
   xmlns:local="clr-namespace:CatalystGUI"
   mc:Ignorable="d"
   Title="Catalyst Project (Confidential)" Height="780" Width
      ="840">
<Window.Resources>
   <Style TargetType="{x:Type TextBlock}">
      <Setter Property="FontSize" Value="12pt"/>
   </Style>
   <Style TargetType="{x:Type TextBox}">
      <Setter Property="FontSize" Value="12pt"/>
   </Style>
   <Style TargetType="{x:Type Button}">
      <Setter Property="FontSize" Value="12pt"/>
   </Style>
</Window.Resources>


<Grid Name="UIgrid">
   <Grid.ColumnDefinitions>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="Auto"/>
      <ColumnDefinition Width="*"/>
   </Grid.ColumnDefinitions>

   <!--Arduino control grid-->
   <Grid Name="ArduinoGrid" Grid.Column="0">
      <Grid.RowDefinitions>
         <RowDefinition Height="Auto"/>
         <RowDefinition Height="Auto"/>
         <RowDefinition Height="Auto"/>
         <RowDefinition Height="Auto"/>
         <RowDefinition Height="*"/>
      </Grid.RowDefinitions>

      <!--connect and COM port-->
      <StackPanel Grid.Row="0" Orientation="Horizontal">
         <Button Click="ConnectUSB_Click" Content="Connect USB"/>
         <ComboBox x:Name="PortSelector" MinWidth="100"/>
      </StackPanel>

      <!--temperature control-->
      <StackPanel Grid.Row="1" Margin="0,5,0,0" Orientation="
         Horizontal">
         <TextBlock Text="Temp" Width="50"/>
         <TextBlock Text="{Binding Temperature1}" MinWidth="50"/>
         <Button Click="Temp1Minus_Click" Content="-" Width
            ="20"/>
         <TextBox Text="{Binding Temperature1Set}" MinWidth
```

```
      ="40"/>
   <Button Click="Temp1Plus_Click" Content="+" Width="20"/>
   <Button Click="Temp1Stop_Click" Content="Stop" Width
      ="50"/>

</StackPanel>

<!--fan control-->
<StackPanel Grid.Row="2" Margin="0,25,0,0" Orientation="
   Horizontal">
   <TextBlock Text="Fan %" Width="100"/>
   <!--<TextBlock Text="{Binding Fan}" MinWidth="40"/>-->
   <Button Click="FanMinus_Click" Content="-" Width="20"/>
   <TextBox Text="{Binding Fan}" MinWidth="40"/>
   <Button Click="FanPlus_Click" Content="+" Width="20"/>
   <Button Click="FanStop_Click" Content="Stop" Width
      ="50"/>

</StackPanel>

<!--needle control-->
<StackPanel Grid.Row="3" Margin="0,25,0,0" Orientation="
   Horizontal">
   <TextBlock Text="Ndl x" Width="50"/>
   <TextBlock Text="{Binding Potentiometer}" MinWidth
      ="50"/>
   <Button Click="NeedlePositionMinus_Click" Content="-"
      Width="20"/>
   <TextBox Text="{Binding NeedlePositionSet}" MinWidth
      ="40"/>
   <Button Click="NeedlePositionPlus_Click" Content="+"
      Width="20"/>
   <Button Click="NeedleStop_Click" Content="Freeze" Width
      ="50"/>
</StackPanel>

<!--pressures display (auto generated by items control)-->
<ItemsControl x:Name="PressuresItemsControl" Grid.Row="4">

   <ItemsControl.ItemTemplate>
      <DataTemplate>
         <DataTemplate.Resources>
            <!--Makes it inherit style from the outside-->
            <Style TargetType="TextBlock" BasedOn="{
               StaticResource {x:Type TextBlock}}"/>
            <Style TargetType="TextBox" BasedOn="{
               StaticResource {x:Type TextBox}}"/>
         </DataTemplate.Resources>
```

```
            <StackPanel Margin="0,25,0,0" Orientation="
               Horizontal">
               <TextBlock Text="{Binding DisplayName}" Width
                  ="60"/>
               <TextBlock Text="{Binding Value}" MinWidth
                  ="40"/>
               <Button Click="PressureMinus_Click" Content="-"
                   Width="20"/>
               <TextBox Text="{Binding SetPoint}" MinWidth
                  ="40"/>
               <Button Click="PressurePlus_Click" Content="+"
                  Width="20"/>
               <Button Click="RegulatorMotorStop_Click"
                  Content="Freeze" Width="50"/>
               <Button Click="Solenoid_Click" Content="is OFF"
                   Width="50"/>
            </StackPanel>
         </DataTemplate>
      </ItemsControl.ItemTemplate>

   </ItemsControl>



</Grid>
<!--end arduino grid-->

<!--Grid Splitter that enables resizing of each cell-->
<GridSplitter HorizontalAlignment="Right"
       VerticalAlignment="Stretch"
       Grid.Column="1" ResizeBehavior="PreviousAndNext"
       Width="3" Background="#FFBCBCBC"/>
<!--x ############################################## x-->

<!--Main camera grid that holds everything in rows-->
<Grid Name="CameraGrid" Grid.Column="2">
   <Grid.RowDefinitions>
      <RowDefinition Height="Auto" />
      <RowDefinition Height="Auto" />
      <RowDefinition Height="*" />
      <RowDefinition Height="Auto" />
   </Grid.RowDefinitions>

   <!--Holds the row of live,stop,capture,commandline elements
       in columns-->
   <Grid Grid.Row="0">
      <Grid.ColumnDefinitions>
         <ColumnDefinition Width="Auto" />
         <ColumnDefinition Width="1*" />
```

```
      <ColumnDefinition Width="1*" />
      <ColumnDefinition Width="1*" />
      <ColumnDefinition Width="3*" />
   </Grid.ColumnDefinitions>

   <Button Grid.Column="0" Name="StartCamButton" Content="
      Start Cam" Click="StartCam_Click"/>
   <Button Grid.Column="1" Name="LiveButton" Content="Live"
       Click="Live_Click"/>
   <Button Grid.Column="2" Name="StopButton" Content="Stop"
       Click="Stop_Click"/>
   <Button Grid.Column="3" Name="CaptureButton" Content="
      Capture" Click="Capture_Click"/>
   <TextBox Grid.Column="4" Name="CommandLine" KeyDown="
      CommandLine_KeyDown"/>
</Grid>

<!--Each control is a vertical StackPanel of label/controls
   . Controls is [-][textbox][+]-->
<UniformGrid Rows="1" Grid.Row="1">
   <StackPanel Orientation="Vertical">
      <!--label-->
      <TextBlock Text="Exposure (micros)" TextAlignment="
         Center"/>
      <!--exposure controls-->
      <StackPanel Orientation="Horizontal"
         HorizontalAlignment="Center">
         <Button Content="-" Width="20" Click="
            ExposureMinus"/>
         <TextBox Width="70pt" Text="{Binding ExposureTime
            }"/>
         <Button Content="+" Width="20" Click="ExposurePlus
            "/>
      </StackPanel>
   </StackPanel>

   <StackPanel Orientation="Vertical">
      <!--label-->
      <TextBlock Text="Frame Rate (Hz)" TextAlignment="
         Center"/>
      <!--framerate controls-->
      <StackPanel Orientation="Horizontal"
         HorizontalAlignment="Center">
         <Button Content="---" Click="FrameRateMinus3"
            Width="35" Margin="0,0,5,0"/>
         <Button Content="--" Click="FrameRateMinus2" Width
            ="25" Margin="0,0,5,0"/>
         <Button Content="-" Click="FrameRateMinus" Width
            ="20"/>
```

```
            <TextBox Width="70pt" Text="{Binding FrameRate}"/>
            <Button Content="+" Click="FrameRatePlus" Width
               ="20" Margin="0,0,5,0"/>
            <Button Content="++" Click="FrameRatePlus2" Width
               ="25" Margin="0,0,5,0"/>
            <Button Content="+++" Click="FrameRatePlus3" Width
               ="35"/>
         </StackPanel>
      </StackPanel>

      <StackPanel Orientation="Vertical">
         <!--label-->
         <TextBlock Text="Frame Count" TextAlignment="Center
            "/>
         <!--count controls-->
         <StackPanel Orientation="Horizontal"
            HorizontalAlignment="Center">
            <Button Content="-" Width="20" Click="
               FrameCountMinus"/>
            <TextBox Width="70pt" Text="{Binding FrameCount
               }"/>
            <Button Content="+" Width="20" Click="
               FrameCountPlus"/>
         </StackPanel>
      </StackPanel>
   </UniformGrid>

   <!--Big UI Image-->
   <Image Name ="imageBox" Source="{Binding UIimage}" Margin
      ="5" Grid.Row="2"/>

   <!--sam: the "selected item" binds the object selected (
      calls "set") to a property in the context inherited by
      ListBox-->
   <ListBox ItemsSource="{Binding ImageSourceFrames}"
      SelectedItem="{Binding UIimage}" Height="110" Grid.Row
      ="3">
      <!--sam: once inside here, the context gets updated to
         the objects inside the listbox. If you want to bind
         to the object itself use
   the {Binding .}, else use the property name straight up. E.
      g. if listbox has inside a object containing "string txt
      " you can do
   {Binding txt} and it will show the text-->
      <ListBox.ItemTemplate>
         <DataTemplate>
            <Image Source="{Binding .}" MaxHeight="80"/>
         </DataTemplate>
      </ListBox.ItemTemplate>
```

```
            <ItemsControl.ItemsPanel>
               <ItemsPanelTemplate>
                  <StackPanel Orientation="Horizontal" />
               </ItemsPanelTemplate>
            </ItemsControl.ItemsPanel>

         </ListBox>
      </Grid>



   </Grid>
</Window>
```

E.1.3 CameraStuff.cs

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;
using System.ComponentModel;
using System.Threading.Tasks;
using System.Windows.Media;
using System.Windows.Media.Imaging;
using System.Windows.Threading;
using SpinnakerNET;
using SpinnakerNET.GenApi;
using Emgu.CV;
using Emgu.Util;
using Emgu.CV.Structure;
using System.Drawing;
using System.Drawing.Imaging;
using System.IO;

namespace CatalystGUI
{
   // TO DO: implement IDisposable interface, DeInit() camera, clean
       up everything
   internal class CameraStuff : INotifyPropertyChanged
   {
      #region Misc. Class Fields
      public static readonly string DEFAULT_PATH_ROOT = "D:\\"; //
         where stuff will be saved (SD card)
      const int WIDTH = 1280; // image format
      const int HEIGHT = 1024;
      private IManagedCamera currentCam; // the blackfly
      private ManagedSystem spinnakerSystem; // spinnaker class
      private Dispatcher UIDispatcher; // invokes actions to run on
         UI thread
```

```
// this is the pic that gets posted on the UI:
private ImageSource _UIimage; public ImageSource UIimage
{
   get { return _UIimage; }
   set
   {
      _UIimage = value;
      NotifyPropertyChanged("UIimage");
   }
}
# endregion

public CameraStuff(Dispatcher UIDispatcher)
{
   this.UIDispatcher = UIDispatcher;

   // create collecton on UI thread so I won't have any
      problems with scope BS
   UIDispatcher?.BeginInvoke( new Action( () =>
   {
      ImageSourceFrames = new ObservableCollection<ImageSource
         >();
   }));

}

public bool InitializeCamera()
{ // call only once
   spinnakerSystem = new ManagedSystem();
   // get list of cams plugged in:
   {
      List<IManagedCamera> camList = spinnakerSystem.
         GetCameras();
      if (0 == camList.Count)
      {
         System.Windows.MessageBox.Show("No camera connected
            .");
         return false;
      }
      else currentCam = camList[0]; // get the first one
   } // camlist is garbage collected
   currentCam.Init(); // don't know what this does
   return true;
}

// Called when Live button is clicked. Should run as task
public bool liveMode; // live mode or frame capture on UI
public void Live()
{
```

```
   SetAcqusitionMode(AcquisitionMode.Continuous, 0);
   currentCam.BeginAcquisition();
   liveMode = true;

   Task.Run(() =>
   {
      while (liveMode)
      {
         // if don't use "using" the frame freezes
         using (var rawImage = currentCam.GetNextImage())
         {
            // updating UI image has to be done on UI thread.
               Use Dispatcher
            UIDispatcher.Invoke(new Action(() =>
            {
               UIimage = ConvertRawToBitmapSource(rawImage);
            }));
         }
      }

      currentCam.EndAcquisition();
   });
}



// gets called when "Capture" UI button is clicked
private ObservableCollection<ImageSource> _imageSourceFrames;
public ObservableCollection<ImageSource> ImageSourceFrames
{  //"observable collection" automatically notifies UI when
   changed
   get
   {
      return _imageSourceFrames;
   }

   set
   {
      _imageSourceFrames = value;
      NotifyPropertyChanged("ImageSourceFrames");
   }
}
//List<IManagedImage> RawImages; // keeps raws
List<byte[]> rawBytes;
public void Capture()
{
   if (liveMode)
   {  // if it's on Live Mode when "Capture" is clicked
      liveMode = false;
```

```
   System.Threading.Thread.Sleep(100); // give it time to
      end acquisition
}

ImageSourceFrames.Clear(); // wipe out image source
   collection
Task.Run(new Action(() =>
{
   SetAcqusitionMode(AcquisitionMode.Multi, FrameCount);
   currentCam.BeginAcquisition();

   // initialize arrays:
   rawBytes = new List<byte[]>((int)this.FrameCount); //
      preallocate memory
   ulong[] timeStamps = new ulong[FrameCount];

   // get image to camera, get its bytes (ManagedData) and
      TimeStamp
   for (int k = 0; k < FrameCount; k++)
   {
      using (IManagedImage rawImage = currentCam.
         GetNextImage())
      {
         rawBytes.Add(rawImage.ManagedData); // ManagedData
             is byte[] of rawImage
         timeStamps[k] = rawImage.TimeStamp;
      }
   }
   currentCam.EndAcquisition(); // done with camera

   // loop again to add burn in timestamps using Emgu and
      post on UI
   var t_0 = timeStamps[0] / 1000; // timestamp is in
      nanoseconds
   Point timeStampPoint = new Point(10, 80); // bottom left
       corner of timestamp text

   for (int k = 0; k < rawBytes.Count; k++)
   {
      // "Image" type is EmguCV image (matrix or something)
      Image<Gray, Byte> cvImage = new Image<Gray, byte>(
         WIDTH, HEIGHT);
      cvImage.Bytes = rawBytes[k];
      // now add the text on the cvImage
      CvInvoke.PutText(
                  cvImage,
                  (timeStamps[k] / 1000 - t_0) + " us",
                  timeStampPoint, // bottom-left corner of
                     first letter
```

```
                        Emgu.CV.CvEnum.FontFace.HersheySimplex,
                        3, // font scale
                        new Bgr(255, 0, 0).MCvScalar, // font
                           color specified using this werid thing
                        2 // thickness of lines used to draw text
                        );

            // post on UI
            UIDispatcher.Invoke(() =>
            { // needs to be done with Dispatcher or else it
               doesn't get a chance to update UI cuz this task
               hogs the thread
               ImageSourceFrames.Add(ConvertBytesToBitmapSource(
                  cvImage.Bytes, WIDTH, HEIGHT));
               if (k == 0) UIimage = ImageSourceFrames[0]; // put
                    first one on screen
            });
         }

         rawBytes.Clear(); // free up some memory

         // use parallel loop
         //Parallel.ForEach(RawImages, (rawImg) =>
         //{

         //});

      }));
   }

   // saves images on SD card from the ImageSourceFrames
      collection
   public void SaveImages(string directoryName, int quality)
   {
      try
      {
         string folderPath = DEFAULT_PATH_ROOT + directoryName;
         // creates new folder if it doesn't exist
         System.IO.Directory.CreateDirectory(folderPath);
         for (int k = 0; k < ImageSourceFrames.Count; k++)
         {
            string path = folderPath + "\\" + k + ".jpg";
            BitmapSource image = (BitmapSource)ImageSourceFrames[
               k];
            SaveJPEG(path, quality, image);
         }
      }
      catch (Exception e)
      {
```

```
            System.Windows.MessageBox.Show("Exception thrown in
               SaveImages() method in CameraStuff. Message: " + e.
               Message);
        }

    }
    // default quality of 60
    public void SaveImages(string directoryName)
    {
        SaveImages(directoryName, 60);
    }

    private void SaveJPEG(string path, int quality, BitmapSource
        bmp)
    {
        JpegBitmapEncoder encoder = new JpegBitmapEncoder();
        BitmapFrame outputFrame = BitmapFrame.Create(bmp);
        encoder.Frames.Add(outputFrame);
        encoder.QualityLevel = quality; // in percent I think (so 0
            - 100)

        using (FileStream file = new FileStream(path, FileMode.
            Create)) //FileMode.Create will overwrite
        {
            encoder.Save(file);
        }
    }

    public void GetImage()
    {
        IManagedImage rawImage = null;
        SetAcqusitionMode(AcquisitionMode.Single, 0); // maybe
            allow client to call this method
        currentCam.BeginAcquisition(); // need to start this every
            time
        rawImage = currentCam.GetNextImage().Convert(
            PixelFormatEnums.Mono8);

        // "Image" type is EmguCV image (matrix or something)
        Image <Gray, Byte> cvImage = new Image<Gray, byte>((int)
            rawImage.Width, (int)rawImage.Height);

        cvImage.Bytes = rawImage.ManagedData; // ManagedData is
            byte[] of rawImage

        Point pt1 = new Point(300, 300);
        Point pt2 = new Point(800, 800);
        LineSegment2D line = new LineSegment2D(pt1, pt2);
        cvImage.Draw(line, new Gray(1), 3); // changes bytes in
```

```
      cvImage
   //cvImage.Save( file path );



   //UIimage = ConvertBytesToBitmapSource(cvImage.Bytes ,
      rawImage.Height);

   // now back to UI thread no?
   //rawImage.Save("C:/afterTask.bmp");
   //ImageSource thing = new ImageSource(convertedImage);
   currentCam.EndAcquisition();
}

#region Acquisition Mode
public enum AcquisitionMode {Single, Multi, Continuous }; //
   made this for kicks
public void SetAcqusitionMode(AcquisitionMode mode, uint
   numFrames)
{
   // get the handle on the properties (called "nodes")
   INodeMap nodeMap = this.currentCam.GetNodeMap();
   // Retrieve enumeration node from nodemap
   IEnum iAcquisitionMode = nodeMap.GetNode<IEnum>("
      AcquisitionMode");

   switch (mode)
   {
      case AcquisitionMode.Continuous:
      {
         // Retrieve entry node from enumeration node
         IEnumEntry iAqContinuous = iAcquisitionMode.
            GetEntryByName("Continuous");
         // Set symbolic from entry node as new value for
            enumeration node (no idea wtf this is necessary)
         iAcquisitionMode.Value = iAqContinuous.Symbolic;
         break;
      }
      case AcquisitionMode.Single:
      {
         IEnumEntry iAqSingle = iAcquisitionMode.
            GetEntryByName("SingleFrame");
         iAcquisitionMode.Value = iAqSingle.Symbolic;
         break;
      }
      case AcquisitionMode.Multi:
      {
         IEnumEntry iAqMultiFrame = iAcquisitionMode.
            GetEntryByName("MultiFrame");
```

```
            iAcquisitionMode.Value = iAqMultiFrame.Symbolic;
            // set burst
            IInteger frameCount = nodeMap.GetNode<IInteger>("
               AcquisitionFrameCount");
            frameCount.Value = numFrames;
            break;
         }

      }
   }
#endregion

#region Exposure Time
public uint ExposureTime
{
   get
   {
      if (currentCam != null) return (uint)currentCam.
         ExposureTime.Value;
      return 0;
   }
   set
   {
      SetExposure(value); // when loses focus, sets camera to
         what user inputed
      NotifyPropertyChanged("ExposureTime"); // update UI box
         (calls get)
   }
}
public void SetExposure(uint micros)
{
   try
   {
      currentCam.ExposureAuto.Value = ExposureAutoEnums.Off.
         ToString();
      // set exposure, make sure it's within limits
      if (micros > currentCam.ExposureTime.Max) // ~30 sec for
          blackfly
      {
         currentCam.ExposureTime.Value = currentCam.
            ExposureTime.Max;
      }
      else if (micros < currentCam.ExposureTime.Min) // ~6
         micros for blackfly
      {
         currentCam.ExposureTime.Value = currentCam.
            ExposureTime.Min;
      }
      else
```

```
        {
            currentCam.ExposureTime.Value = micros;
        }
    }
    catch (Exception e)
    {
        System.Windows.MessageBox.Show("Exception thrown in
            SetExposure(uint micros) method. Exception message: "
             + e.Message);
    }
}
#endregion

#region Frame Rate
public double FrameRate
{
    get
    {
        if (currentCam != null)
            //return (uint)(1000 * currentCam.
               AcquisitionFrameRate.Value) / 1000.0;
            return currentCam.AcquisitionFrameRate.Value;
            return 0.0;
    }
    set
    {
        SetFramerate(value); // when loses focus, sets camera to
             what user inputed
        NotifyPropertyChanged("FrameRate"); // update UI box (
            calls get)
    }
}
public void SetFramerate(double Hz)
{
    try
    {
        // frame rate enable property (otherwise won't let you
            change it)
        IBool iAqFrameRateEnable = currentCam.GetNodeMap().
            GetNode<IBool>("AcquisitionFrameRateEnable");
        iAqFrameRateEnable.Value = true;

        // set frame rate, make sure it's within limits
        if (Hz > currentCam.AcquisitionFrameRate.Max) // ~170
            fps for blackfly
        {
            currentCam.AcquisitionFrameRate.Value = currentCam.
               AcquisitionFrameRate.Max;
        }
```

```
      else if (Hz < currentCam.AcquisitionFrameRate.Min)
      {
         currentCam.AcquisitionFrameRate.Value = currentCam.
            AcquisitionFrameRate.Min;
      }
      else
      {
         currentCam.AcquisitionFrameRate.Value = Hz;

      }
   } catch (Exception e)
   {
      System.Windows.MessageBox.Show("Exception throw in
         SetFramerate(double Hz) method. Exception message: "
         + e.Message);

   }
}
#endregion

#region Frame Count
private uint frameCount;
public uint FrameCount
{
   get
   {
      return frameCount < 2 ? 2 : frameCount; // can't be <2
         for MultiFrame mode
   }
   set
   {
      frameCount = value;
      NotifyPropertyChanged("FrameCount");
   }
}
#endregion

BitmapSource ConvertBytesToBitmapSource(byte[] imageBytes, int
    width, int height)
{
   System.Windows.Media.PixelFormat format = System.Windows.
      Media.PixelFormats.Gray8;

   return BitmapSource.Create(
            (int)width,
            (int)height,
            96d,
            96d,
            format,
```

```
                null,
                imageBytes,
                ((int)width * format.BitsPerPixel + 7) / 8
                );

        }

        // convert raw image (IManagedImage) to BitmapSource
        private BitmapSource ConvertRawToBitmapSource(IManagedImage
            rawImage)
        {
            // convert and copy raw bytes into compatible image type
            using (IManagedImage convertedImage = rawImage.Convert(
                PixelFormatEnums.Mono8))
            {
                byte[] bytes = convertedImage.ManagedData;

                System.Windows.Media.PixelFormat format = System.Windows
                    .Media.PixelFormats.Gray8;

                return BitmapSource.Create(
                      (int)rawImage.Width,
                      (int)rawImage.Height,
                      96d,
                      96d,
                      format,
                      null,
                      bytes,
                      ((int)rawImage.Width * format.BitsPerPixel + 7) /
                          8
                      );
            }
        }

        #region PropertyChanged stuff
        // call this method invokes event to update UI elements which
            use Binding
        public event PropertyChangedEventHandler PropertyChanged;
        protected void NotifyPropertyChanged(string name)
        {
            PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(
                name));
        }
        #endregion

    }
}
```

E.1.4 ArduinoStuff.cs

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.IO.Ports;
using System.Threading.Tasks;
using System.Windows.Controls;
using System.Windows.Threading;

namespace CatalystGUI
{
   // what I have:
   // Pressure - analog (read), motor (set)
   // Needle position - data from image not arduino (maybe count
      steps? but then need to keep track offline)
   // Solenoid - digital (on/off)
   // Fan - digital (on/off), serial (sets PWM duty cycle)
   // LED ring - serial (PWM duty cycle)
   // Future:
   // heater - serial (in), serial (PWM duty cycle)

   internal class ArduinoStuff : INotifyPropertyChanged
   {
      #region Pin Constants
      public const int SOLENOID_PIN = 4;
      public const int FAN_PIN = 5;
      public const int LED_PIN = 6;
      #endregion

      #region Misc fields
      const int MAX_TEMP = 270; // limit on how much you can set
         temp
      public const int BAUD_RATE = 115200;
      public const int FAST_TIMER_TIMESPAN = 125; // 8 Hz
      public const int SLOW_TIMER_TIMESPAN = 511; // 2 Hz
      public const int FAN_TRESHOLD = 50; // below this it's all the
          same for some reason

      SerialPort usb;
      Dispatcher UIDispatcher;
      DispatcherTimer slowTimer; // for requesting solenoid, fan,
         LED ring, moving motors
      DispatcherTimer fastTimer; // for requesting pressure readings
      Task serialTask; // handles incoming data from usb on separate
          thread
      int[] analogValues;
      int[] digitalValues;
      public Dictionary<String, int> motorNameMap; // maps motor
         name (i.e. NeedleMotor, MainPressureMotor) to its motor #
         in arduino code
```

```
#endregion

#region Properties for Control/UI
public bool SIunits { get; set; } // true = SI (kPa, microns),
    false = standard (psi, thou)

int _temperature1;
public int Temperature1
{
   get
   {
      return _temperature1;
   }
   set
   {
      _temperature1 = value;
      NotifyPropertyChanged("Temperature1");
   }
}

int _temperature1Set;
public int Temperature1Set
{
   get
   {
      return _temperature1Set;
   }
   set
   {
      if (value > MAX_TEMP)
      {
         _temperature1Set = MAX_TEMP;
      }
      else if (value < 0)
      {
         _temperature1Set = 0;
      }
      else
      {
         _temperature1Set = value;
      }
      NotifyPropertyChanged("Temperature1Set");
   }
}

int _potentiometer;
public int Potentiometer // actually returns needle position
   in microns
{
```

```
   get
   {
      return PotentiometerToMicrons(_potentiometer);
   }
   set
   {
      _potentiometer = value;
      NotifyPropertyChanged("Potentiometer");
   }
}

public bool controlNeedleBool; // makes it stop controlling by
    "Stop" button on UI
int _needlePositionSet;
public int NeedlePositionSet
{
   get
   {
      return _needlePositionSet;
   }
   set
   {
      _needlePositionSet = value;
      controlNeedleBool = true;
   }
} // set only by UI

public bool LEDring // true = on
{
   get
   {
      return digitalValues[LED_PIN] != 0;
   }
   set
   {
      DigitalWrite(LED_PIN, value ? 1 : 0);
      //this.serialOutgoingQueue.Enqueue(String.Format("%W
         {0},{1};", LED_PIN, value ? 1 : 0));
   }
}

//public bool Solenoid // true = current through solenoid (
   valve open)
//{
//   get
//   {
//      return digitalValues[SOLENOID_PIN] != 0;
//   }
//   set
```

```
//   {  // true does digitalWrite(SOLENOID_PIN, HIGH);

//      this.serialOutgoingQueue.Enqueue(String.Format("%W
   {0},{1};", SOLENOID_PIN, value ? 1 : 0));
//   }
//}

int _fan;

public int Fan
{
   get
   {
      return _fan;
   }
   set
   {
      if (value == 0)
      {  // just switch the MOSFET off
         _fan = 0;
         DigitalWrite(FAN_PIN, 0);
         //this.serialOutgoingQueue.Enqueue(String.Format("%W
            {0},{1};", FAN_PIN, _fan));
         goto Notify;
      }

      if (digitalValues[FAN_PIN] == 0)
      {  // getting here means value != 0, but fan shows as
         OFF
         // switch on the fan MOSFET
         DigitalWrite(FAN_PIN, 1);
         //this.serialOutgoingQueue.Enqueue(String.Format("%W
            {0},{1};", FAN_PIN, 1));
      }


      if (value >= FAN_TRESHOLD && value < 100)
      {
         _fan = value;
      }
      else if (value >= 100)
      {  // max out at 100 independent of value
         _fan = 100;
      }
      else
      {  // anything 1 to FAN_TRESHOLD sets fan to
         FAN_TRESHOLD. To set to zero, have to enter "0"
         _fan = FAN_TRESHOLD;
      }
```

```
        this.serialOutgoingQueue.Enqueue(String.Format("%F{0};",
            _fan));

        Notify: NotifyPropertyChanged("Fan");
    }
}

// ItemsControl collection for pressures:
public List<AnalogValue> Pressures { get; set; }

#endregion

// Constructor
public ArduinoStuff(Dispatcher UIDispatcher)
{
    this.UIDispatcher = UIDispatcher;

    // names for pressure display
    string mainPressure = "Main p";
    string liquidPressure = "Liq p";
    string needlePressure = "Ndl p";

    // dictionary - names of motors
    motorNameMap = new Dictionary<string, int>(); // <name,
        motor # in .ino code>
    motorNameMap.Add("NeedlePositionMotor", 0);
    motorNameMap.Add(liquidPressure, 1);
    motorNameMap.Add(needlePressure, 2);
    motorNameMap.Add(mainPressure, 3);

    // make AnalogValue objects that map to pressure
    Pressures = new List<AnalogValue>();
    Pressures.Add(new AnalogValue(mainPressure, 0, 8));
    Pressures.Add(new AnalogValue(liquidPressure, 1, 9));
    Pressures.Add(new AnalogValue(needlePressure, 2, 10));
    NotifyPropertyChanged("Pressures"); // is this necessary?

    // arrays
    //_pressures = new List<AnalogValue>(); // list of objects
        that have DisplayName, Pin, Value. For UI binding
    analogValues = new int[16]; // stores 10-bit numbers as
        they come in from Arduino (MEGA has 16 pins)
    digitalValues = new int[16]; // stores 0 or 1 as they come
        in from Arduino (only monitoring 0-13 (PWM pins))
    serialIncomingQueue = new Queue<string>(); // initialize
    serialOutgoingQueue = new Queue<string>(); // initialize

    #region Tasks and Timers
    // create task to obtain tokens from serial, add to queue,
```

```
      then process queue
   serialTask = new Task(new Action(() =>
   {
      while (true)
      {
         GetSerialTokens();
         ProcessIncomingQueue();
      }
   }));

   this.slowTimer = new DispatcherTimer();
   this.slowTimer.Interval = TimeSpan.FromMilliseconds(
      SLOW_TIMER_TIMESPAN);
   this.slowTimer.Tick += SlowLoop_Tick;

   this.fastTimer = new DispatcherTimer();
   this.fastTimer.Interval = TimeSpan.FromMilliseconds(
      FAST_TIMER_TIMESPAN);
   this.fastTimer.Tick += FastLoop_Tick;
   // all these start on "Connect()"
   #endregion

}

#region Controls
// move stepper motor by name in motorMap
public void MoveStepper(string motorName, int steps)
{
   if (motorNameMap.TryGetValue(motorName, out int motor))
   {  // if motor name exists, add to outgoing queue
      serialOutgoingQueue.Enqueue(String.Format("%M{0},{1};",
         motor, steps));
   }
}

// Digital Pins
public void DigitalWrite(int pin, int value)
{
   if (value != 0) value = 1; // to avoid bugs
   this.serialOutgoingQueue.Enqueue(String.Format("%W
      {0},{1};", pin, value));

}

// reads from array, so it's not necessarily up-to-date. Have
   to request updates in timer method
public bool DigitalRead(int pin)
{
   return this.digitalValues[pin] == 1; // true means HIGH pin
```

```
}

void ControlTemperature(int tempSet, int tempActual) //if
   controlling more than 1, this needs to b redone
{
   if (tempSet != 0)
   {
      int delta = tempSet - tempActual;

      if (delta > 0)
      {  // too cold
         // when using mechanical relay, DON'T do PWM, use
            bang-bang control
         int signal = 255; // since it goes to analogWrite(-)
         //int signal = delta > 5 ? 255 : 255 * delta / 6; //
            slow it down when it gets within 5 deg C
         this.usb.Write(String.Format("%H0,{0};", signal));
      }
      else
      {  // too hot -> off
         this.usb.Write(String.Format("%H0,{0};", 0));
      }
   }
}

void ControlNeedlePosition()
{
   if (controlNeedleBool)
   {
      int error = NeedlePositionSet - Potentiometer;
      const int steps = 10; // arbitrary, make it enough to
         last through a Fast_Loop cycle
      const int tolerance = 20; // +/- microns to call it good
          enough

      if (error > tolerance) // needs to increase
      {
         MoveStepper("NeedlePositionMotor", steps);
      }
      else if (error < -tolerance) // needs to decrease
      {
         MoveStepper("NeedlePositionMotor", -steps);
      }
      else
      {
         MoveStepper("NeedlePositionMotor", 0);
         controlNeedleBool = false;
      }
   }
```

```
}

// take in the motor name from button -> object.DisplayName
void ControlPressureRegulator(AnalogValue obj)
{
   if (obj.controlPressureBool)
   {
      float error = obj.SetPoint - obj.Value;
      const int steps = 10;
      const float tolerance = 0.05f; // psi tolerance

      if (error > tolerance) // needs to increase
      {
         MoveStepper(obj.DisplayName, -steps); // -steps
            increases pressure
      }
      else if (error < -tolerance) // needs to decrease
      {
         MoveStepper(obj.DisplayName, steps);
      }
      else
      {
         MoveStepper(obj.DisplayName, 0);
         obj.controlPressureBool = false;
      }
   }

}

#endregion

#region Timer Stuff - Serial Outgoing
Queue<string> serialOutgoingQueue;

// updates pressures, processes outgoing queue: moves steppers
   , sets fan, solenoid, LED ring
private void FastLoop_Tick(object sender, EventArgs e)
{
   // take care of outgoing queue
   while (serialOutgoingQueue.Count > 0)
   {
      string token = this.serialOutgoingQueue.Dequeue();
      this.usb.Write(token);
      Console.WriteLine(token);
   }

   // calls for ADS1115 readings
   for (uint ch = 0; ch < 4; ch++)
   {
```

```
            this.usb.Write(String.Format("%C{0};", ch));
        }

        //if (this.usb.BytesToWrite > 128) return; // to prevent
            buffer overflow (64 bytes RX buffer on MEGA)

        // temperature control
        this.usb.Write(String.Format("%T{0};", 1)); // calls for T1
             reading
        ControlTemperature(Temperature1Set, Temperature1);

        // needle control
        ControlNeedlePosition();

        // pressure regulator control
        foreach (AnalogValue p in Pressures)
        {
            ControlPressureRegulator(p);
        }



}

// updates state of: fan, solenoids, LED ring.
private void SlowLoop_Tick(object sender, EventArgs e)
{
    this.usb.Write(String.Format("%R{0};", LED_PIN));
    this.usb.Write(String.Format("%R{0};", FAN_PIN));

    // check solenoids
    foreach (var p in Pressures)
    {
        this.usb.Write(String.Format("%R{0};", p.SolenoidPin));
    }

}
#endregion

#region Serial Incoming
string tokenBuffer; // adds chars until a complete token is
    made (till it sees ";")
Queue<string> serialIncomingQueue; // once a token is made it
    gets added to queue

// enqueues tokens coming in from serial
void GetSerialTokens()
{
    while (null != this.incomingSerialBuffer && "" != this.
```

```
      incomingSerialBuffer)
   {
      if (incomingSerialBuffer[0] == ';')
      {  // buffer is a complete command, add to queue
         this.serialIncomingQueue.Enqueue(this.tokenBuffer);
         //Console.WriteLine(this.tokenBuffer); // for
            debugging
         this.tokenBuffer = String.Empty; // clear the buffer
            for next token
         this.incomingSerialBuffer = this.incomingSerialBuffer
            .Remove(0, 1); // throw away the ";"
      }
      else
      {  // move one char from incomingSerialBuffer to
         tokenBuffer
         this.tokenBuffer += this.incomingSerialBuffer[0];
         this.incomingSerialBuffer = this.incomingSerialBuffer
            .Remove(0, 1);
      }

   }
}

// processes tokens from queue - basically updates fields/UI
   with data that came from Arduino
private void ProcessIncomingQueue()
{
   while (this.serialIncomingQueue.Count != 0)
   {
      string[] elements = this.serialIncomingQueue.Dequeue().
         Split(',');

      try // gotta try because sometimes I get index out of
         range exception (no idea why)
      {
         // elements[0] possibilities are: A (analog pin
            reading), D (digital pin reading)
         switch ((elements[0])[0]) // last [0] is to switch to
             char like charAt(0)
         {
            case 'A': // analog reads FROM ARDUINO'S BUILT IN
               ADC
               {
                  // should split into [A], [pin], [value]
                  int pin; int value; // placeholders

                  if (int.TryParse(elements[1], out pin)
                     && int.TryParse(elements[2], out value))
                  {
```

```
            // update the analogValues array with
               this new data
            this.analogValues[pin] = value;

            //// if pressure sensor is attached to
               this pin, update Pressures collection
               elements
            //foreach (var p in this.Pressures)
            //{ // inefficient but I don't know how
               to map "name" to Property.
            //   if (p.Pin == pin)
            //   {
            //      p.Value = ConvertRawToPressure(
               value);
            //   }
            //}
         }
         break;
      }

   case 'D': // digital reads
      {
         // should split into [D], [pin], [value
            (0/1)]
         int pin; int value; // placeholders

         if (int.TryParse(elements[1], out pin)
            && int.TryParse(elements[2], out value))
         {
            // update digitalValues array that holds
               latest data
            this.digitalValues[pin] = value;

         }

         break;
      }

   case 'C': // from ADS1115
      {
         // should split into [C], [channel/pin], [
            value]
         // arduino guarantees that 0 <= channel/pin
            <= 3
         int pin; int value; // placeholders

         if (int.TryParse(elements[1], out pin)
            && int.TryParse(elements[2], out value))
         {
```

```
                // channel/pin 3 is the potentiometer
                if (pin == 3)
                {
                   Potentiometer = value;
                }
                else // it's a pressure, figure out which
                    one and update the object's Value
                   property
                {
                   foreach (var p in Pressures)
                   {
                      if (p.Pin == pin)
                      {
                         p.Value = ConvertRawToPressure(
                            value);
                      }
                   }
                }
             }

             break;
          }

       case 'T':
          {
             // T,1,25; means TC#1 is at 25 Celsius
             int thermocoupleNumber; int tempCelsius; //
                placeholders

             if (int.TryParse(elements[1], out
                thermocoupleNumber)
                && int.TryParse(elements[2], out
                   tempCelsius))
             {
                Temperature1 = tempCelsius;
             }

             break;
          }

       default:
          // was some garbage identifier, token is
             dequeued so don't worry
          break;
    }
}
catch (IndexOutOfRangeException ex)
{
   System.Windows.MessageBox.Show("
```

```
            IndexOutOfRangeException in switch case: " + ex.
            Message + "\n \"elements[]\" length: " + elements.
            Length);
      }
   }
}

// USB receiving bytes (event handler)
private string incomingSerialBuffer; // holds incoming bytes
   as string
// event handler for USB data received. All it does is read
   all available bytes and puts them in string buffer
private void USB_DataReceived(object sender,
   SerialDataReceivedEventArgs e)
{
   this.incomingSerialBuffer += usb.ReadExisting();
}

#endregion

// makes the "usb" object
// PortSelector is dropdown thing from UI that selects COM
public void Connect(ComboBox PortSelector)
{
   try
   {
      if (this.usb == null)
      {
         if (SerialPort.GetPortNames().Length == 1)
         {  // if there's only 1 COM available, pick that one
            automatically
            usb = new SerialPort(SerialPort.GetPortNames()[0],
                BAUD_RATE);
            PortSelector.SelectedIndex = 0; // automatically
               show what's selected (first one)
         }
         else if (PortSelector.SelectedValue == null)
         {
            PortSelector.ItemsSource = SerialPort.GetPortNames
               (); // in case USB was plugged in after
            System.Windows.MessageBox.Show("No COM port
               selected.");
            return;
         }
         else
         { // use the COM selected from combo box
            usb = new SerialPort(PortSelector.SelectedValue.
               ToString(), BAUD_RATE);
         }
```

```
            // subscribe handler to DataReceived event (gets
               raised kind of randomly after it receives byte in
               serial pipe)
            usb.DataReceived += USB_DataReceived;
            usb.Open();

            serialTask.Start(); // starts task that processes
               incoming serial data
            slowTimer.Start(); // start the timers that send
               requests to Arduino
            fastTimer.Start();

         }
      }
      catch (Exception e)
      {
         System.Windows.MessageBox.Show("Exception thrown in
            ArduinoStuff class' public void Connect(ComboBox
            PortSelector) method. Exeption message: " + e.Message
            );
      }

   }

   // takes raw data from ADS1115 and gives pressure (PSI or kPa)
   float ConvertRawToPressure(int raw)
   {
      float bitstAt5V = 26550; // count at 5.0 volts
      // the honeywell sensors have range .1(bitstAt5V) to .9(
         bitstAt5V) which map to 0-30 psi
      float psi = 30.0f * (raw - bitstAt5V / 10) / (bitstAt5V * 8
          / 10);

      if (this.SIunits)
      { // 1 psi = 6.89476 kPa
         return psi * 6.89476f;
      }
      return psi;
   }

   // turn potentiometer reading into needle distance (microns)
   int PotentiometerToMicrons(int analogCounts)
   {
      return (int)(0.2430 * analogCounts - 4508.9);
   }

   #region PropertyChanged stuff
   public event PropertyChangedEventHandler PropertyChanged;
```

```
        protected void NotifyPropertyChanged(string name)
        {
            // since most things in this class are done on background
                thread, always use UI Dispatcher
            UIDispatcher?.BeginInvoke(new Action(() =>
            {
                PropertyChanged?.Invoke(this, new
                    PropertyChangedEventArgs(name));

            }));

        }
        #endregion
    }
}
```

E.1.5 AnalogValue.cs

```
using System.ComponentModel;

namespace CatalystGUI
{
    internal class AnalogValue : INotifyPropertyChanged
    {
        public string DisplayName { get; set; } // e.g. "Ndl P"

        public int Pin { get; set; } // pin or "channel" on ADS1115
        public int SolenoidPin { get; private set; } // digital pin
            that controls solenoid MOSFET gate

        float[] valuesArray; // pressure in whatever units added in
            array to take moving average
        int i;
        const int ARRAY_SIZE = 6; // # of readings do moving average
        public float Value
        {
            get
            {
                float sum = 0;
                for (int k = 0; k < ARRAY_SIZE; k++)
                {
                    sum += valuesArray[k];
                }
                return (int)(100 * sum / ARRAY_SIZE) / 100f; // make it
                    show only 2 decimal places
            }
            set
            {
                valuesArray[i] = value;
```

```
            i++;
            if (i >= ARRAY_SIZE) i = 0; // reset index to not go out
               -of-bounds

            // need to notify it here and it will update the
               appropriate element in the ItemsControl
            NotifyPropertyChanged("Value");
        }
    }
    public bool controlPressureBool;
    float _setPoint;
    public float SetPoint
    {
        get
        {
            return _setPoint;
        }
        set
        {
            _setPoint = value;
            controlPressureBool = true;
        }
    }

    // constructor
    public AnalogValue(string DisplayName, int Pin, int
        SolenoidPin)
    {
        this.DisplayName = DisplayName;
        this.Pin = Pin;
        this.SolenoidPin = SolenoidPin;

        valuesArray = new float[ARRAY_SIZE];
        i = 0;

        // initialize all to -1
        for (uint n = 0; n < ARRAY_SIZE; n++)
        {
            valuesArray[n] = -1;
        }
    }


    public event PropertyChangedEventHandler PropertyChanged;
    protected void NotifyPropertyChanged(string name)
    {
        PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(
            name));
    }
```

```
    }
}
```

E.2 Microcontroller Code

E.2.1 Main

```
#include "ADS1115.h"
#include <Wire.h>
#include <SPI.h>
#include "MAX31855.h"

// Thermocouple Reader
// on MEGA use:
// pin 50 for DO (MISO - data from slave to master)
// 52 for CLK (SCK - serial clock)
#define CS 48 // chip select pin (don't use 51, won't work)
MAX31855 tc = MAX31855(CS); // thermocouple object ref
ADS1115 ads = ADS1115(250); // ADC object: possible sample rates:
    860 475 250 128

// Stepper Stuff
#define numOfMotors 4
const int motorPin[numOfMotors][4] =
  { {22, 23, 24, 25}, {28, 29, 30, 31}, {34, 35, 36, 37}, {40, 41,
     42, 43} }; // row is motor, column is pin
int stepperState[numOfMotors] = {0}; // state machine for stepper
    motors
int steps[numOfMotors] = {0}; // steps for steppers to move: 0 means
     doesn't have to move
int requestedSteps[numOfMotors]; // requests to update "steps",
    updated by Serial
bool newRequest[numOfMotors] = {false}; // keeps track if request
    was fulfilled
long lastStepTime[numOfMotors] = {0}; // to space out the steps
#define stepPause 5 // ms to wait between steps (basically sets
    stepper speed)

// Serial stuff
int parseState = 0;
char commandType = 0; // the letter to say which function to call
char buf[15];
int idx = 0; // buffer index
int valuesArrayIndex = 0;
int valuesArray[5] = {}; // numbers to be passed to functions (e.g.
    sensor #, fan percent, temperature, pressure)
int16_t ADCdata[4] = {}; // data from ADS1115, signed 16 bit
char serialOutBuffer[20]; // stores outgoing tokens from sprintf
```

```
#define heaterMOSFET 7
#define mainSolenoidPin 8
#define liqSolenoidPin 9
#define ndlSolenoidPin 10
#define fanOnOffPin 5
#define LEDringPin 6

void setup()
{
 pinMode(heaterMOSFET, OUTPUT);
 pinMode(liqSolenoidPin, OUTPUT);
 pinMode(ndlSolenoidPin, OUTPUT);
 pinMode(mainSolenoidPin, OUTPUT);

 pinMode(LEDringPin, OUTPUT);
 pinMode(fanOnOffPin, OUTPUT);
 pinMode(CS, OUTPUT);

 for (int motor = 0; motor < numOfMotors; motor++)
 {
  pinMode(motorPin[motor][0], OUTPUT);
  pinMode(motorPin[motor][1], OUTPUT);
  pinMode(motorPin[motor][2], OUTPUT);
  pinMode(motorPin[motor][3], OUTPUT);
 }
 // set registers for timer3
 // pins 2, 3, 5 use timer3, see https://arduino-info.wikispaces.com
    /Timers-Arduino
 TCCR3A = B10100010;
 TCCR3B = B00010001; // 001 (no prescale)
 ICR3 = 320; // TOP = 320 for 25 kHZ
 OCR3B = 32; // pin 2 goes by OCR3B. Start fan low.
 pinMode(2, OUTPUT);
 // frequency is 16,000,000 / prescale / ICRx / 2
 // duty cycle will be OCRBxN / ICRx, look on datasheet or
    experiment

 Serial.begin(115200); // max baud rate
 ads.begin();

 while(!Serial.available()); // wait for UI to start up

}

void loop()
{
UpdateCommands();
```

```
// moves steppers
for (int m = 0; m < numOfMotors; m++)
{
 stepper(m);
}

// safe heater
if (tc.readCelsius() > 270) analogWrite(heaterMOSFET, 0);

// update ADC readings
ads.updateAll(ADCdata); // updates the array, non-blocking

}

// reads Serial
void UpdateCommands()
{
 int currentByte = Serial.read();
 if (-1 == currentByte)
 {
  return;
 }
 char currentChar = (char) currentByte;
 switch (parseState)
 {
  case 0:   // idle
   if ('%' == currentChar)
   {
    parseState = 1;
    idx = 0;
    valuesArrayIndex = 0;
   }
  break;

  case 1: // saw a "%" char, so looks for command
   // list here all allowable commantType chars:
   if ('A' == currentChar || 'M' == currentChar || 'F' ==
      currentChar || 'H' == currentChar
      || 'W' == currentChar || 'R' == currentChar || 'T' ==
         currentChar || 'C' == currentChar)
   {
    commandType = currentChar;
    parseState = 2;
   }
   else
   { // means got some garbage after the "%", reset
    parseState = 0;
   }
   break;
```

```
   case 2: // param
     if (';' == currentChar)
     {
      buf[idx] = '\0'; // makes a null terminated string so atoi()
         stops there
      valuesArray[valuesArrayIndex] = atoi(buf);
      idx = 0; // got the first integer
      parseState = 0;
      doTasks(commandType, valuesArray);
     }
     else if (',' == currentChar)
     {
      buf[idx] = '\0';
      valuesArray[valuesArrayIndex] = atoi(buf); // capture the
         number, store in valuesArray
      valuesArrayIndex++; // moves over since it wrote a number
      idx = 0; // resets index buffer to ready it for the next
         number
     }
     else
     {
      buf[idx] = currentChar; // builds up the number until "," or
         ";"
      idx++;
     }
     //break; // no need because it's last
  } // end switch
}

// perform the tasks requested by serial
void doTasks(char command, int values[])
{
 switch (command)
 {
  case 'A': // reads analog pin (e.g. %A0;)
  {// has only one value (value[0]) which is the pin to read
    //Serial.print( "A," + (String)values[0] + "," + analogRead(
       values[0]) + ";"); // this hogs/fragments memory apparently
    int result = sprintf(serialOutBuffer, "A,%d,%d;", values[0],
       analogRead(values[0]));
    if (result > 0)
    {
     Serial.print(serialOutBuffer);
    }
    break;
  }

  case 'F': // sets fan speed percent (e.g. %F90; sets fan to 90%)
```

```
  if (values[0] > 100) values[0] = 100;
  if (values[0] < 0) values[0] = 0;
  OCR3B = values[0] * ICR3 / 100; // pin 2 on MEGA
  break;

case 'M': // %M(a),(b); requests motor (a) to move # of steps (b) (
    e.g. %M2,-100; makes motor 2 go 100 steps clockwise)
  requestedSteps[values[0]] = values[1];
  newRequest[values[0]] = true;
  break;

case 'W':
  // gives direct control to digital pins output (write). Careful
     to be a writeable pin (pinMode)
  // %W,8,1; means pin8 HIGH, %W,8,0; is pin8 LOW
  if (values[1] == 0)
  {
   digitalWrite(values[0], LOW);
  }
  else if (values[1] == 1)
  {
   digitalWrite(values[0], HIGH);
  }
  break;

case 'R': // reads digital pin. %R7; reads state of digital pin7,
    returns 1 or 0 for HIGH/LOW
  //Serial.print( "D," + (String)values[0] + "," + (String)
     digitalRead(values[0]) + ";");
{
  int result = sprintf(serialOutBuffer, "D,%d,%d;", values[0],
     digitalRead(values[0]));
  if (result > 0)
  {
   Serial.print(serialOutBuffer);
  }
  break;
}
case 'T': // T0; returns internal temp, %T1; returns first
    thermocouple
{
  int temperature;
  if (values[0] == 0)
  { // give internal temp
   //Serial.print("T," + (String)values[0] + "," + (String)((int)tc
      .readInternal()) + ";");
   temperature = (int)tc.readInternal();
  }
  else if (values[0] == 1)
```

```
  {
   //Serial.print("T," + (String)values[0] + "," + (String)tc.
      readCelsius() + ";");
   temperature = tc.readCelsius();
  }
  else
  {
   break;
  }
  int result = sprintf(serialOutBuffer, "T,%d,%d;", values[0],
     temperature);
  if (result > 0)
  {
   Serial.print(serialOutBuffer);
  }
  break;
}


case 'C': // ADS converter readings from global array: %C1; returns
    ADCdata[1]
{
  int ch = values[0];
  if (ch > 3 || ch < 0) break;
  //Serial.print("C," + (String)ch + "," + (String)ADCdata[ch] +
     ";");
  int result = sprintf(serialOutBuffer, "C,%d,%d;", ch, ADCdata[ch
     ]);
  if (result > 0)
  {
   Serial.print(serialOutBuffer);
  }
  break;
}

case 'H': // actuates the heaters (MOSFET PWM) %H0,235; heater 0,
   at 235/255 duty cycle
  if (values[1] > 255 || values[1] < 0)
  {
   values[1] = 0; // any bad signal - switch off the heater for
      safety.
  }
  analogWrite(heaterMOSFET, values[1]); // if I get more heaters,
     turn heaterMOSFET into array
  break;

default: break;
}
```

```
}

// STEPPER MOTOR
// moves "motor" by requestedSteps
void stepper(int motor) // requestedSteps is updated by Serial.
{
 if (millis() - lastStepTime[motor] < stepPause)
 { //Serial.println( (String)motor + " too early");
  return; // too early to step again
 }

 if (newRequest[motor] && 0 == stepperState[motor])
 {
    // so won't change direction before it finishes all 8 sequences
    steps[motor] = requestedSteps[motor];
    newRequest[motor] = false;
    stepperState[motor] = 1;
 }

 if (0 == steps[motor] && 0 == stepperState[motor])
 {
  return;
 }
 else if (steps[motor] < 0) // clockwise
 {
  switch (stepperState[motor])
  {
   case 0:
    digitalWrite(motorPin[motor][3], HIGH);
    digitalWrite(motorPin[motor][2], LOW);
    digitalWrite(motorPin[motor][1], LOW);
    digitalWrite(motorPin[motor][0], LOW);
    break;
   case 1:
    digitalWrite(motorPin[motor][3], HIGH);
    digitalWrite(motorPin[motor][2], HIGH);
    digitalWrite(motorPin[motor][1], LOW);
    digitalWrite(motorPin[motor][0], LOW);
    break;
   case 2:
    digitalWrite(motorPin[motor][3], LOW);
    digitalWrite(motorPin[motor][2], HIGH);
    digitalWrite(motorPin[motor][1], LOW);
    digitalWrite(motorPin[motor][0], LOW);
    break;
   case 3:
    digitalWrite(motorPin[motor][3], LOW);
    digitalWrite(motorPin[motor][2], HIGH);
    digitalWrite(motorPin[motor][1], HIGH);
```

```
    digitalWrite(motorPin[motor][0], LOW);
    break;
   case 4 :
    digitalWrite(motorPin[motor][3], LOW);
    digitalWrite(motorPin[motor][2], LOW);
    digitalWrite(motorPin[motor][1], HIGH);
    digitalWrite(motorPin[motor][0], LOW);
    break;
   case 5:
    digitalWrite(motorPin[motor][3], LOW);
    digitalWrite(motorPin[motor][2], LOW);
    digitalWrite(motorPin[motor][1], HIGH);
    digitalWrite(motorPin[motor][0], HIGH);
    break;
   case 6:
    digitalWrite(motorPin[motor][3], LOW);
    digitalWrite(motorPin[motor][2], LOW);
    digitalWrite(motorPin[motor][1], LOW);
    digitalWrite(motorPin[motor][0], HIGH);
    break;
   case 7:
    digitalWrite(motorPin[motor][3], HIGH);
    digitalWrite(motorPin[motor][2], LOW);
    digitalWrite(motorPin[motor][1], LOW);
    digitalWrite(motorPin[motor][0], HIGH);
    break;
   default:
    digitalWrite(motorPin[motor][0], LOW);
    digitalWrite(motorPin[motor][1], LOW);
    digitalWrite(motorPin[motor][2], LOW);
    digitalWrite(motorPin[motor][3], LOW);
    steps[motor]++; // signal that one CW step was completed
    stepperState[motor] = -1; // that way it's 0 after the ++
    break;
  }
}
else
{ // counterclockwise
  switch (stepperState[motor])
  {
   case 0 :
    digitalWrite(motorPin[motor][0], HIGH);
    digitalWrite(motorPin[motor][1], LOW);
    digitalWrite(motorPin[motor][2], LOW);
    digitalWrite(motorPin[motor][3], LOW);
    break;
   case 1 :
    digitalWrite(motorPin[motor][0], HIGH);
    digitalWrite(motorPin[motor][1], HIGH);
```

```
    digitalWrite(motorPin[motor][2], LOW);
    digitalWrite(motorPin[motor][3], LOW);
    break;
  case 2 :
    digitalWrite(motorPin[motor][0], LOW);
    digitalWrite(motorPin[motor][1], HIGH);
    digitalWrite(motorPin[motor][2], LOW);
    digitalWrite(motorPin[motor][3], LOW);
    break;
  case 3 :
    digitalWrite(motorPin[motor][0], LOW);
    digitalWrite(motorPin[motor][1], HIGH);
    digitalWrite(motorPin[motor][2], HIGH);
    digitalWrite(motorPin[motor][3], LOW);
    break;
  case 4 :
    digitalWrite(motorPin[motor][0], LOW);
    digitalWrite(motorPin[motor][1], LOW);
    digitalWrite(motorPin[motor][2], HIGH);
    digitalWrite(motorPin[motor][3], LOW);
    break;
  case 5 :
    digitalWrite(motorPin[motor][0], LOW);
    digitalWrite(motorPin[motor][1], LOW);
    digitalWrite(motorPin[motor][2], HIGH);
    digitalWrite(motorPin[motor][3], HIGH);
    break;
  case 6 :
    digitalWrite(motorPin[motor][0], LOW);
    digitalWrite(motorPin[motor][1], LOW);
    digitalWrite(motorPin[motor][2], LOW);
    digitalWrite(motorPin[motor][3], HIGH);
    break;
  case 7 :
    digitalWrite(motorPin[motor][0], HIGH);
    digitalWrite(motorPin[motor][1], LOW);
    digitalWrite(motorPin[motor][2], LOW);
    digitalWrite(motorPin[motor][3], HIGH);
    break;
  default :
    digitalWrite(motorPin[motor][0], LOW);
    digitalWrite(motorPin[motor][1], LOW);
    digitalWrite(motorPin[motor][2], LOW);
    digitalWrite(motorPin[motor][3], LOW);
    stepperState[motor] = -1; // that way it's 0 after the ++
    steps[motor]--; // signal that one CCW step was completed
    break;
  }
}
```

```
  stepperState[motor]++;
  lastStepTime[motor] = millis();
  //Serial.println( (String)motor + " moved");
}
```

E.2.2 ADS1115.h

```
#include "Arduino.h"
#include <Wire.h>

#define ADS1115_ADDRESS (0X48)

#define ADS1015_REG_CONFIG_PGA_6_144V (0x0000) // +/-6.144V range =
   Gain 2/3
#define ADS1015_REG_CONFIG_PGA_4_096V (0x0200) // +/-4.096V range =
   Gain 1
#define ADS1015_REG_CONFIG_PGA_2_048V (0x0400) // +/-2.048V range =
   Gain 2 (default)
#define ADS1015_REG_CONFIG_PGA_1_024V (0x0600) // +/-1.024V range =
   Gain 4
#define ADS1015_REG_CONFIG_PGA_0_512V (0x0800) // +/-0.512V range =
   Gain 8
#define ADS1015_REG_CONFIG_PGA_0_256V (0x0A00) // +/-0.256V range =
   Gain 16

// configs
#define ADS1015_REG_CONFIG_CQUE_NONE (0x0003) // Disable the
   comparator and put ALERT/RDY in high state (default)
#define ADS1015_REG_CONFIG_CLAT_NONLAT (0x0000) // Non-latching
   comparator (default)
#define ADS1015_REG_CONFIG_CPOL_ACTVLOW (0x0000) // ALERT/RDY pin is
    low when active (default)
#define ADS1015_REG_CONFIG_CMODE_TRAD (0x0000) // Traditional
   comparator with hysteresis (default)
#define ADS1015_REG_CONFIG_DR_128SPS (0x0080) // this is 128sps
   default 1000 0000 bits 7:5
#define ADS1015_REG_CONFIG_DR_250SPS (0x00A0) // this is 250 sps
   1010 0000
#define ADS1015_REG_CONFIG_DR_475SPS (0x00C0) // this is 450 sps
   1100 0000
#define ADS1015_REG_CONFIG_DR_860SPS (0x00E0) // this is 860sps (max
   ) 1110 0000

#define ADS1015_REG_CONFIG_MODE_SINGLE (0x0100) // Power-down single
   -shot mode (default)

#define ADS1015_REG_CONFIG_OS_SINGLE (0x8000) // Write: Set to start
    a single-conversion
```

```
typedef enum
{
      GAIN_TWOTHIRDS = ADS1015_REG_CONFIG_PGA_6_144V,
      GAIN_ONE = ADS1015_REG_CONFIG_PGA_4_096V,
      GAIN_TWO = ADS1015_REG_CONFIG_PGA_2_048V,
      GAIN_FOUR = ADS1015_REG_CONFIG_PGA_1_024V,
      GAIN_EIGHT = ADS1015_REG_CONFIG_PGA_0_512V,
      GAIN_SIXTEEN = ADS1015_REG_CONFIG_PGA_0_256V
} adsGain_t;

class ADS1115
{
protected:
      // instance specific properties
      uint8_t _i2cAddress;
      uint16_t _configDefaults; // base values for the ADS register
      uint16_t _config; // adjustable (for gain)
      uint8_t _channel = 0; // runs 0-3
      void writeRegister(uint8_t reg, uint16_t value);
      uint16_t readRegister(uint8_t reg);
      long tStart;
      int msDelay;

public:
      ADS1115(int sampleRate, uint8_t i2cAddress = ADS1115_ADDRESS )
         ;
      void begin();
      void setGain(adsGain_t gain);
      void updateAll(int ADCvalues[4]);
      int16_t readChannel(uint8_t channel);
};
```

E.2.3 ADS1115.cpp

```
#include "Arduino.h"
#include <Wire.h>
#include "ADS1115.h"

// constructor
ADS1115::ADS1115(int sampleRate, uint8_t i2cAddress)
{
      _i2cAddress = i2cAddress;

      // I'll use the defaults
      _configDefaults = ADS1015_REG_CONFIG_CQUE_NONE | // Disable
         the comparator (default val)
                                    ADS1015_REG_CONFIG_CLAT_NONLAT |
                                        // Non-latching (default val
```

```
                                        )
                                    ADS1015_REG_CONFIG_CPOL_ACTVLOW
                                        | // Alert/Rdy active low (
                                        default val)
                                    ADS1015_REG_CONFIG_CMODE_TRAD |
                                        // Traditional comparator (
                                        default val)
                                    ADS1015_REG_CONFIG_MODE_SINGLE
                                        | // Single-shot mode (
                                        default)
                                    ADS1015_REG_CONFIG_OS_SINGLE; //
                                         signal to start conversion
      //_configDefaults = 0x8583; // page 18 of datasheet **CAREFUL
         ** - default is +/- 2.048V

      switch (sampleRate)
      {
            case (860):
                  _configDefaults |= ADS1015_REG_CONFIG_DR_860SPS;
                     // data rate (samples per second)
                  msDelay = 3;
                  break;
            case (475):
                  _configDefaults |= ADS1015_REG_CONFIG_DR_475SPS;
                  msDelay = 4;
                  break;
            case (250):
                  _configDefaults |= ADS1015_REG_CONFIG_DR_250SPS;
                  msDelay = 6;
                  break;
            case(128):
                  _configDefaults |= ADS1015_REG_CONFIG_DR_128SPS;
                  msDelay = 9;
                  break;
            default:
                  _configDefaults |= ADS1015_REG_CONFIG_DR_475SPS;
                  msDelay = 4;
                  break;
      }

      setGain(GAIN_TWOTHIRDS); // makes 2/3 the default

      // start at channel 0
      _channel = 0;
      _config |= 0x4000; // Single-ended AIN0
}

// starts I2C comm, also
void ADS1115::begin() {
```

```
      Wire.begin();
      writeRegister(0x01, _config); // take the first reading on
         channel 0.
      tStart = millis();
}

// Writes 16 bits to specified destination register
void ADS1115::writeRegister(uint8_t reg, uint16_t value)
{
      Wire.beginTransmission(_i2cAddress);
      Wire.write((uint8_t)reg);
      Wire.write((uint8_t)(value >> 8)); // can only do 1 byte at
         the time so has to split it up
      Wire.write((uint8_t)(value & 0xFF));
      Wire.endTransmission();
}

// reads from conversion register (contains data from analog reading
   )
uint16_t ADS1115::readRegister(uint8_t reg)
{
      Wire.beginTransmission(_i2cAddress);
      Wire.write(reg);
      Wire.endTransmission();
      Wire.requestFrom(_i2cAddress, (uint8_t)2); // requests 2 bytes
      return ((Wire.read() << 8) | Wire.read());
}

// set ADS gain (default is 2/3 +/- 6V)
void ADS1115::setGain(adsGain_t gain)
{
      //0b 1111 0001 1111 1111 = 0xF1FF because bits 9-11 control
         amplifier
      _configDefaults &= 0xF1FF;
      _configDefaults |= gain;
}

// slow way to read 1 channel
int16_t ADS1115::readChannel(uint8_t channel)
{
      if (channel > 3) return 0;

      switch (channel)
      {
            case (0):
                  _config = _configDefaults | 0x4000; // Single-
                     ended AIN0
                  break;
            case (1):
```

```
                  _config = _configDefaults | 0x5000; // Single-
                     ended AIN1
                  break;
            case (2):
                  _config = _configDefaults | 0x6000; // Single-
                     ended AIN2
                  break;
            case (3):
                  _config = _configDefaults | 0x7000; // Single-
                     ended AIN3
                  break;
      }

      // Write config register to the ADC
      writeRegister(0x01, _config); // config register is 0x01 -
         page 18 in datasheet

      // Wait for the conversion to complete
      delay(10);

      return readRegister(0); // conversion register is 0x00 - page
         18 of datasheet
}

void ADS1115::updateAll(int ADCvalues[4])
{
      if (millis() - tStart < msDelay) return; // ms to wait

      ADCvalues[_channel++] = (int)readRegister(0); // saves data in
          array, increment channel

      if (_channel > 3) _channel = 0; // loop it

      // prepare the next acquisition
      switch (_channel)
      {
            case (0):
                  _config = _configDefaults | 0x4000; // Single-
                     ended AIN0
                  break;
            case (1):
                  _config = _configDefaults | 0x5000; // Single-
                     ended AIN1
                  break;
            case (2):
                  _config = _configDefaults | 0x6000; // Single-
                     ended AIN2
                  break;
            case (3):
```

```
                    _config = _configDefaults | 0x7000; // Single-
                       ended AIN3
                    break;
        }

        writeRegister(0x01, _config); // start next acquisition
        tStart = millis(); // reset conversion timer

}
```

E.2.4 MAX31855.h

```
#include "Arduino.h"

class MAX31855 {
        public:
              // 2 constructors
        MAX31855(int8_t SCLK, int8_t CS, int8_t MISO);
        MAX31855(int8_t CS);

        double readInternal(void);
        int16_t readCelsius(void);
        int readFarenheit(void);
        uint8_t readError();

        private:
        int8_t sclk, miso, cs, hSPI;
        uint32_t spiread32(void);
        uint32_t hspiread32(void);
};
```

E.2.5 MAX31855.cpp

```
#include "MAX31855.h"
#include <util/delay.h>
#include <stdlib.h>
#include <SPI.h>

// constructor for software SPI
MAX31855::MAX31855(int8_t SCLK, int8_t CS, int8_t MISO)
{
        sclk = SCLK;
        cs = CS;
        miso = MISO;
        hSPI = 0;

        //define pin modes
        pinMode(cs, OUTPUT);
        pinMode(sclk, OUTPUT);
        pinMode(miso, INPUT);
```

```
      digitalWrite(cs, HIGH);
}

// hardware SPI constructor
MAX31855::MAX31855(int8_t CS)
{
  cs = CS;
  hSPI = 1;

  //define pin modes
  pinMode(cs, OUTPUT);

  //start and configure hardware SPI
  SPI.begin();
  SPI.setBitOrder(MSBFIRST);
  SPI.setDataMode(SPI_MODE0);
  SPI.setClockDivider(SPI_CLOCK_DIV4);

  digitalWrite(cs, HIGH);
}

double MAX31855::readInternal(void)
{
      uint32_t v;

      v = spiread32();

      // ignore bottom 4 bits - they're just thermocouple data
      v >>= 4;

      // pull the bottom 11 bits off
      float internal = v & 0x7FF;
      // check sign bit!
      if (v & 0x800)
      {
      // Convert to negative value by extending sign and casting to
         signed type.
      int16_t tmp = 0xF800 | (v & 0x7FF);
      internal = tmp;
      }
      internal *= 0.0625; // LSB = 0.0625 degrees

      return internal;
}

int16_t MAX31855::readCelsius(void)
{
      int32_t bits = spiread32();
```

```
//    Serial.print("0x"); Serial.println(v, HEX);

      if (bits & 0x7) return 999; // error thrown

      // extract bits 30 to 20 (which is a 11-bit signed int, bit 31
          is sign)
      int16_t celsius = 0; // make sure it's all zeros to begin with
      celsius = bits >> 20; // gets rid of bits 0 - 19. Now 20 is
         bit 0

      // check for sign on bit 11 (old bit 31):
      if (bitRead(celsius, 11))
      {     // means it's a negative number
            bitClear(celsius, 11); // clear it so it won't mess up
               the number
            bitSet(celsius, 15); // put the sign bit in the righ
               place (most significant bit)
      }

      if (0x800 & celsius) // means it's negative
      {
            celsius &= 0x7FF; // clear bit 11 so it won't screw up
               number
            celsius |= 0x8000; // put sign bit on 15 (most
               significant bit)
      }

      //// check for sign on bit 11 (old bit 31):
      //if (bitRead(celsius, 11))
      //{   // means it's a negative number
      //    bitClear(celsius, 11); // clear it so it won't mess up
         the number
      //    bitSet(celsius, 15); // put the sign bit in the righ
         place (most significant bit)
      //}

  return celsius;
}

uint8_t MAX31855::readError()
{
      return spiread32() & 0x7; // bits 0, 1, 2 are 1 if there's an
         error
}

int MAX31855::readFarenheit(void)
{
      float f = readCelsius();
```

```
	f *= 9.0;
	f /= 5.0;
	f += 32;
	return (int)f;
}

uint32_t MAX31855::spiread32(void)
{
	if(hSPI)
	{
	return hspiread32(); // uses hardware SPI
	}

	int i;
	uint32_t d = 0;

	digitalWrite(sclk, LOW);
	_delay_ms(1);
	digitalWrite(cs, LOW);
	_delay_ms(1);

	for (i = 31; i >= 0; i--)
	{
	digitalWrite(sclk, LOW);
	_delay_ms(1);
	d <<= 1;
	if (digitalRead(miso)) {
		d |= 1;
	}

	digitalWrite(sclk, HIGH);
	_delay_ms(1);
	}

	digitalWrite(cs, HIGH);
	//Serial.println(d, HEX);
	return d;
}

uint32_t MAX31855::hspiread32(void)
{
	int i;
	// easy conversion of four uint8_ts to uint32_t
	union bytes_to_uint32
	{
	uint8_t bytes[4];
	uint32_t integer;
	} buffer;
```

```
        digitalWrite(cs, LOW);
        _delay_ms(1);

        for (i = 3; i >= 0; i--)
        {
        buffer.bytes[i] = SPI.transfer(0x00);
        }

        digitalWrite(cs, HIGH);

        return buffer.integer;
}
```