8-12-2016

Multiscale Structure-Property Relationships of Ultra-High Performance Concrete

Megan Noel Burcham

Multiscale structure-property relationships of ultra-high performance concrete

By

Megan Noel Burcham

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Mechanical Engineering
in the Department of Mechanical Engineering

Mississippi State, Mississippi

August 2016

Multiscale structure-property relationships of ultra-high performance concrete

By

Megan Noel Burcham

Approved:

Mark F. Horstemeyer
(Major Professor)

Youssef Hammi
(Committee Member)

Tonya W. Stone
(Committee Member)

D. Keith Walters
(Graduate Coordinator)

Jason M. Keith
Dean
Bagley College of Engineering

Name: Megan Noel Burcham

Date of Degree: August 12, 2016

Institution: Mississippi State University

Major Field: Mechanical Engineering

Major Professor: Mark F. Horstemeyer

Title of Study: Multiscale structure-property relationships of ultra-high performance concrete

Pages in Study: 55

Candidate for Degree of Master of Science

The structure-property relationships of Ultra-High Performance Concrete (UHPC) were quantified using imaging techniques to characterize the multiscale hierarchical heterogeneities and the mechanical properties. Through image analysis the average size, percent area, nearest neighbor distance, and relative number density of each inclusion type was determined and then used to create Representative Volume Element (RVE) cubes for use in Finite Element (FE) analysis. Three different size scale RVEs at the mesoscale were found to best represent the material: the largest length scale (35 mm side length) included steel fibers, the middle length scale (0.54 mm side length) included large voids and silica sand grains, and the smallest length scale (0.04 mm side length) included small voids and unhydrated cement grains. By using three length scales of mesoscale FE modeling, the bridge of information to the macroscale cementitious material model is more physically based.

DEDICATION

I would like to dedicate this work to all my family and friends who have helped me get this far. To my parents for their love and support in everything I have done. I have been blessed to have two parents with engineering graduate degrees to help me through this process. To my sister, Hannah, who has proofread every document I have ever written (including this one). To my Granddaddy Olan who has inspired me with his passion for math and physics.

ACKNOWLEDGEMENTS

I would like to sincerely thank my advisor, Dr. Mark Horstemeyer, for his support throughout my research. I am also very appreciative of Dr. Howard, who helped me understand the ins and outs of concrete. I would like to thank the Engineering Research and Development Center (ERDC) for providing the funding to make this research possible. Lastly, a big thanks is necessary to all my colleagues at the Center for Advanced Vehicular Systems (CAVS) who helped me along the way.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER I

INTRODUCTION AND LITERATURE REVIEW

Ultra-High Performance Concrete (UHPC) is defined as having a compressive strength over 150 MPa and a tensile strength over 8 MPa [Habel et al. 2006]. This strength is accomplished by the exclusion of coarse aggregate, inclusion of silica fume, and reduction of the water to binder ratio [Zadeh et al. 2008]. Figure 1.1 shows the macroscopic differences between ordinary concrete and UHPC. The adjustments made to UHPC also improve the density of the mixture, making it less permeable than ordinary concrete. The impermeability makes the material useful in highly corrosive environments and environments subjected to high strain rates [Charron et al. 2006].

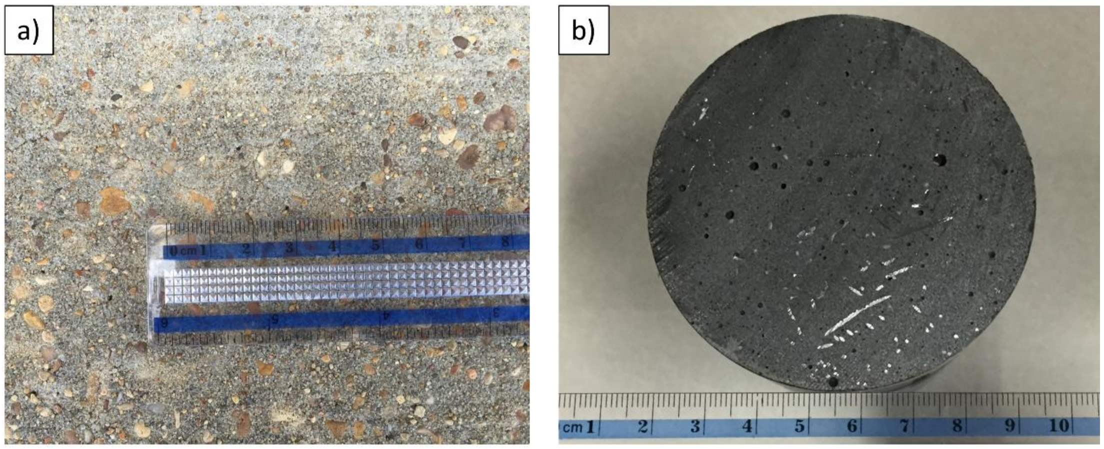

Figure 1.1 Macroscopic comparison of a) ordinary concrete and b) ultra-high performance concrete.

The differences in aggregate sizes and the presence of steel fibers differentiate the two materials.

A key aspect in the enhanced performance of UHPC is the use of steel fibers. For over 50 years, Fiber-Reinforced Concrete (FRC) has been a promising avenue for cementitious material improvement [Zollo 1997]. Ordinary concrete displays characteristics associated with brittle materials, but FRC has shown enhanced ductility in tensile testing. These improvements have been linked to fibers that bridge cracks at different length scales [Scott et al. 2015]. Work has been done to determine the material characteristics of these fibers and methods to improve their performance [Rivera-Soto et al.]. Experimentation of fiber size, shape, and material has helped improve the pull-out properties of the fibers. Figure 1.2 shows the size and shape of fibers used throughout this study.

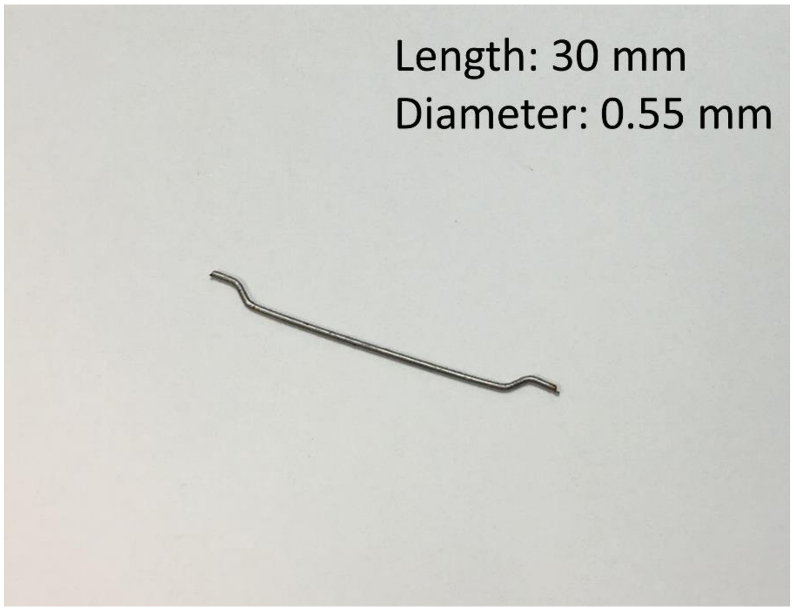

Figure 1.2 Size and shape of fibers used in ultra-high performance concrete throughout this work.

With UHPC's densely packed morphology, the primary length scale of focus is the mesoscale. This range shows the distribution and sizes of all the pertinent constituents, which include steel fibers, sand grains, unhydrated cement grains, silica fume, and voids. Nano-indentation tests were conducted to quantify some of the mesoscale material properties of these constituents by Moser et al. [2013]. Computed Tomography (CT) has also been used to quantify the volumetric characteristics of the material's inclusions [Huang et al. 2015]. These characteristics further confirm the multiscale properties of cementitious materials, which are key in fully understanding the material's behavior [Unger and Eckardt 2011].

The goal of modeling cementitious materials was in place long before the technology, but by the early 1990s strides were made towards this objective [Vecchio 1992]. Further models began to utilize the two main phases of concrete, the aggregate and matrix, and development of these basic models has led to more robust and reliable results [Lopez et al. 2008]. The complexity of these models has increased, and multiple constituents have been included along with the Interfacial Transition Zone (ITZ) [Garboczi and Bentz 1995]. From there, the size and shapes of constituents have been considered in finite element models [Wang et al. 2016].

From nanometer sized voids to centimeter length steel fibers, UHPC has a range of lengths that is incompatible with structural scale Finite Element Analysis (FEA). This disagreement stems from the element sizes that are best suited for each inclusion. These difficulties in modeling can result in an inability to represent the "coupling effect" that the microstructural features have at increasing length scales [Gokhale and Yang 1999].

The use of multiple length scales in modeling would help capture this effect, and improve cementitious modeling.

The Chemistry-Process-Structure-Property-Performance (CPSPP) sequence is a useful technique in multiscale modeling. In this work, the CPSPP structure will aid in determining the cause-effect quantification of relationships within UHPC. Figure 1.3 outlines this method. While cement chemistry is not the main focus of this work, it is important to consider the unique constitution of cement when modeling concrete. Curing is the method by which UHPC hardens and results from hydration of the cement particles creating a glue to bond the inclusions together. UHPC’s structure has features at various length scales, and the distribution of inclusions impact the material’s performance. Several properties were found via mechanical testing, and these results helped determine the performance of the material in terms of fracture and ductility.

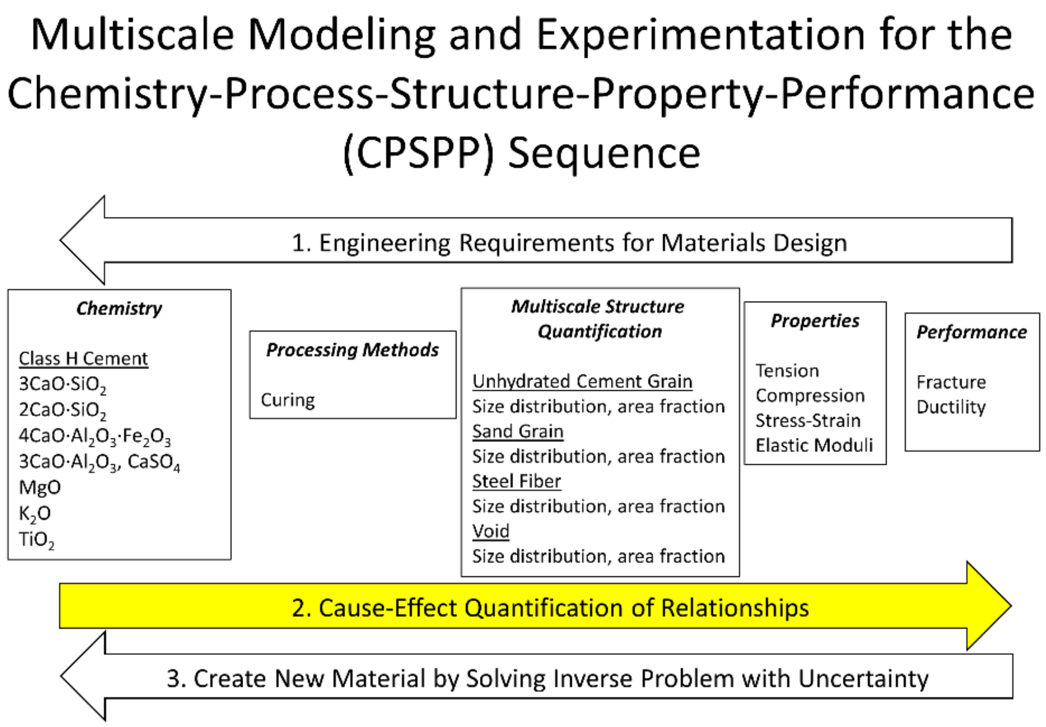
Multiscale Modeling and Experimentation for the Chemistry-Process-Structure-Property-Performance (CPSPP) Sequence
1. Engineering Requirements for Materials Design
Chemistry
Class H Cement
$3CaO{\cdot}SiO_2$
$2CaO{\cdot}SiO_2$
$4CaO{\cdot}Al_2O_3{\cdot}Fe_2O_3$
$3CaO{\cdot}Al_2O_3$, $CaSO_4$
MgO
K_2O
TiO_2
Processing Methods
Curing
Multiscale Structure Quantification
Unhydrated Cement Grain
Size distribution, area fraction
Sand Grain
Size distribution, area fraction
Steel Fiber
Size distribution, area fraction
Void
Size distribution, area fraction
Properties
Tension
Compression
Stress-Strain
Elastic Moduli
Performance
Fracture
Ductility
2. Cause-Effect Quantification of Relationships
3. Create New Material by Solving Inverse Problem with Uncertainty

Figure 1.3 Chemistry-process-structure-property-performance method for determining cause and effect relationships.

This work focuses on the forward looking cause-effect relationships.

Since the 1980s, the Engineer Research and Development Center (ERDC) has been testing UHPC in varying applications and has developed a UHPC material called Cor-Tuf Baseline [Green et al. 2014]. All further references to UHPC will imply the use of Cor-Tuf Baseline mix and its curing procedures. This material will be the focus of this study, as accurately modeling this material is of current interest. Knowledge of the mechanical properties and mesoscale characteristics of Cor-Tuf Baseline will enable improved predictability of the material model.

CHAPTER II

MATERIAL CHARACTERIZATION

To determine the cause-effect relationship within UHPC, several experimental methods were employed. These processes included examination of the chemistry of cementitious materials, the curing procedure of UHPC, Scanning Electron Microscope (SEM) imaging, ImageJ image analysis, CT scanning, and mechanical testing. At the structural scale, a UHPC beam in bending was modeled with FEA assuming a homogenous medium. Then, the mesoscale characteristics of UHPC, which included different length scales of heterogeneous structures, were broken down into three length scales for improved modeling results. Constituents quantified were steel fibers, large voids, sand grains, unhydrated cement grains, and small voids.

2.1 Cement Chemistry

Cement chemistry is a complicated field, as cement is made by heating materials like limestone and clay until they partially fuse. These materials are then mixed with calcium sulfate and ground down into a powder [Taylor 1997]. This process can result in cement grains with varying properties, which can result in different reactions during hydration. Ordinary concrete is produced by mixing water, Portland cement, and aggregate. The cement is hydrated by the water and forms a paste; this paste acts as a glue that holds the aggregate together. Class H cement, which is commonly used in oil wells, was used in the UHPC tested throughout this work. The chemistry associated with this material is shown in Table 2.1. Upon hydration, the unhydrated cement grains go through the following reactions,

$$2(3CaO \cdot SiO_2) + 11H_2O = 3CaO \cdot 2SiO_2 \cdot 8H_2O + 3(CaO \cdot H_2O) \quad (2.1)$$

$$2(2CaO \cdot SiO_2) + 9H_2O = 3CaO \cdot 2SiO_2 \cdot 8H_2O + CaO \cdot H_2O \quad (2.2)$$

$$4CaO \cdot Al_2O_3 \cdot Fe_2O_3 + 10H_2O + 2(CaO \cdot H_2O) = 6CaO \cdot Al_2O_3 \cdot Fe_2O_3 \cdot 12H_2O \quad (2.3)$$

$$3CaO \cdot Al_2O_3 + 3(CaO \cdot SO_3 \cdot 2H_2O) + 26H_2O = 6CaO \cdot Al_2O_3 \cdot 3SO_3 \cdot 32H_2O \quad (2.4)$$

where $3CaO \cdot 2SiO_2 \cdot 8H_2O$ is Calcium Silicate Hydrate (C-S-H), which is the primary form of cementitious binder [Kosmatka and Wilson 2011].

Table 2.1 Phases and percentages present in Class H cement [Jupe *et al*. 2008].

Phases Present	**Cement Chemist**	**% Present**
3 CaO • SiO_2	C_3S	47.1
2 CaO • SiO_2	C_2S	28.3
4 CaO • Al_2O_3• Fe_2O_3	C_4AF	17.0
3 CaO • Al_2O_3	C_3A	0.65
$CaSO_4$	$C\bar{S}$	4.72
MgO	M	1.09
K_2O	K	0.45
TiO_2	T	0.18

2.2 Curing

All UHPC examined in this study followed the same curing procedure, which lasted a total of two weeks. Specimens were mixed together, poured into molds, and then kept moist for 24-48 hours until completely set. After a day or two in the molds, specimens were removed and placed in a Fog Room (100% humidity and 21.1-25°C) for six days. When the six days were over, the specimens were placed into a room temperature water bath and then the bath was heated to 90°C to reduce the likelihood of cracking due to sudden temperature change. Steam curing in the water bath lasted seven days, after which the water bath was dropped back to room temperature. The specimens were then removed from the bath and shelved to await testing.

2.3 Imaging

Lower magnification images were taken using a ZEISS Axiovert 200 optical microscope at the Center for Advanced Vehicular Systems (CAVS). A SUPRA 40 FEG-SEM, also at CAVS, was used to take higher magnification images. The surfaces of the concrete specimens were ground smooth before imaging to ensure quality pictures. The combination of the low and high magnification microscopes allowed for images to be taken at a variety of length scales to accurately capture the distributions of each constituent.

2.4 ImageJ

The image processing tool, ImageJ, was used for analyzing images [Schneider *et al.* 2012]. This software was utilized to identify, isolate, and measure constituents. Figure 2.1 shows this process. These results allowed for the determination of the average size, number density, area fraction, and nearest neighbor distance of each constituent. ImageJ was used on multiple images at varying length scales to ensure accurate results were found for each of the constituents.

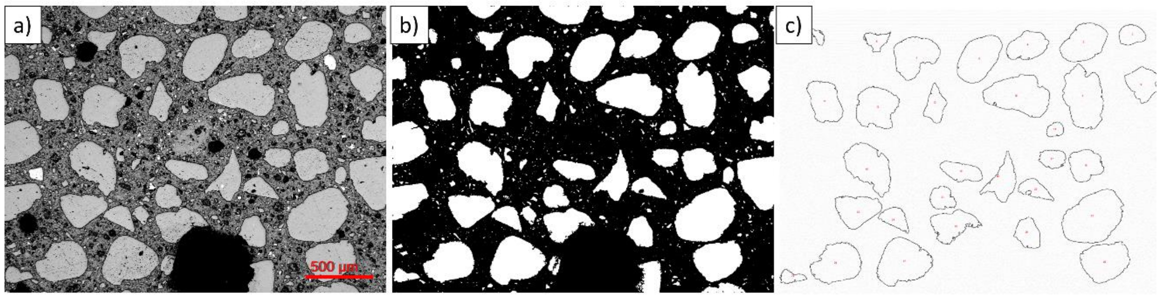

Figure 2.1 ImageJ process for turning a) scanning electron microscope images to a b) binary image and then to c) isolated inclusions.

2.5 Computed Tomography (CT) Scan

The Phoenix X-Ray CT system with dual focus, reaching one micron resolution, at CAVS was used for 3D imaging. CT scans were conducted on a UHPC cube with sides of approximately 50 mm. The CT scan could only distinguish the larger constituents, large voids, and steel fibers, helping to determine the 3D characteristics of UHPC. The large voids, which arise from entrapped air during mixing, had previously been ignored until these scans showed their high frequency of occurrence. The orientation of the fibers was assumed to be random, but the CT scan aided in observing the locations of the fibers within the matrix.

2.6 Mechanical Testing

Mechanical testing was accomplished using the equipment at the Construction Materials Research Center at Mississippi State. Cylinders of Cement Paste (CP), Mortar (M), Fiber-Reinforced Paste (FRP), and UHPC were subjected to testing to obtain elastic moduli, stress-strain behavior, compressive strengths, and tensile strengths. The materials involved in each of the constituent mixtures are outlined in Table 2.2. Compressive tests were conducted according to ASTM C39, tensile tests were conducted according to ASTM C496, and elastic modulus tests were conducted according to ASTM C469. Figure 2.2 shows the setup that was used in the compression and tension tests. The testing of these combinations of constituents enabled more accurate FE modeling, as the characteristics of each constituent were known.

Table 2.2 Breakdown of the material (inclusion) types used in each constituent mixture.

Constituent	Cement Matrix	Voids	Unhydrated Cement Grains	Sand Grains	Steel Fibers
Cement Paste	X	X	X		
Mortar	X	X	X	X	
Fiber-Reinforced Paste	X	X	X		X
Ultra-High Performance Concrete	X	X	X	X	X

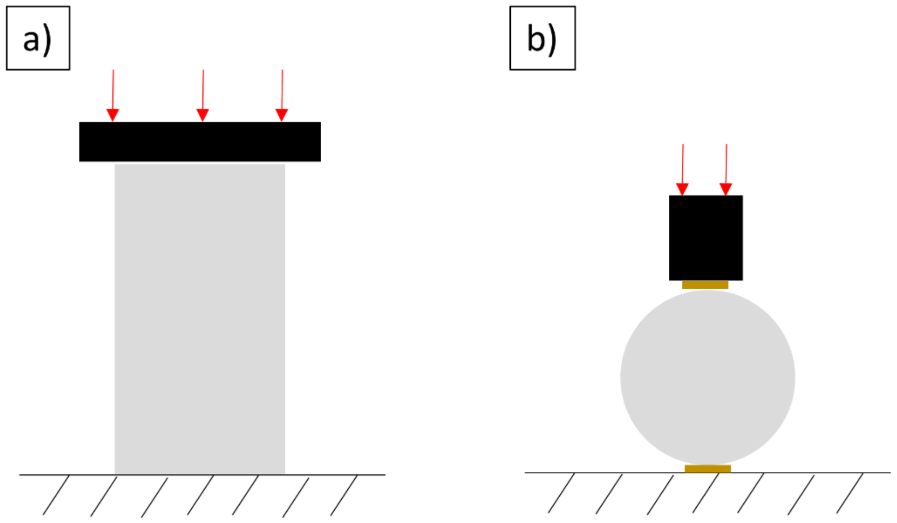

Figure 2.2 Schematic of the a) compression and b) indirect tensile tests completed on ultra-high performance concrete and constituents.

CHAPTER III

FINITE ELEMENT MODELING

UHPC modeling was accomplished using the software Abaqus CAE version 6.14 [Abaqus 2014]. This work focused on the generation of Representative Volume Element (RVE) cubes with various inclusions suspended in a matrix.

3.1 Structural Scale Finite Element Modeling

Within Abaqus CAE, a structural scale beam in bending was selected for modeling in order to fully capture the heterogeneous properties of UHPC. The size of the UHPC beam and the boundary conditions applied mirrored previous experiments conducted at ERDC [Roth 2008]. Figure 3.1 shows the test setup, the beam's dimensions, and the boundary conditions used of the FEA. This beam was treated as a continuum, containing no inclusions, with material properties determined by ERDC's testing results [Scott *et al*. 2015].

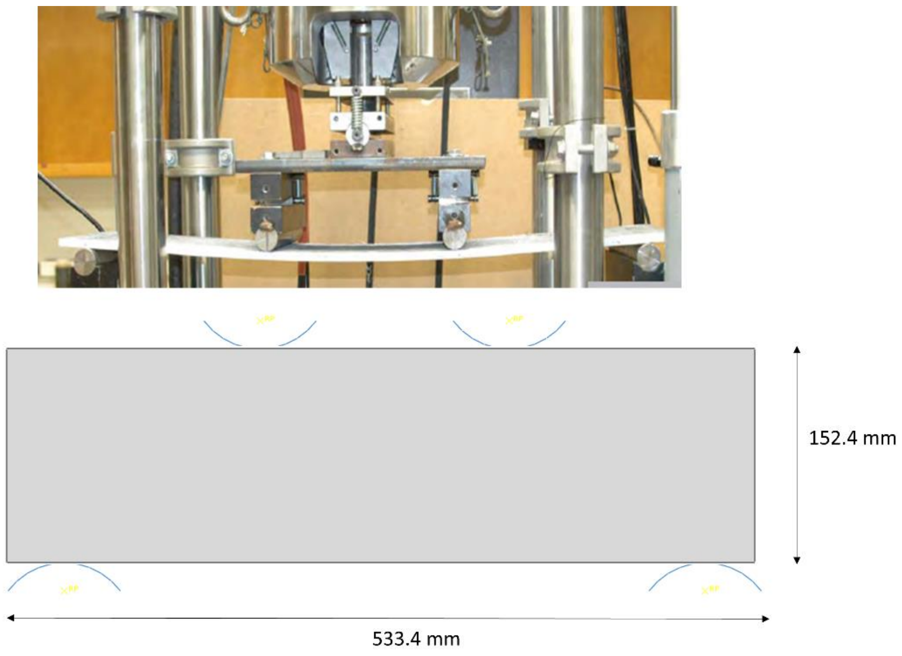

Figure 3.1 Ultra-high performance concrete beam used in finite element simulations with dimensions and boundary conditions shown.

This beam represents the structural scale (Length Scale 4) and is modeled after physical tests.

3.2 Mesoscale Finite Element Modeling

It was determined that three length scales would aid modeling the mesoscale characteristics of the UHPC beam in bending, as there were such differences in the size of constituents. The largest cube had sides of 35 mm, and it included only steel fibers within the matrix. The middle length scale included large voids and silica sand grains and had sides of 0.54 mm. The smallest length scale was a cube with sides of 0.04 mm, and it included small voids and unhydrated cement grains. To best portray the material, inclusions were randomly distributed throughout the matrix according to their measured distributions.

Two python scripts were written to generate these RVE cubes. The first script asked for an average inclusion size, found using ImageJ, and inclusions of that size were inserted into the matrix until the desired volume fraction was reached. Plots of constituent size versus number of occurrences revealed that using only the average constituent size left out the range of inclusion sizes found in UHPC. The size versus number of occurrences graph was fitted with a distribution curve, and a second script was created that asked for the mean and variance of this fitted curve. The inclusions inserted into the matrix by this second code were of sizes that varied according to the distribution curve.

Both random generation codes required the size of the cube, as well as the number, shape, and volume fraction of each constituent. Two shapes, spheres and cylinders, were used in modeling the inclusions in subsequent simulations. Spheres were used to model the voids, unhydrated cement grains, and sand grains; cylinders were used to model the steel fibers. The algorithms then randomly inserted and distributed these shapes, making sure none overlapped (see Appendix A).

CHAPTER IV

RESULTS

The results from image analysis, mechanical testing, geometry generation, and finite element analysis are discussed in the following section.

4.1 Multiscale Materials Morphology

Area fractions of constituents were found using Image J, and Figure 4.1 shows a sample of the images used for analysis. The length scales of images varied greatly to accurately capture the constituents in RVE cubes with sides of 35 mm, 0.54 mm, and 0.04 mm. In the case of voids, an area criteria of 0.01 mm2 was established to distinguish between small and large voids. This distinction aided in modeling, as both microscopic and macroscopic voids were present in the UHPC. Large voids occurred as the wet concrete was poured into molds. Vibrating tables were used to shake the material down into the molds and reduce the size and occurrence of voids. Smaller voids were likely caused by improper bonding of the matrix to inclusions, such as sand grains and fibers.

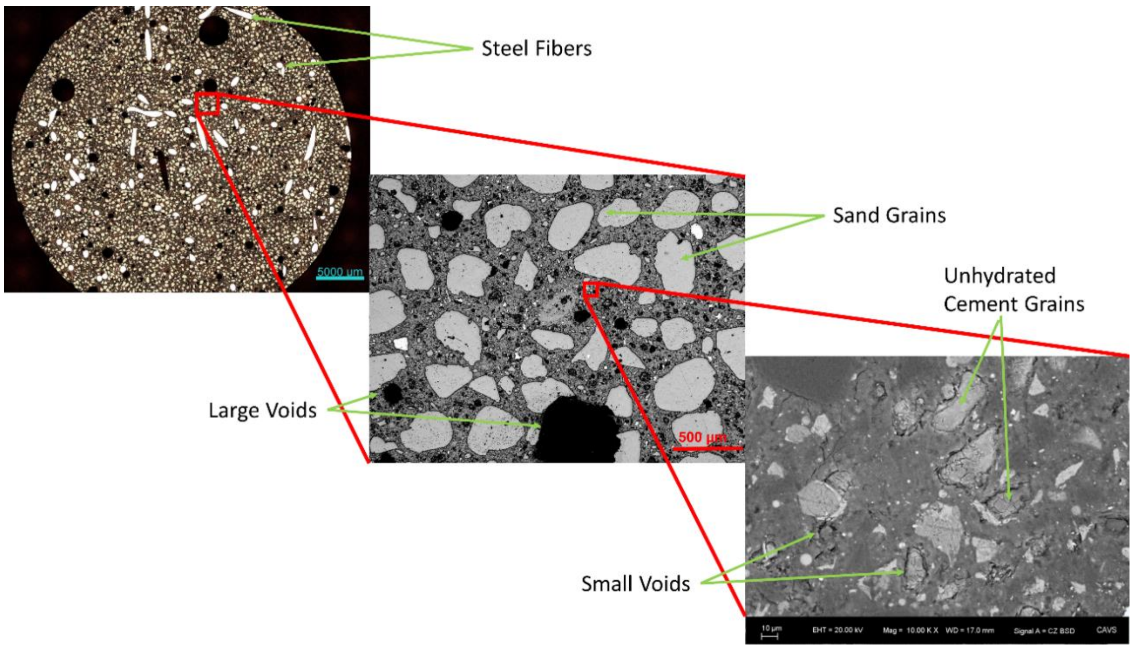

Figure 4.1 Images of ultra-high performance concrete at three mesoscale length scales with inclusions of interest pointed out.

To determine the sizes of RVE cubes for simulations, the number of occurrences for each size of each inclusion was determined. Figure 4.2 shows the resulting curves. Divisions of each length scale are also shown for reference. These length scales became the side length for RVE cubes. Large voids were included in the middle RVE, and small voids were included in the small RVE. Table 4.1 shows the resulting area fractions found through image analysis.

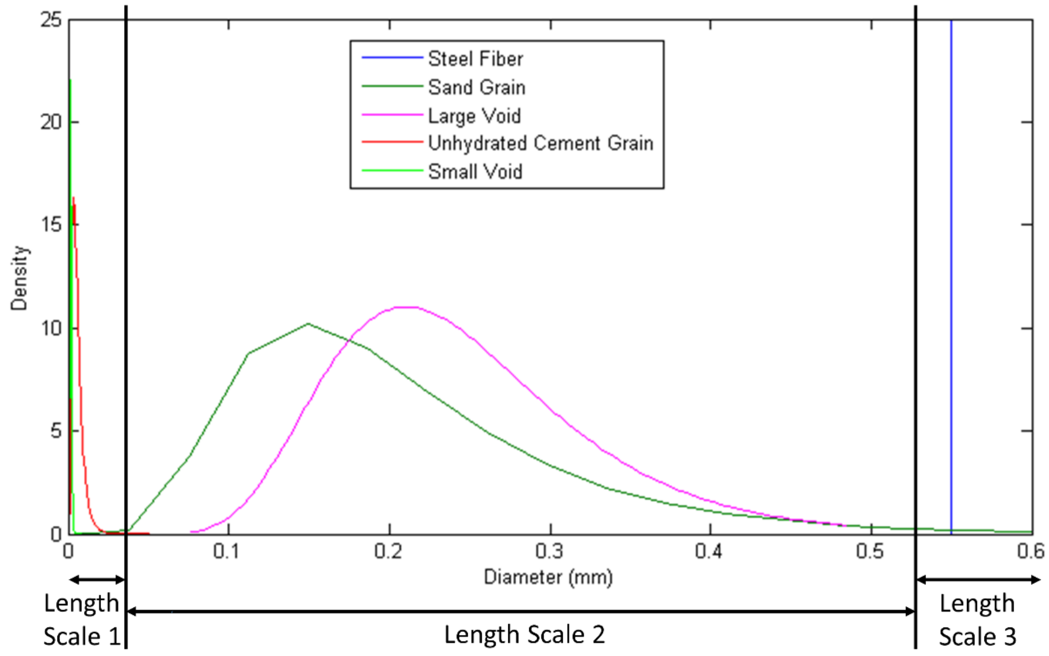

Figure 4.2 Size distributions of the inclusions in ultra-high performance concrete with length scales used in representative volume element cubes.

Table 4.1 Average values for inclusions in ultra-high performance concrete found through image analysis.

Inclusion	Radius (mm)	Area %	Nearest Neighbor Distance (mm)	Number Density (mm^{-2})
Small Void	0.000853	0.317	0.00832	1,350
Unhydrated Cement	0.00310	11.7	0.0114	2,290
Sand	0.123	22.3	0.322	4.44
Large Void	0.108	6.49	0.526	0.951
Steel Fiber	0.275	3.68	0.919	0.0593
Matrix	-	55.51	-	-

From the average values determined through image analysis, the damage can be quantified using

$$\varphi = \eta \upsilon \tag{4.1}$$

where φ is the damage, η is the number density (mm-2), and υ is the inclusion area (mm2). The use of this equation with values from Table 4.1 allowed for the results displayed in Table 4.2. These values are approximations, as the area of each void was assumed to be a perfect circle. The poor bond between the fibers and the matrix causes the fibers to act as cracks when the material is in compression. Using the volume of each fiber (7.4 mm3), volume of the cylindrical specimen (667,000 mm3), and number of fibers in each specimen (~2830), the damage associated with the fibers was found to be 0.0314.

Table 4.2 Calculated damage values for inclusions in ultra-high performance concrete.

Inclusion	**Damage**
Small Void	0.00309
Large Void	0.0349

The specimen of UHPC used for CT scanning gave interesting insight into the UHPC material. Steel fibers within the matrix are supposed to be randomly distributed and oriented, but as observed in Figure 4.3, the fibers all tend toward a similar orientation. The large number of macroscopic voids within the material was not known until this scan showed a high concentration of sizable voids. At this point, the decision to account for voids at two length scales was made.

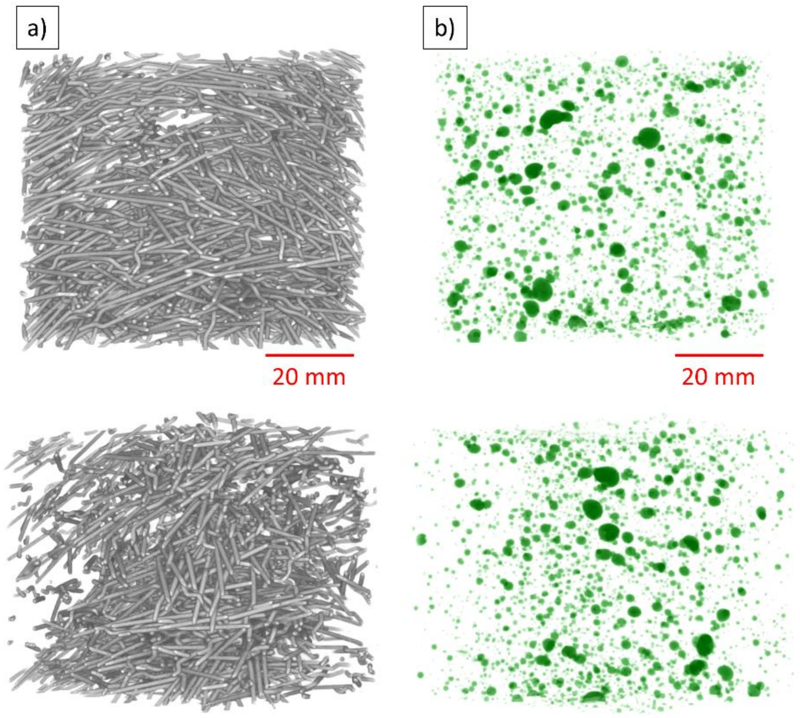

Figure 4.3 Computed Tomography (CT) scan of ultra-high performance concrete showing a) approximately 3% steel fibers and b) large voids illustrated; note that although the material's behavior is isotropic, there is local anisotropy from the fibers.

The volume fractions of air, cement, sand, and fibers were calculated for each specimen tested. These values were found using the known batching mass percentages of each inclusion, and the mass of each inclusion in a specimen was determined. Using the specific gravity of each inclusion (besides air), the volume of these inclusions were established. These values were found for M, FRP, and UHPC specimens tested.

By adding the volume fractions of the inclusions with the volume fractions of water and various admixtures, an "ideal" volume was found. This "ideal" volume contained no air; therefore, the difference between the "ideal" volume and the recorded volume of the specimen was assumed to be air. Two major assumptions were made in making these calculations. The first assumption was that no water was absorbed by the sand, and that all the water in each specimen was retained throughout the entire curing and testing process. The second assumption was that each of these specimens was ideally batched, as each batch had enough materials to make 3 specimens. The results of these calculations are shown in Table 4.3.

Table 4.3 Average volume fractions of the inclusions in ultra-high performance concrete specimens.

	Average Volume %		
Inclusion	**Mortar**	**Fiber-Reinforced Paste**	**Ultra-High Performance Concrete**
Air	4.54	1.83	2.42
Sand Grains	18.1	-	29.2
Fibers	-	5.18	3.18

4.2 Mechanical Behavior

Mechanical testing enabled appropriate modeling by revealing information about constituents in UHPC. The results from compression and tension tests are shown in Figure 4.4. As cracks developed in the specimens, the stress-strain behavior became unreliable; therefore, only the earlier (mostly linear) portions of the stress-strain curves were plotted. A direct comparison of each constituent’s compressive and tensile properties is shown in Figure 4.5 to highlight the different stress state dependences.

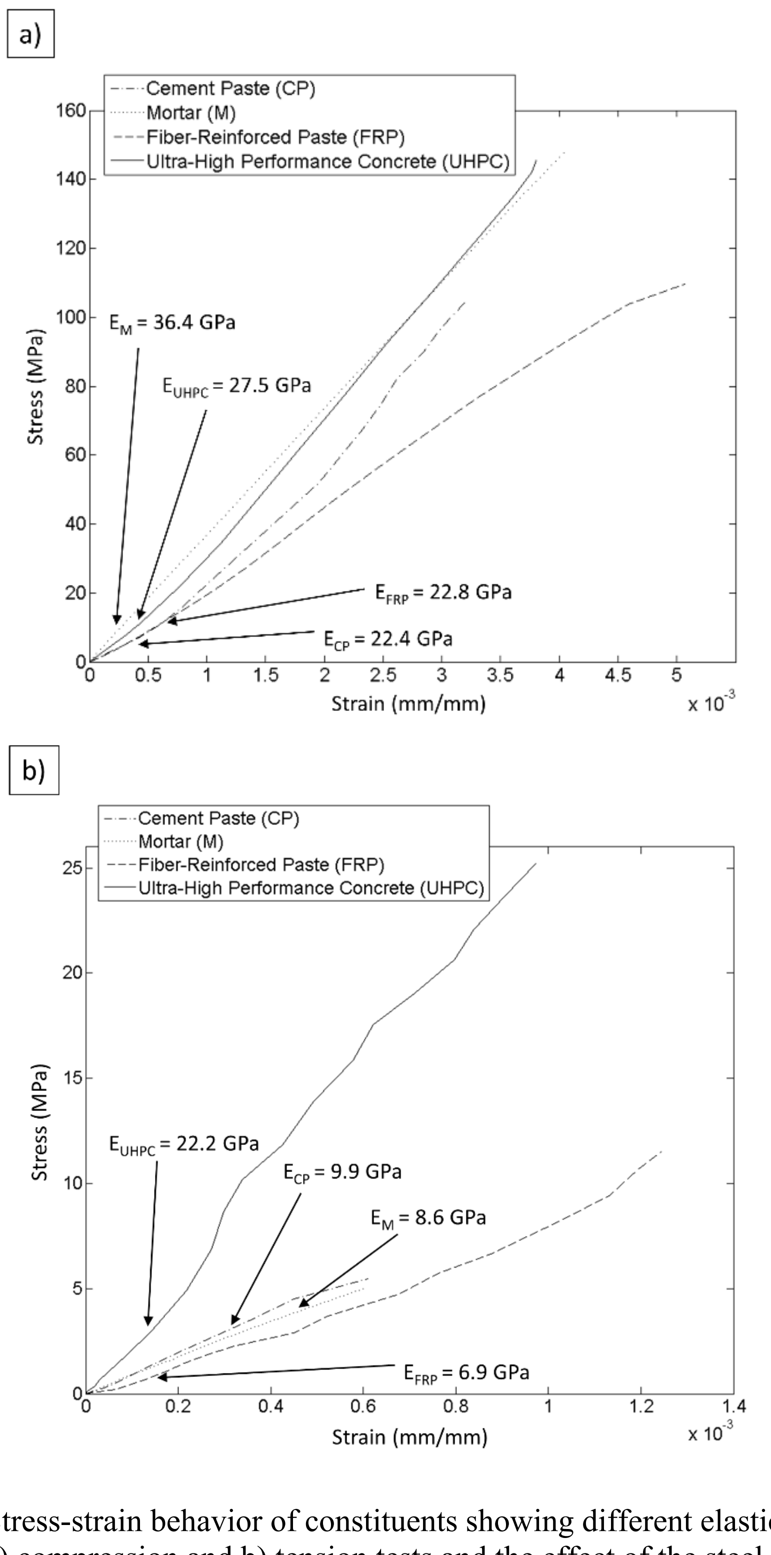

Figure 4.4 Stress-strain behavior of constituents showing different elastic moduli for a) compression and b) tension tests and the effect of the steel fibers included.

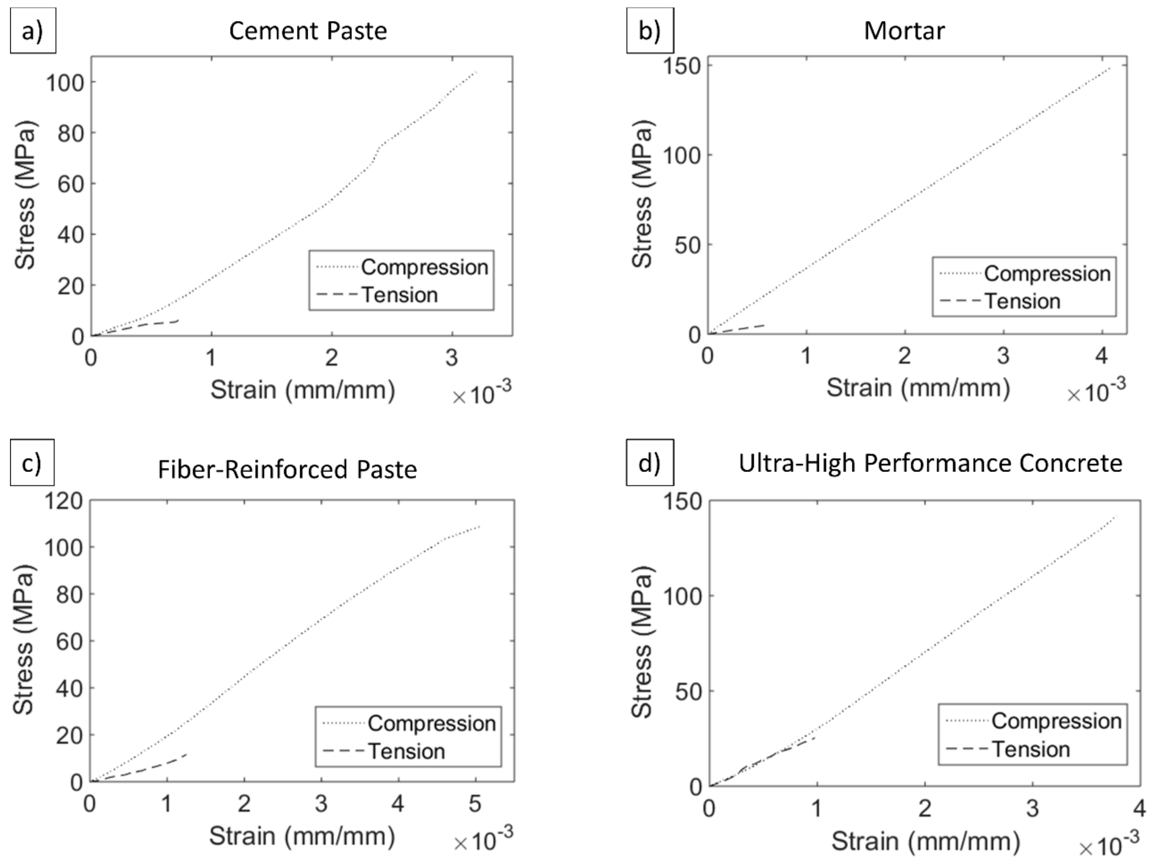

Figure 4.5 Comparison of each material's compressive and tensile stress-strain behavior to highlight cementitious materials' heterogeneity in a) cement paste, b) mortar (cement paste and sand), c) fiber-reinforced paste (cement paste and steel fibers), and d) ultra-high performance concrete (cement paste, sand, and steel fibers).

During testing, the elastic modulus was also found using a compressometer. These values were compared to the elastic moduli determined from compressive stress-strain testing, and the results are shown in Table 4.4. Elastic moduli and Poisson's ratios for inclusions in UHPC were found in the literature and are shown in Table 4.5. These values were used with the Simple Rule of Mixtures (SROM) which was calculated by

$$E_M = (EV)_{air} + (EV)_{sand} + (EV)_{cement\ paste} \tag{4.2}$$

$$E_{FRP} = (EV)_{air} + (EV)_{steel\ fibers} + (EV)_{cement\ paste} \tag{4.3}$$

$$E_{UHPC} = (EV)_{air} + (EV)_{sand} + (EV)_{steel\,fibers} + (EV)_{cement\,paste} \tag{4.4}$$

where E is the elastic modulus from Tables 4.4 and 4.5, and V is the volume fraction from Table 4.3. The information gathered from stress-strain behavior, compressometer results, pulse velocity results, and the SROM are compared in Table 4.6.

Table 4.4 Percentage difference in elastic moduli found through compressometer and stress-strain behavior, which led to the calculation of an average modulus and standard deviation

Constituent	% Difference	Average Modulus (GPa)	Standard Deviation (GPa)
Cement Paste	38.93	29.54	11.5
Mortar	12.29	38.99	4.8
Fiber-Reinforced Paste	29.53	27.60	8.2
Ultra-High Performance Concrete	44.39	38.52	17.1

Table 4.5 Elastic modulus and Poisson's ratio for inclusions in ultra-high performance concrete.

Inclusion	Elastic Modulus (GPa)	Poisson's Ratio
Unhydrated Cement Grain	135 [Smilauer and Bittnar 2006]	0.3 [Davydov *et al*. 2011]
Sand Grain	87.6 [Lutz *et al.* 1997]	0.17 [Lutz *et al.* 1997]
Steel Fiber	200 [Maalej and Li 1994]	0.3 [Maalej and Li 1994]

Table 4.6 Comparison of elastic moduli found through various procedures for constituents of ultra-high performance concrete.

Constituent	**Compression Strain Gage (GPa)**	**Tension Strain Gage (GPa)**	**Compresso-meter (GPa)**	**Pulse Velocity (GPa)**	**Simple Rule of Mixtures (GPa)**
Cement Paste	22.40	9.866	36.68	-	-
Mortar	36.44	8.612	41.55	-	44.21
Fiber-Reinforced Paste	22.81	6.965	32.37	-	44.47
Ultra-High Performance Concrete	27.53	22.19	49.51	57.90 [Williams *et al.* 2009]	55.58

The mean size of each constituent was found using a Matlab script. This script took the size of each individual constituent and plotted the number of occurrences of each size. These plots show the wide distribution of sizes within this material, and a lognormal curve was fitted to the probability curve. Figure 4.6 shows the fitted distributions for the constituents of concern in UHPC. The equations for the lognormal curves enabled modeling of constituents with distributed sizes. Values were input into the random generation code, and RVE’s with different sized inclusions were made. Table 4.7 compares the mean radii and variances found through these fitted curves.

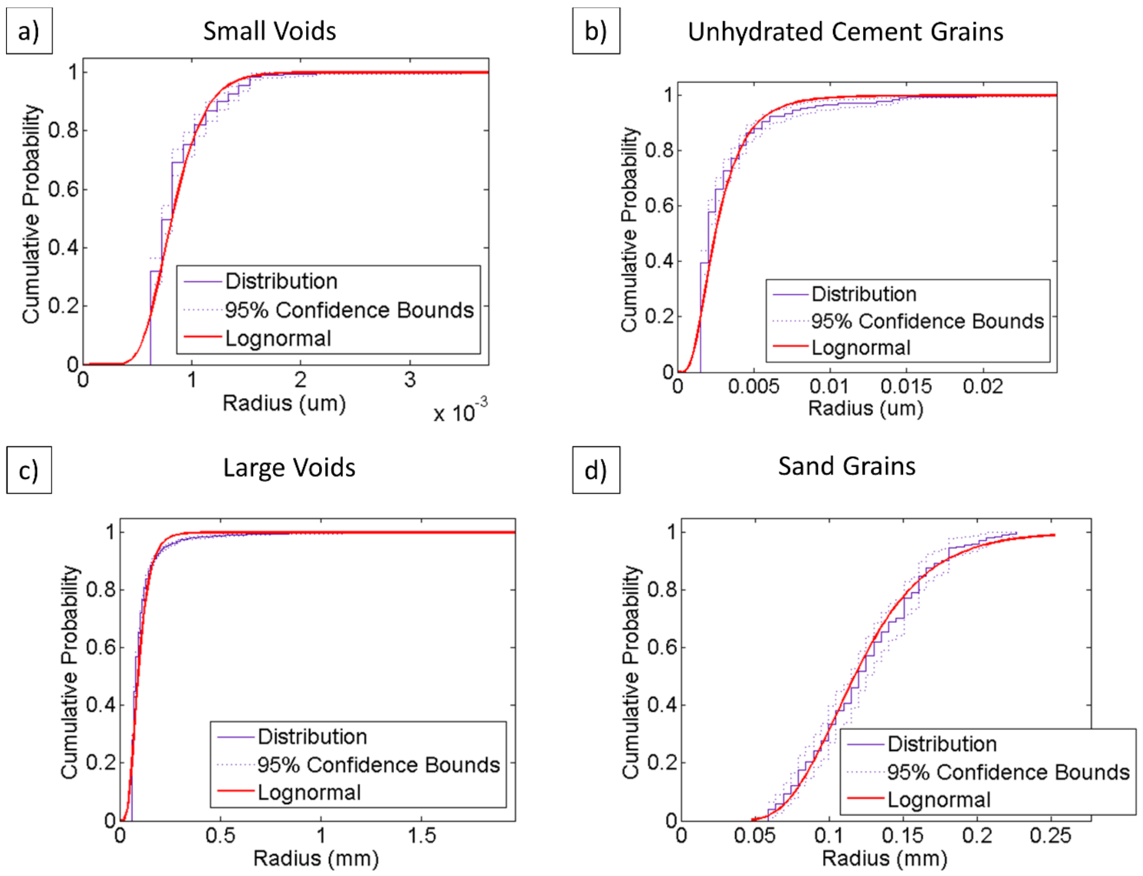

Figure 4.6 Probability of radius occurrence for inclusions in ultra-high performance concrete fitted with lognormal curves.

Table 4.7 Mean radius and variance for each inclusion type from found through lognormal fit curves.

Inclusion	Mean Radius (mm)	Radius Variance
Small Void	0.00090	0.000000063
Unhydrated Cement Grain	0.0029	0.0000035
Large Void	0.10	0.0026
Sand Grain	0.12	0.0017

4.3 Finite Element Analysis

A structural scale, UHPC beam in bending was simulated in the FE code Abaqus [Abaqus 2014] using the specifications of an experiment conducted by ERDC [Scott et al.

2015]. The mesh created on this beam included 2,702 quadratic triangular elements and 5,577 nodes. The displacements caused by the loading conditions are shown in Figure 4.7. Also from the simulation, the crack path can be observed by examining the principal strain. This is shown in Figure 4.8. Results from the test ERDC completed were documented, and Figure 4.9 shows the crack propagation from the experiment.

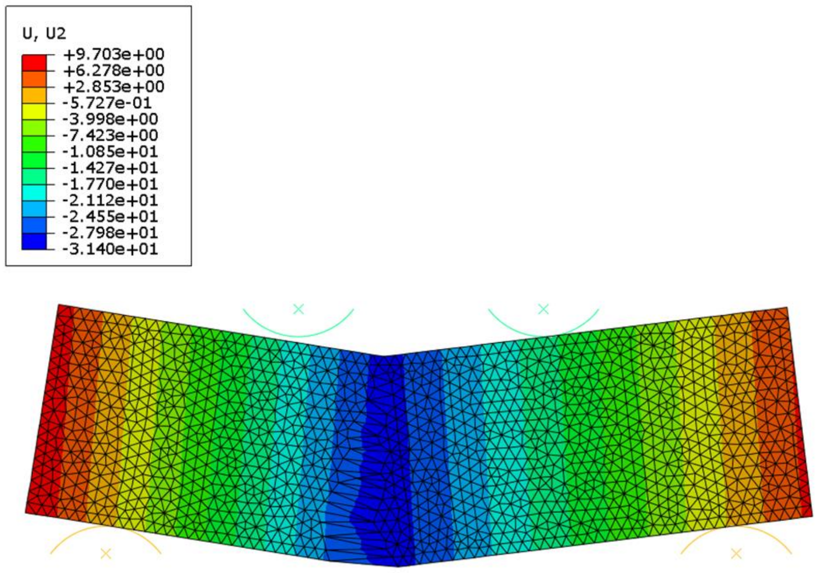

Figure 4.7 Displacement (in mm) in the y-direction caused by the velocity applied to the upper cylinders.

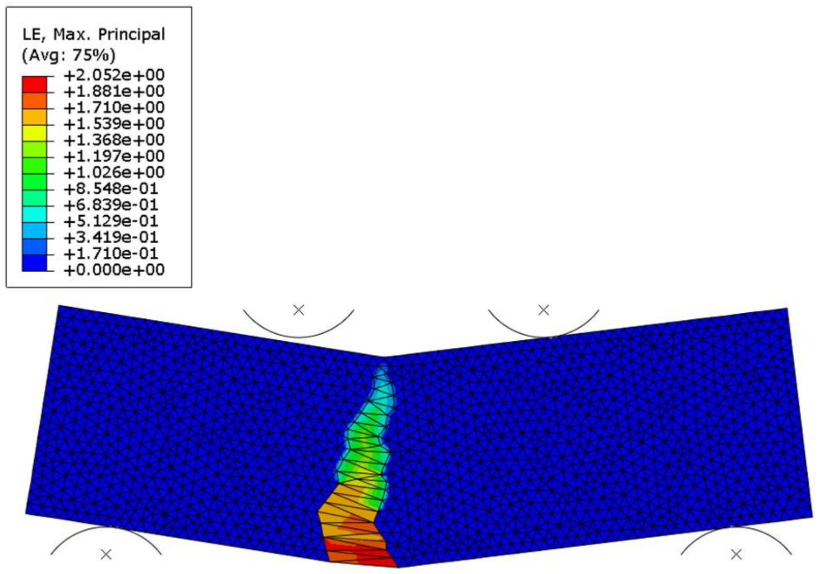

Figure 4.8 Max principal strain of the beam in bending.

Note the likely path of crack propagation that would have started on the tensile face.

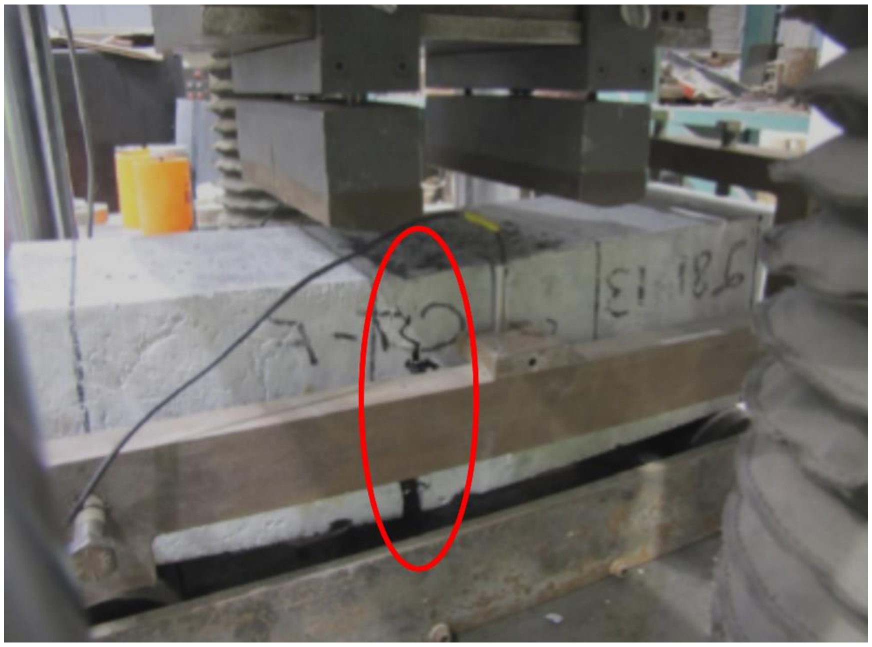

Figure 4.9 Max principal strain of the beam in bending.

Note the likely path of crack propagation that would have started on the tensile face.

4.4 Geometry Generation

The resulting values from Table 4.1 were used to generate three RVE cubes, each of a different size and including different inclusion types. Results from the structural scale beam gave valuable insight into the UHPC's behavior, but simulations at the mesoscale will enable results with a new level of accuracy to be found. Figure 4.10 shows the matrix of the each of the three mesoscale RVE cubes. These cubes were then meshed, and the results are shown in Figure 4.11. The mesh of Length Scale 1 was made up of 1,344,350 tetrahedral elements, Length Scale 2 was made up of 887,345 tetrahedral elements, and Length Scale 3 was made up of 2,389,963 tetrahedral elements and 125,557 hexahedral elements. Tetrahedral elements were used on the matrix and spherical inclusions, while hexahedral elements were applied to the cylindrical inclusions. All elements generated were linear to reduce simulation time. Simulations on these meshed cubes will occur in the future using the material properties outlined in this study.

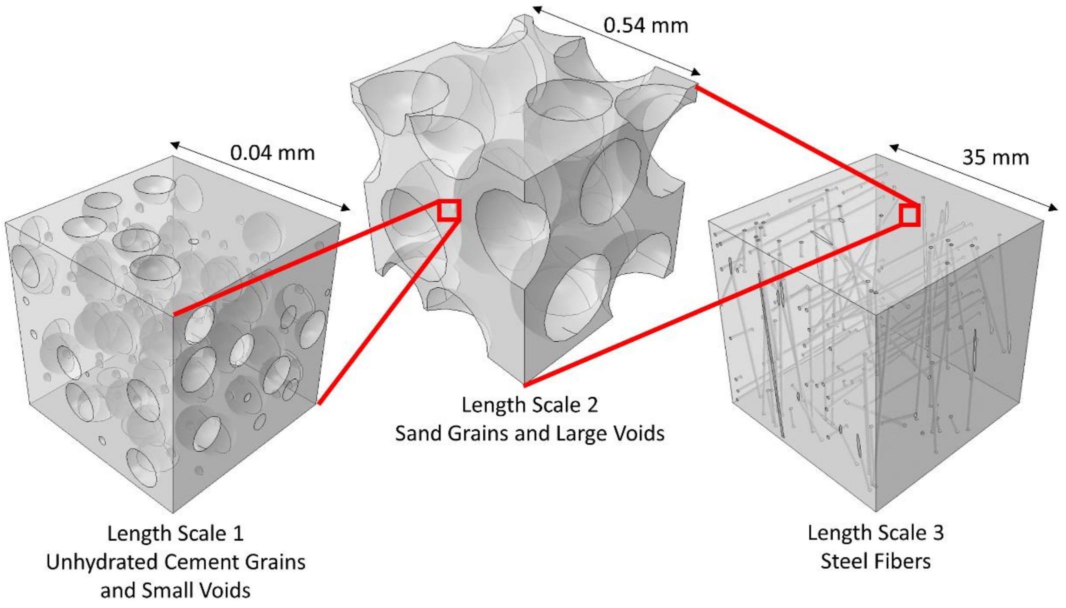

Figure 4.10 Geometries generated using the area fractions and length scales found through image analysis.

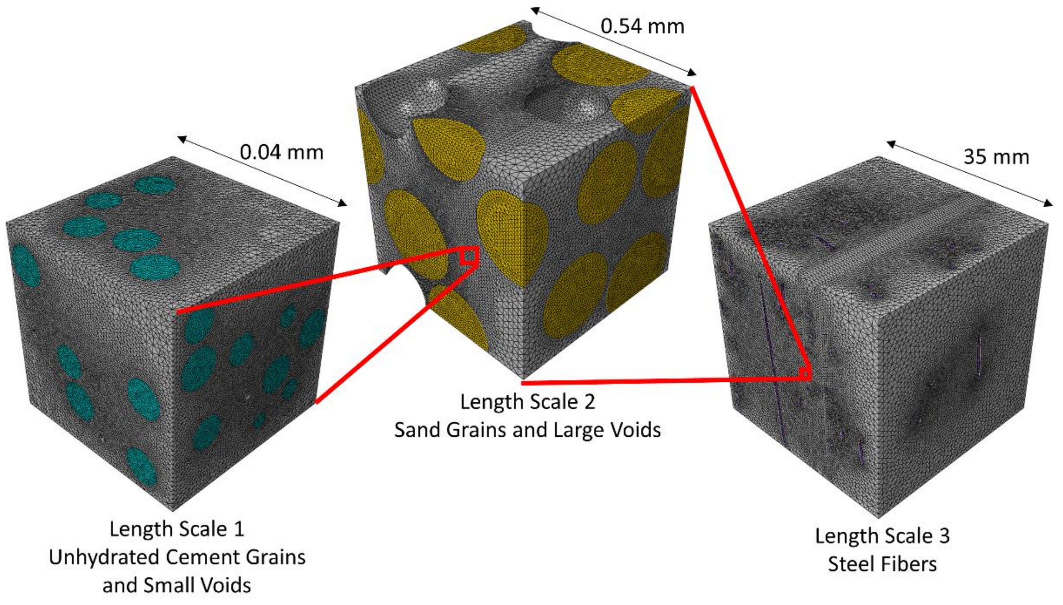

Figure 4.11 Meshed representative volume element cubes at varying length scales showing unhydrated cement grains (blue), sand grains (yellow), and steel fibers (purple).

To capture the distributed sizes of inclusions in UHPC, the results from the lognormal fitted curves were input into the generation process, and Figure 4.12 displays the distributed size RVE cubes. Since the variances are quite small, the different sizes are hard to discern. A cube with inclusions of distributed sizes was not constructed for Length Scale 3, as the fibers were manmade to be the same size. These RVE cubes will be beneficial in future work involving mesoscale finite element simulations, as comparisons will be made between results from cubes using the average size inclusion and cubes with inclusions of distributed sizes.

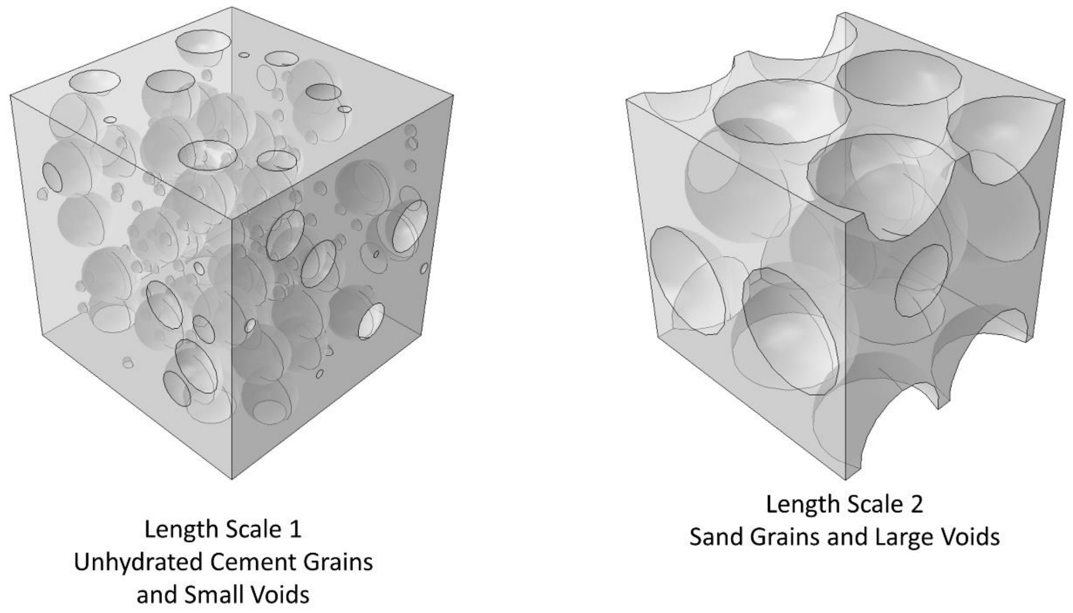

Figure 4.12 Distributed sizes of inclusions based on lognormal fit curves in representative volume element cubes.

CHAPTER V

CONCLUSIONS

Comparing the elastic moduli of the four constituents tested led to several observations. The inclusion of fibers reduced the elastic modulus, as the fibers did not fully bond to the matrix. Sand provided a much better cohesion to the matrix, since mortar had a higher elastic modulus than fiber-reinforced paste in compression. The variances in elastic moduli results from different methods increased the complication of drawing further conclusions about the effects of different inclusions. Tensile testing of cementitious materials provided many challenges, and the results varied widely. The stress-strain curves shown in Figures 4.4 and 4.5 were based on one experiment; therefore, more tests would need to be conducted to make more observations.

The orientation in fibers, observed through the CT scan, occurred during the pouring of the concrete. The similar orientation of the fibers allowed for a greater packing density within the matrix. When randomly generating RVE cubes, this orientation also occurred. The large volume fraction of fibers resulted in this non-random distribution, as all the fibers tried to fit and bond to the matrix.

In the FEA of the UHPC beam in bending, the crack started on the tensile face, which would accurately represent the material. This model would be useful in future work of transferring boundary conditions through the mesoscale lengths. The actual fracture location on the physical beam started closer to the load head, while the FEA simulation showed the crack starting almost equidistant between the two load heads.

Continued work on this project will work to provide more accurate simulations.

REFERENCES

Abaqus CAE (version 6.14). 2014. Dassault Systems.

Charron, J P, E Denarie, and E Bruhwiler. 2006. “Permeability of Ultra High Performance Fiber Reinforced Concretes (UHPFRC) under High Stresses.” *Materials and Structures* 40 (3): 269–77.

Davydov, D, M Jirasek, and L Kopcky. 2011. “Critical Aspects of Nano-Indentation Technique in Application to Hardened Cement Paste.” *Cement and Concrete Research* 41 (1): 20–29.

Garboczi, E, and D Bentz. 1995. “Modelling of the Microstructure and Transport Properties of Concrete.” *Construction and Building Materials* 10 (5): 293–300.

Gokhale, A M, and S Yang. 1999. “Application of Image Processing for Simulation of Mechanical Response of Multi-Length Scale Microstructures of Engineering Alloys.” *Metallurgical and Material Transactions A* 30 (9): 2369–81.

Green, Brian, Robert Moser, Dylan Scott, and Wendy Long. 2014. “Ultra-High Performance Concrete History and Usage by the United States Army Engineer Research and Development Center.” *Advances in Civil Engineering Materials* 4 (2): 132–43.

Habel, K, M Viviani, E Denarie, and E Bruhwiler. 2006. “Development of the Mechanical Properties of an Ultra-High Performance Fiber Reinforced Concrete (UHPFRC).” *Cement and Concrete Research* 36 (7): 1362–70.

Huang, Y, Z Yang, W Ren, G Liu, and C Zhang. 2015. “3D Meso-Scale Fracture Modelling and Validation of Concrete Based on In-Situ X-Ray Computed Tomography Images Using Damage Plasticity Model.” *International Journal of Solids and Structures* 67–68 (August): 340–52.

Jupe, A C, A P Wilkinson, K Luke, and G P Funkhouser. 2008. “Class H Cement Hydration at 180 Degree C and High Pressure in the Presence of Added Silica.” *Cement and Concrete Research* 38 (5): 660–66.

Kosmatka, S H, and M L Wilson. 2011. *Design and Control of Concrete Mixtures*. 15thed. Skokie, Illinois, USA: Portland Cement Association.

Lopez, Z, I Carol, and A Aguado. 2008. "Meso-Structural Study of Concrete Fracture Using Interface Elements. I: Numerical Model and Tensile Behavior." *Materials and Structures* 41 (3): 583–99.

Lutz, M, P Monteiro, and R Zimmerman. 1997. "Inhomogenous Interfacial Transition Zone Model for the Bulk Modulus of Mortar." *Cement and Concrete Research* 27 (7): 1113–22.

Maalej, M, and V Li. 1994. "Flexural/tensile-Strength Ratio in Engineered Cementitious Composites." *Journal of Materials in Civil Engineering* 6 (4): 513–28.

Moser, Robert, P Allison, and M Chandler. 2013. "Characterization of Impact Damage in Ultra-High Performance Concrete Using Spatially Correlated Nanoindentation/SEM/EDX." *Journal of Materials Engineering and Performance* 22 (12): 3902–8.

Rivera-Soto, Paola, R D Moser, Z B McClelland, B A Williams, and S L Williams. n.d. "Thermal Processing and Alloys Selection to Modify Steel Fiber Performance in Ultra-High Performance Concrete."

Roth, Michael. 2008. "Flexural and Tensile Properties of Thin, Very High-Strength, Fiber-Reinforced Concrete Panels." Engineer Research and Development Center: US Army Corps of Engineers.

Schneider, C A, W S Rasband, and K W Eliceiri. 2012. "NIH Image to ImageJ: 25 Years of Image Analysis." *Nature Methods* 9 (7): 671–75.

Scott, Dylan, Wendy Long, Robert Moser, Brian Green, James O'Daniel, and Brett Williams. 2015. "Impact of Steel Fiber Size and Shape on the Mechanical Properties of Ultra-High Performance Concrete." ERDC/GSL-TR-15-22. Engineer Research and Development Center: Engineer Research and Development Center Geotechnical and Structures Lab.

Smilauer, V, and Z Bittnar. 2006. "Microstructure-Based Micromechanical Prediction of Elastic Properties in Hydrating Cement Paste." *Cement and Concrete Research* 36 (9): 1708–18.

Taylor, H F W. 1997. *Cement Chemistry*. Second.

Unger, J, and S Eckardt. 2011. "Multiscale Modeling of Concrete." *Archives of Computational Methods in Engineering* 18 (3): 341–93.

Vecchio, F J. 1992. "Finite Element Modeling of Concrete Expansion and Confinement." *Journal of Structural Engineering* 118 (9): 2390–2406.

Wang, Xiaofeng, Mingzhong Zhang, and Andrey P Jivkov. 2016. “Computational Technology for Analysis of 3D Meso-Structure Effects on Damage and Failure of Concrete.” *International Journal of Solids and Structures* 80 (February): 310–33.

Williams, Erin, Steven Graham, Paul Reed, and Todd Rusing. 2009. “Laboratory Characterization of Cor-Tuf Concrete With and Without Steel Fibers.” Engineer Research and Development Center: US Army Corps of Engineers.

Zadeh, D, A Bahari, and F Tirandaz. 2008. “Ultra-High Performance Concrete.” In *Excellence in Concrete Construction through Innovation*, 275–78.

Zollo, Ronald. 1997. “Fiber-Reinforced Concrete: An Overview after 30 Years of Development.” *Cement and Concrete Composites* 19 (2): 107–22.

APPENDIX A

RANDOM GEOMETRY GENERATION PYTHON SCRIPT

```
# -----------------------------------------------------------------------------------------------------
---------------------------
# -----------------------------------------------------------------------------------------------------
---------------------------
#       G E O M E T R Y   G E N E S I S
# -----------------------------------------------------------------------------------------------------
---------------------------
# -----------------------------------------------------------------------------------------------------
---------------------------
# For: Dual phase nanocomposites adapted for concrete
# Developed by: W.B. Lawrimore II - wbl59@msstate.edu
# P.I.: M.Q. Chandler, PhD. - mei.q.chandler@erdc.dren.mil
# U.S. Army Engineering Research and Development Center (ERDC)
# -----------------------------------------------------------------------------------------------------
---------------------------
# Version: 2.0 - (12.15.13)

# Imports and intitialization for ABAQUS
import random
import math
import time
import re
import string
from part import *
from material import *
from section import *
from assembly import *
from step import *
from interaction import *
from load import *
from mesh import *
from job import *
from sketch import *
from visualization import *
from connectorBehavior import *
from abaqus import *


# -----------------------------------------------------------------------------------------------------
---------------------------
# -----------------------------------------------------------------------------------------------------
---------------------------
# C L A S S  D E F I N I T I O N S
# -----------------------------------------------------------------------------------------------------
---------------------------
```

```
# ----------------------------------------------------------------------------------------------------
---------------------------

# Class Declaration: Cylinder
# Variables:
#   index - A numbering scheme for the system
#   majAx       - Disk major axis
#   minAx       - Disk minor axis
#   thickness   - Disk thickness
#   orient      - 3D orientation [alpha, beta, gamma]
#   cen         - Center point of disk
#   boundBox    - Hexagonal volume disk occupies
#                       -[minX, maxX]
#                       -[minY, maxY]
#                       -[minZ, maxZ]
#   boundPts    - List of 8 nodes that form bounding box
#   volume      - Volume of the disk

# ----------------------------------------------------------------------------------------------------
---------------------------
# ----------------------------------------------------------------------------------------------------
---------------------------
class Cylinder:
        def __init__(self, ind, data, orient, center, protuberance, inBT):
                # Instance Variables: filler attributes
                # Generated randomly from function below
                self.index = ind
                self.incType = 'cylinder'
                self.mat = data[0]
                self.majAx = data[3]
                self.minAx = data[4]
                self.intFactor = 1
                self.height = data[5]
                self.orient = orient
                self.cen = center

                self.inT = inBT[1]
                self.inB = inBT[0]

                self.boundBox = defBounds(self)
                # From boundaries, aggregate a list of the 8 points making the "hit box"
around each disk
                self.boundPts = []
                for x in range(0,2):
                        for y in range(0,2):
                                for z in range(0,2):
                                        self.boundPts.append([self.boundBox[0][x],
self.boundBox[1][y], self.boundBox[2][z]])
                self.protuberance = protuberance

                self.volume = math.pi*(self.majAx*self.minAx)*self.height*self.intFactor

        # Function: __str__
        #         Purpose: Print a dictionary type description of disk
        #         Returns: String containing disk attributes

        def __str__(self):
                return str(self.__dict__)

        def __setIn__(self,inB,inT):
                self.inB=inB
                self.inT=inT

        # Function: __eq__
        #         Purpose: Is the proposed disk exactly the same as this one?
        #         Returns: Boolean value: True if disks are equal False if they are not

        def __eq__(self, other):
                return self.__dict__==other.__dict__

# ----------------------------------------------------------------------------------------------------
---------------------------
# ----------------------------------------------------------------------------------------------------
---------------------------



# Class Declaration: Sphere
# Variables:
#   index - A numbering scheme for the system
#   radius       - Cylinder Radius
#   cen          - Center point of cylinder
#   boundBox  - Hexagonal volume disk occupies
#                      -[minX, maxX]
#                      -[minY, maxY]
#                      -[minZ, maxZ]
#   boundPts    - List of 8 nodes that form bounding box
#   volume       - Volume of the cylinder
# -----------------------------------------------------------------------------------------------------
---------------------------
# -----------------------------------------------------------------------------------------------------
---------------------------
class Sphere:
        def __init__(self, ind, data, center, protuberance, inBT):
                # Instance Variables: filler attributes
                # Generated randomly from function below
                self.index = ind
                self.incType = 'sphere'
                self.mat = data[0]
                self.radius = data[3]
                self.cen = center
                self.inT = inBT[1]
                self.inB = inBT[0]

                self.boundBox = defBounds(self)
                # From boundaries, aggregate a list of the 8 points making the "hit box"
around each disk
                self.boundPts = []
                for x in range(0,2):
                        for y in range(0,2):
                                for z in range(0,2):
                                        self.boundPts.append([self.boundBox[0][x],
self.boundBox[1][y], self.boundBox[2][z]])
                self.protuberance = protuberance

                self.volume = math.pi*(self.radius**3)

        # Function: __str__
        #       Purpose: Print a dictionary type description of disk
        #       Returns: String containing disk attributes

        def __str__(self):
                return str(self.__dict__)

        def __setIn__(self,inB,inT):
                self.inB=inB
                self.inT=inT

        # Function: __eq__
        #       Purpose: Is the proposed disk exactly the same as this one?
        #       Returns: Boolean value: True if disks are equal False if they are not

        def __eq__(self, other):
                return self.__dict__==other.__dict__

# ----------------------------------------------------------------------------------------------------
---------------------------
# ----------------------------------------------------------------------------------------------------
---------------------------

def defBounds(inclusion):
# Initial dummy values
        boundX = [1000, -1000]
        boundY = [1000, -1000]
        boundZ = [1000, -1000]

        if inclusion.incType =='cylinder':
                # Using a radial mesh with a density of one node every 2 degrees, find the
bounds in cartesian space for the lower face
                # Account for nanodisk position and orientation in global 3D space
                for theta in drange(0, 2*math.pi, math.pi/180.0):
                        stage1 = [inclusion.majAx * math.cos(theta), inclusion.minAx *
math.sin(theta), 0]
                        stage2 = rot_trans_3D(stage1, inclusion.orient)
                        stage3 = [inclusion.cen[0] + stage2[0], inclusion.cen[1] +
stage2[1], inclusion.cen[2] + stage2[2]]
                        boundX = [min(stage3[0], boundX[0]), max(stage3[0],
boundX[1])]
                        boundY = [min(stage3[1], boundY[0]), max(stage3[1],
boundY[1])]
                        boundZ = [min(stage3[2], boundZ[0]), max(stage3[2], boundZ[1])]

                # Repeat process for upper face of disk.
                offset = inclusion.height
                for theta in drange(0, 2*math.pi, math.pi/180.0):
                        stage1 = [inclusion.majAx * math.cos(theta), inclusion.minAx *
math.sin(theta), offset]
                        stage2 = rot_trans_3D(stage1, inclusion.orient)
                        stage3 = [inclusion.cen[0] + stage2[0], inclusion.cen[1] +
stage2[1], inclusion.cen[2] + stage2[2]]
                        boundX = [min(stage3[0], boundX[0]), max(stage3[0],
boundX[1])]
                        boundY = [min(stage3[1], boundY[0]), max(stage3[1],
boundY[1])]
                        boundZ = [min(stage3[2], boundZ[0]), max(stage3[2], boundZ[1])]

        elif inclusion.incType =='sphere':
```

```
        # Using a radial mesh with a density of one node every 1 degree, find the
bounds in cartesian space for the lower face
        # Account for cylinder position and orientation in global 3D space
        boundX = [-inclusion.radius + inclusion.cen[0],
inclusion.radius+inclusion.cen[0]]
        boundY = [-inclusion.radius + inclusion.cen[1],
inclusion.radius+inclusion.cen[1]]
        boundZ = [-inclusion.radius + inclusion.cen[2],
inclusion.radius+inclusion.cen[2]]

    return [boundX, boundY, boundZ]

# -----------------------------------------------------------------------------------------------------
---------------------------
# -----------------------------------------------------------------------------------------------------
---------------------------


def makeNewInclusion(RVE_Dim, data, index):

    # Generate random location
    xCoord = random.uniform(0, 0.95*RVE_Dim)
    yCoord = random.uniform(0, 0.95*RVE_Dim)
    zCoord = random.uniform(0, 0.95*RVE_Dim)+0.5*data[5]

    protuberance=[]
    for i in range(3):
        protuberance.append([0,0])


    if data[1] == 'cylinder':
        alpha = random.uniform(-data[6], data[6])
        beta = random.uniform(-data[7], data[7])
        gamma = random.uniform(-data[8], data[8])

        newInc = Cylinder(index, data, [alpha, beta, gamma], [xCoord, yCoord,
zCoord], protuberance, [-1,-1])

    elif data[1] =='sphere':
        data_copy = data[:]
        if data_copy[6] == 1:
            rad = random.normalvariate(data[3],data[4])
        elif data_copy[6] == 2:
            safety = 1000
            trys = 0
				rad = 0
				while rad < data[7] and trys < safety:
					trys += 1
					rad = random.lognormvariate(data[3],data[4])
				if rad < data[7]:
					raise Exception("Try a smaller minimum size, or a larger
mean radius")
			data_copy[3] = rad
			newInc = Sphere(index, data_copy, [xCoord, yCoord, zCoord],
protuberance, [-1,-1])


	# Determine protuberance
	for i in range(len(newInc.boundBox)):
		for j in range(len(newInc.boundBox[i])):
			if (newInc.boundBox[i][j] < 0.0 or newInc.boundBox[i][j] >
RVE_Dim):
				protuberance[i][j] = 1
	newInc.protuberance = protuberance
	return newInc


# ---------------------------------------------------------------------------------------------------
---------------------------
# ---------------------------------------------------------------------------------------------------
---------------------------

# Function: verifyInclusion
#	Purpose: Determine whether a new inclusion can be placed in matrix without
intersecting other inclusions or lying outside boundaries
#	Returns: Boolean value: True if the new inclusion is unique, completely within
matrix boundaries, has no intersections with other
#			inclusions, and is well spaced from other inclusions, False if any one of
those are not true
#
# ---------------------------------------------------------------------------------------------------
---------------------------
# ---------------------------------------------------------------------------------------------------
---------------------------

def verifyInclusion(currInc, RVE_Dim, masterList, maxRadius):
	# Booleans to denote whether a inlclusion satisfies all requirements.
	noIntersect = 1
	"""
	if currInc.incType == 'cylinder':
```

```
            #criteria = 2*max(2*currInc.majAx, 2*currInc.minAx, currInc.height)
            criteria = RVE_Dim
        elif currInc.incType == 'sphere':
            criteria = 2*max(maxRadius, currInc.radius)
        """

        collisionList = []
        if len(masterList) > 0:
            for entry in masterList:
                #if distance(currInc.cen, entry[0].cen) <= criteria:
                    collisionList.append(entry[0])

        i = 0
        #for i in range(0, len(regions)):
        testIncInd = 0
        #while noIntersect and testIncInd < len(regionList[regions[i]]):
        while noIntersect and testIncInd < len(collisionList):
            testInc = collisionList[testIncInd]
            collided = collisionDetect(currInc, testInc)
            if collided:
                noIntersect = 0
            testIncInd += 1

        return noIntersect

# ----------------------------------------------------------------------------------------------------
---------------------------
# ----------------------------------------------------------------------------------------------------
---------------------------

def collisionDetect(inc1, inc2):
        # Boolean Value
        collisionDetected = 0
        # Intersection vector

        if inc1.incType =='sphere' and inc2.incType =='sphere':
            if distance(inc1.cen, inc2.cen) <= (inc1.radius+inc2.radius):
                collisionDetected = 1
        else:
            xyzIn = [0,0,0]
            offset = 0
            # Test each bound edge for coincidence with the other inclusion's box
            for i in range(0, len(inc1.boundBox)):
                if (inc1.boundBox[i][0] >= inc2.boundBox[i][0]-offset) and
(inc1.boundBox[i][0] <= inc2.boundBox[i][1]+offset):
```

```
                    xyzIn[i] = 1
                elif inc1.boundBox[i][1] >= inc2.boundBox[i][0]-offset and
inc1.boundBox[i][1] <= inc2.boundBox[i][1]+offset:
                    xyzIn[i] = 1
            for i in range(0, len(inc2.boundBox)):
                if (inc2.boundBox[i][0] >= inc1.boundBox[i][0]-offset) and
(inc2.boundBox[i][0] <= inc1.boundBox[i][1]+offset):
                    xyzIn[i] = 1
                elif inc2.boundBox[i][1] >= inc1.boundBox[i][0]-offset and
inc2.boundBox[i][1] <= inc1.boundBox[i][1]+offset:
                    xyzIn[i] = 1
            #For intersections xyzIn = [1,1,1]
            if sum(xyzIn) == 3:
                collisionDetected = 1

    return collisionDetected
# -----------------------------------------------------------------------------------------------------
---------------------------
# -----------------------------------------------------------------------------------------------------
---------------------------

def generateCopies(newInc, RVE_Dim, index, maxRadius):
    goodCopies = 1
    axis = 0
    copies = []
    finalCopy = [0,0,0]
    currIndex = index
    for axis in range(len(newInc.protuberance)):
        alteredCoords=[]
        if newInc.protuberance[axis][0] ==1.0:
            finalCopy[axis] = 1.0
        elif newInc.protuberance[axis][1] == 1.0:
            finalCopy[axis] = -1.0
        if finalCopy[axis] != 0:
            currIndex = currIndex + 1
            for x in newInc.cen:
                alteredCoords.append(x)
            alteredCoords[axis] = alteredCoords[axis] +
finalCopy[axis]*RVE_Dim
            currCopy = copyInclusion(newInc, alteredCoords, currIndex,
RVE_Dim)
            if not currCopy[1]:
                goodCopies = 0
                #print 'MMT %i too small 1\n' %currIndex
            else:
                copies.append([currCopy[0],[0]])
    if len(copies) > 1:
        currIndex = currIndex + 1
        alteredCoords = []
        for x in newInc.cen:
                alteredCoords.append(x)
        for x in range(len(finalCopy)):
            alteredCoords[x] = alteredCoords[x] + finalCopy[x]*RVE_Dim
        currCopy = copyInclusion(newInc, alteredCoords, currIndex, RVE_Dim)
        if not currCopy[1]:
                goodCopies = 0
                #print 'MMT %i too small 2\n' %currIndex
                currIndex = currIndex-1
        else:
            copies.append([currCopy[0],[0]])
    copyIndex = 0
    while goodCopies and copyIndex < len(copies):
        verifyCopy = verifyInclusion(copies[copyIndex][0], RVE_Dim,
masterList, maxRadius)
        if not verifyCopy:
            goodCopies = 0
        else: copies[copyIndex][1] = verifyCopy
        copyIndex = copyIndex + 1
    return [goodCopies, copies, currIndex]

# -------------------------------------------------------------------------------------------------
-------------------------
# -------------------------------------------------------------------------------------------------
-------------------------

def copyInclusion(currInc, coords, index, RVE_Dim):
    tooSmall = 0
    if currInc.incType == 'cylinder':
        incData = [currInc.mat, currInc.incType,0, currInc.majAx, currInc.minAx,
currInc.height]
        newInc = Cylinder(index, incData, currInc.orient, coords,
[[0,0],[0,0],[0,0]],[-5, -5])
        contained= RVEcontainer(newInc,RVE_Dim)
    elif currInc.incType == 'sphere':
        incData = [currInc.mat, currInc.incType,0, currInc.radius, 0, 0]
        newInc = Sphere(index, incData, coords, [[0,0],[0,0],[0,0]],[-5, -5])
        contained= RVEcontainer(newInc,RVE_Dim)
    return [newInc, contained]
```

```
# ----------------------------------------------------------------------------------------------------
---------------------------
# ----------------------------------------------------------------------------------------------------
---------------------------


def RVEcontainer(inclusion, RVE_Dim):
        incount_B = 0
        incount_T = 0
        incount_XY = 0
        incount_XZ = 0
        incount_YZ = 0
        contained = 1
        if inclusion.incType == 'cylinder':
                theta = 0
                while theta < 2*math.pi:
                        stage1 = [inclusion.majAx * math.cos(theta), inclusion.minAx *
math.sin(theta), 0]
                        stage2 = rot_trans_3D(stage1, inclusion.orient)
                        stage3 = [inclusion.cen[0] + stage2[0], inclusion.cen[1] +
stage2[1], inclusion.cen[2] + stage2[2]]
                        if (stage3[0] > 0) and (stage3[0] < RVE_Dim):
                                if (stage3[1] > 0) and (stage3[1] < RVE_Dim):
                                        if (stage3[2] > 0) and (stage3[2] < RVE_Dim):
                                                incount_B += 1
                        theta = theta + math.pi/180
                theta = 0
                while theta < 2*math.pi:
                        stage1 = [inclusion.majAx * math.cos(theta), inclusion.minAx *
math.sin(theta), (13+(inclusion.intFactor-1)*4)]
                        stage2 = rot_trans_3D(stage1, inclusion.orient)
                        stage3 = [inclusion.cen[0] + stage2[0], inclusion.cen[1] +
stage2[1], inclusion.cen[2] + stage2[2]]
                        if (stage3[0] > 0) and (stage3[0] < RVE_Dim):
                                if (stage3[1] > 0) and (stage3[1] < RVE_Dim):
                                        if (stage3[2] > 0) and (stage3[2] < RVE_Dim):
                                                incount_T += 1
                        theta = theta + math.pi/180
                if incount_B < 180 and incount_T <180:
                        contained = 0
                inclusion.inB = incount_B
                inclusion.inT = incount_T

        elif inclusion.incType == 'sphere':
                theta = 0
        while theta < 2*math.pi:
            stage1 = [inclusion.radius*math.cos(theta), 0,
inclusion.radius*math.sin(theta)]
            stage2 = [inclusion.cen[0] + stage1[0], inclusion.cen[1]+stage1[1],
inclusion.cen[2]+stage1[2]]

            if (stage2[0] > 0) and (stage2[0] < RVE_Dim):
                if (stage2[1] > 0) and (stage2[1] < RVE_Dim):
                    if (stage2[2] > 0) and (stage2[2] < RVE_Dim):
                        incount_XZ += 1
            theta = theta + math.pi/180
        theta = 0
        while theta < 2*math.pi:
            stage1 = [0,inclusion.radius*math.cos(theta),
inclusion.radius*math.sin(theta)]
            stage2 = [inclusion.cen[0] + stage1[0], inclusion.cen[1]+stage1[1],
inclusion.cen[2]+stage1[2]]
            if (stage2[0] > 0) and (stage2[0] < RVE_Dim):
                if (stage2[1] > 0) and (stage2[1] < RVE_Dim):
                    if (stage2[2] > 0) and (stage2[2] < RVE_Dim):
                        incount_YZ += 1
            theta = theta + math.pi/180
        theta = 0
        while theta < 2*math.pi:
            stage1 = [inclusion.radius*math.cos(theta),
inclusion.radius*math.sin(theta),0]
            stage2 = [inclusion.cen[0] + stage1[0], inclusion.cen[1]+stage1[1],
inclusion.cen[2]+stage1[2]]
            if (stage2[0] > 0) and (stage2[0] < RVE_Dim):
                if (stage2[1] > 0) and (stage2[1] < RVE_Dim):
                    if (stage2[2] > 0) and (stage2[2] < RVE_Dim):
                        incount_XY += 1
            theta = theta + math.pi/180
        if incount_XZ < 50 and incount_YZ<50 and incount_XY<50:
            contained = 0
    return contained

# ---------------------------------------------------------------------------------------------------
---------------------------
# ---------------------------------------------------------------------------------------------------
---------------------------


# Function: buildGeometry
```

```
#	Purpose: Take the array of generated inclusions and issue commands to ABAQUS
CAE to create the geometry, mesh, and periodic boundary
#		conditions of the UHPC composite
#	Returns: None

# -----------------------------------------------------------------------------------------------------
---------------------------
# -----------------------------------------------------------------------------------------------------
---------------------------

def buildGeometry(RVE_Dim, masterList, modelName, matName, incMatList):
	indicies = []
	newInd = []
	# Tell abaqus to use detailed commands
	session.journalOptions.setValues(replayGeometry=COORDINATE,
recoverGeometry=COORDINATE)
	# Generate new model with custom name
	# Delete old model
	mdb.Model(name=modelName)
	del mdb.models['Model-1']
	model = mdb.models[modelName]

	rve = 'RVE_Box'
	# Create matrix
	# Cube with side length imported from input file
	model.ConstrainedSketch(name=rve, sheetSize=RVE_Dim)
	model.sketches[rve].rectangle(point1=(0,0), point2=(RVE_Dim, RVE_Dim))
	model.Part(dimensionality=THREE_D, name=rve,
type=DEFORMABLE_BODY)
	model.parts[rve].BaseSolidExtrude(depth=RVE_Dim,
sketch=model.sketches[rve])
	model.parts[rve].BaseShell(sketch=model.sketches[rve])
	del model.sketches[rve]

	model.Material(name=matName)
	model.HomogeneousSolidSection(material=matName, name='Matrix',
thickness=None)
	model.parts[rve].SectionAssignment(offset=0.0, offsetField='',
offsetType=MIDDLE_SURFACE,
region=Region(cells=model.parts[rve].cells.findAt((((0,0,0), ), )), sectionName='Matrix',
thicknessAssignment=FROM_SECTION)

	# Create a material and section for each inclusion material
	for mat in incMatList.keys():
		model.Material(name=mat)
```

```
                model.HomogeneousSolidSection(material=mat, name=mat,
thickness=None)


        # Establish an assembly
        # Import matrix
        model.rootAssembly.DatumCsysByDefault(CARTESIAN)
        model.rootAssembly.Instance(dependent=ON, name=rve, part=model.parts[rve])

        # Define lists for inclusion names and their instance objects
        cNames = []
        instList = []
        voidList = []
        ENames = []
        prtList = []

        assembly = model.rootAssembly


        # Add inclusions into assembly
        for inc in masterList:
                incMatList[inc[0].mat].append(inc[0])
                #incMatNum = len(incMatList[inc[0].mat])
                incMatNum = inc[0].index
                incName = inc[0].mat + '_' + str(incMatNum)
                dbf.write('Inc %i: %4.4f  %4.4f  %4.4f\n' %(inc[0].index, inc[0].cen[0],
inc[0].cen[1], inc[0].cen[2]))

                # Create part
                model.ConstrainedSketch(name=incName, sheetSize=RVE_Dim)

                if inc[0].incType == 'cylinder':

        model.sketches[incName].EllipseByCenterPerimeter(axisPoint1=(inc[0].majAx,
0.0), axisPoint2=(0.0, inc[0].minAx), center=(0.0, 0.0))
                        model.Part(dimensionality=THREE_D, name=incName,
type=DEFORMABLE_BODY)
                        model.parts[incName].BaseSolidExtrude(depth=inc[0].height,
sketch=model.sketches[incName])
                        model.parts[incName].SectionAssignment(offset=0.0,
offsetField='', offsetType=MIDDLE_SURFACE,
region=Region(cells=model.parts[incName].cells.findAt(((0,0,0), ), )),
sectionName=inc[0].mat, thicknessAssignment=FROM_SECTION)
```

```
                              model.parts[incName].Surface(name='Front',
side1Faces=model.parts[incName].faces.findAt(((0, 0, inc[0].height), ),
((inc[0].majAx/2.0, inc[0].minAx/2.0, inc[0].height), ), ))
                              model.parts[incName].Surface(name='Back',
side1Faces=model.parts[incName].faces.findAt(((0.0,0.0, 0.0), ), ((inc[0].majAx/2.0,
inc[0].minAx/2.0, 0.0), ), ))
                              model.parts[incName].Surface(name='Edge',
side1Faces=model.parts[incName].faces.findAt(((inc[0].majAx, 0.0, inc[0].height/2.0), ),
((0.0, inc[0].minAx, inc[0].height/2.0), ), ((-inc[0].majAx, 0.0, inc[0].height/2.0), ), ))
                              for key in incMatList.keys():
                                        results = key.find(inc[0].mat)
                                        if results != -1:

          model.parts[incName].SectionAssignment(offset=0.0, offsetField='',
offsetType=MIDDLE_SURFACE,
region=Region(cells=model.parts[rve].cells.findAt(((inc[0].cen), ), )), sectionName=key,
thicknessAssignment=FROM_SECTION)
                    elif inc[0].incType == 'sphere':
                              model.sketches[incName].ConstructionLine(point1=(0.0, -
RVE_Dim/2.0), point2=(0.0, RVE_Dim/2.0))
                              model.sketches[incName].geometry.findAt((0.0, 0.0))

          model.sketches[incName].FixedConstraint(entity=model.sketches[incName].geo
metry.findAt((0.0, 0.0), ))
                              model.sketches[incName].ArcByCenterEnds(center=(0.0, 0.0),
direction=CLOCKWISE, point1=(0.0, inc[0].radius), point2=(0.0, -inc[0].radius))
                              model.sketches[incName].Line(point1=(0.0, inc[0].radius),
point2=(0.0, -inc[0].radius))
                              model.sketches[incName].geometry.findAt((0.0, -inc[0].radius))
                              model.sketches[incName].VerticalConstraint(addUndoState=False,
entity=model.sketches[incName].geometry.findAt((0.0, -inc[0].radius), ))
                              model.sketches[incName].geometry.findAt((inc[0].radius, 0.0))
                              model.sketches[incName].geometry.findAt((0.0, -inc[0].radius))

          model.sketches[incName].PerpendicularConstraint(addUndoState=False,
entity1=model.sketches[incName].geometry.findAt((inc[0].radius, 0.0), ),
entity2=model.sketches[incName].geometry.findAt((0.0, -inc[0].radius), ))
                              model.Part(dimensionality=THREE_D, name=incName,
type=DEFORMABLE_BODY)
                              model.parts[incName].BaseSolidRevolve(angle=360.0,
flipRevolveDirection=OFF, sketch=model.sketches[incName])
                              model.parts[incName].Surface(name='Surf',
side1Faces=model.parts[incName].faces.findAt(((inc[0].radius, 0, 0), )))
                              for key in incMatList.keys():
                                        results = key.find(inc[0].mat)
                    if results != -1:

        model.parts[incName].SectionAssignment(offset=0.0, offsetField='',
offsetType=MIDDLE_SURFACE,
region=Region(cells=model.parts[rve].cells.findAt(((inc[0].cen), ), )), sectionName=key,
thicknessAssignment=FROM_SECTION)

                del model.sketches[incName]

                assembly.Instance(dependent=OFF, name=incName,
part=model.parts[incName])
                # Rotate and translate inclusion in 3D space
                if inc[0].incType == 'cylinder':
                        assembly.rotate(angle=inc[0].orient[1], axisDirection=(0.0, 1.0,
0.0), axisPoint=(0.0, 0.0, 0.0), instanceList=(incName,))
                        assembly.rotate(angle=inc[0].orient[2], axisDirection=(0.0, 0.0,
1.0), axisPoint=(0.0, 0.0, 0.0), instanceList=(incName,))
                        assembly.rotate(angle=inc[0].orient[0], axisDirection=(1.0, 0.0,
0.0), axisPoint=(0.0, 0.0, 0.0), instanceList=(incName,))
                assembly.translate(instanceList=(incName,), vector=(inc[0].cen))

                if inc[1]:
                        if inc[0].mat != 'Void':
                                try:

        assembly.InstanceFromBooleanCut(cuttingInstances=(assembly.instances[rve], ),
instanceToBeCut=assembly.instances[incName], name=incName + 'T',
originalInstances=SUPPRESS)
                                except (AbaqusException), value:
                                        print "CUT ERROR1"
                                        print "Inclusion %d    inT: %d    inB:%d\n"
%(incMatNum, inc[0].inT, inc[0].inB)
                                        print "Cen: %f, %f, %f"
%(inc[0].cen[0],inc[0].cen[1],inc[0].cen[2])
                                assembly.features[incName].resume()
                                assembly.features[rve].resume()
                                try:

        assembly.InstanceFromBooleanCut(cuttingInstances=(assembly.instances[incNa
me + 'T-1'], ), instanceToBeCut=assembly.instances[incName], name=incName+ '_C',
originalInstances=DELETE)
                                except (AbaqusException), value:
                                        print "CUT ERROR2"
                                        print "Inclusion %d    inT: %d    inB:%d\n"
%(incMatNum, inc[0].inT, inc[0].inB)
```

```
                              print "Cen: %f, %f, %f"
%(inc[0].cen[0],inc[0].cen[1],inc[0].cen[2])
                              print 'Protuberance'
                              print inc[0].protuberance
                         assembly.features.changeKey(fromName=incName+ '_C' +
'-1', toName=incName)
                         del model.parts[incName + 'T']

        assembly.makeIndependent(instances=(assembly.instances[incName], ))
                         del model.parts[incName]
                         model.parts.changeKey(fromName=incName+ '_C',
toName=incName)

             if inc[0].mat == 'Void':
                  voidList.append(assembly.instances[incName])
             else:
                  cNames.append([incName, inc[0].incType])
                  instList.append(assembly.instances[incName])
             # Progress report
             if inc[0].index%100 == 0 and inc.index > 0:
                  print("Inclusions Inserted " + str(inc[0].index))



        print("Carving voids in matrix for inclusions to occupy")
        if len(voidList) > 0:
             assembly.InstanceFromBooleanCut(cuttingInstances=(voidList),
instanceToBeCut=assembly.instances[rve], name=rve, originalInstances=DELETE)
             # After all inclusions added, cut out overlapping sections of the matrix
             assembly.InstanceFromBooleanCut(cuttingInstances=(instList),
instanceToBeCut=assembly.instances[rve+'-1'], name=matName,
originalInstances=SUPPRESS)
             del assembly.features[rve+'-1']
        else:
             assembly.InstanceFromBooleanCut(cuttingInstances=(instList),
instanceToBeCut=assembly.instances[rve], name=matName,
originalInstances=SUPPRESS)

        assembly.resumeFeatures((col(cNames,0)))

        assembly.features.changeKey(fromName=matName + '-1',
toName=matName+'_Box')
        assembly.makeIndependent(instances=(assembly.instances[matName+'_Box'], ))
        # Tie Coplanar faces together
        i = 0
        for i in range(len(cNames)):
                names = col(cNames, 0)
                types = col(cNames,1)
                if types[i] == 'cylinder':
                        surfs = ['Front', 'Back','Edge']
                elif types[i] =='sphere':
                        surfs = ['Surf']
                for s in surfs:
                        j=0
                        found = 0
                        minD = RVE_Dim
                        for face in assembly.instances[names[i]].surfaces[s].faces:
                                bCent= face.getCentroid()[0]
                                while not found and j <
len(assembly.instances[matName+'_Box'].faces):
                                        matFace =
assembly.instances[matName+'_Box'].faces[j]
                                        matCent = matFace.getCentroid()[0]
                                        if distance(bCent, matCent) < minD:
                                                minD = distance(bCent, matCent)
                                                minFace = matFace
                                                minFPoint = matFace.pointOn[0]
                                        j = j+1
                        surfName = str(matName +'_' +names[i])
                        assembly.Surface(name=surfName,
side1Faces=assembly.instances[matName+'_Box'].faces.findAt(((minFPoint), )))
                        model.Tie(adjust=ON, master=assembly.surfaces[surfName],
name=names[i]+'_'+str(s) , positionToleranceMethod=COMPUTED,
slave=assembly.instances[names[i]].surfaces[s], thickness=ON, tieRotations=ON)


# Funtion: drange
#       Purpose: Define a range of values with a non-default, prescribed step.
def drange(start, stop, step):
        r = start
        while r < stop:
                yield r
                r += step

# Funtion: distance
#       Purpose: Calculate the 2D cartesian distance between 2 points.
```

```
def distance(point1,point2):
        return math.sqrt((point2[0]-point1[0])**2 + (point2[1]-point1[1])**2 +
(point2[2]-point1[2])**2)

# Funtion: rot_trans_3D
#       Purpose: Performs vector rotation transformation on a given point by a given
angle.
def rot_trans_3D(point, orient):
        radAlpha = math.radians(orient[0])
        radBeta = math.radians(orient[1])
        radGamma = math.radians(orient[2])
        # Equivalent to the multiplication of the three rotation matricies
        # Order [Rot abt Y-axis] * [Rot abt Z-axis] * [Rot abt X-axis]
        xPrime = math.cos(radGamma)*math.cos(radBeta)*point[0] -
math.sin(radGamma)*point[1] + math.cos(radGamma)*math.sin(radBeta)*point[2]
        yPrime =
(math.cos(radAlpha)*math.sin(radGamma)*math.cos(radBeta)+math.sin(radAlpha)*mat
h.sin(radBeta))*point[0] +
math.cos(radAlpha)*math.cos(radGamma)*point[1]+(math.cos(radAlpha)*math.sin(radG
amma)*math.sin(radBeta)-math.sin(radAlpha)*math.cos(radBeta))*point[2]
        zPrime = (math.sin(radAlpha)*math.sin(radGamma)*math.cos(radBeta)-
math.cos(radAlpha)*math.sin(radBeta))*point[0] +
math.sin(radAlpha)*math.cos(radGamma)*point[1]+(math.sin(radAlpha)*math.sin(radG
amma)*math.sin(radBeta)+math.cos(radAlpha)*math.cos(radBeta))*point[2]
        return [xPrime, yPrime, zPrime]

def col(matrix, n):
        return [column[n] for column in matrix]


# ----------------------------------------------------------------------------------------------------
--------------------------
# ----------------------------------------------------------------------------------------------------
---------------------------


fields = (('Model Name:','Cor-Tuf'), ('Matrix Material:','Cement Paste'), ('RVE
Dimension:','10'), ('Number of Incusion Types:','1'))

modelName, matrixName, RVE_Dim, Num_Inc_types = getInputs(fields=fields,
label='Input Parameters', dialogTitle='Composite Geometry Genesis')

RVE_Dim = float(RVE_Dim)
Num_Inc_types = int(Num_Inc_types)
incData= []
```

```
matList={}

for k in range(Num_Inc_types):
	newMat = 1
	Incfields = (('Material Name:', 'Void'),('Geometry (1-Cylinder, 2-Sphere)',
'2'),('Volume Fraction (%):', '1'),('Geometry Distribution Function (1-Gaussian, 2-
Lognormal)','2'))
	name, geom, vfrac, gdist = getInputs(fields=Incfields, label='Inclusion Type
'+str(k+1), dialogTitle='Inclusion Type '+str(k+1))
	geom = float(geom)
	gdist = int(gdist)

	for mat in matList.keys():
		if mat == name:
			newMat = 0
	if newMat:
		matList[name]=[]

	if geom == 1:
		gType = 'cylinder'
		cylFields = (('Major Radius:','0.08'),('Minor Radius:','0.08'),('Height:','6'),
('Maximum Alpha (deg):','360'), ('Maximum Beta (deg):','360'), ('Maximum Gamma
(deg):','360'))
		majR, minR, height, alpha, beta,gamma = getInputs(fields=cylFields,
label=name+' Parameters', dialogTitle=name+ 'Parameters')
		incData.append([name, gType,  float(vfrac), float(majR), float(minR),
float(height), float(alpha), float(beta), float(gamma)])
	elif geom == 2:
		gType = 'sphere'
		if gdist == 1:
			sphFields = (('Mean Radius:','0.5'),('Min Radius','0.005'),('Standard
Deviation:','0.001'))
		elif gdist == 2:
			sphFields = (('Mean Radius:','0.5'),('Min Radius','0.005'),('Log
Standard Deviation:','1.0'))
		rad, rmin, rstd = getInputs(fields=sphFields, label=name+' Parameters',
dialogTitle=name+ ' Parameters')
		if gdist == 1:
			rad = float(rad)
			rstd = float(rstd)
			rmin = float(rmin)
		elif gdist == 2:
			m = float(rad)
			rad = log( m ) - float(rstd[0])**2 / 2
			rstd = float(rstd)
```

```
                rmin = float(rmin)
            incData.append([name, gType, float(vfrac),  rad, rstd, 0, gdist,rmin,0])



trys = 0
masterList = []
index = 0
num = 0
maxRadius = 0

safety = 500000

dbf = open('debug.txt' ,'w')

for inc in incData:
    currFrac = 0
    currVol = 0
    while currFrac < inc[2] and trys <= safety:
        newInc = makeNewInclusion(RVE_Dim, inc, index)
        verified = verifyInclusion(newInc, RVE_Dim, masterList, maxRadius)
        edgeFlag = 0
        if verified:
            verified2 = [1,0,0]
            for ax in range(len(newInc.protuberance)):
                for lvl in newInc.protuberance[ax]:
                    if lvl == 1:
                        edgeFlag= 1
            if edgeFlag:
                verified2 = generateCopies(newInc, RVE_Dim, index,
maxRadius)
            if verified2[0]:
                num+=1
                currVol = currVol + newInc.volume
                masterList.append([newInc, edgeFlag])
                if newInc.incType == 'sphere':
                    if newInc.radius > maxRadius:
                        maxRadius = newInc.radius

                if verified2[1]:
                    for copy in verified2[1]:
                        masterList.append([copy[0], edgeFlag])
                if verified2[2]:
                    index = verified2[2]+1
                else: index = index + 1
                currFrac = float(currVol/RVE_Dim**3)*100
                if trys%100 == 0 and trys > 0:
                        print("Progress after %d trys." %trys)
                        print(inc[0] + " Inclusions: %d" %num)
                        print(inc[0] + " Volume fraction: " + str(currFrac) + "%")
                trys = trys+1

exit=0
if trys >= safety:
        decision = getWarningReply('Safety reached before deisred volume fraction.\n
Generate geometry anyway?', (YES,NO))
        if str(decision) =='NO':
                exit = 1

if not exit:

        print("Inserting inclusion collection into matrix")
        # After either volume fraction or safety is reached submit geometry to ABAQUS
for construction
        buildGeometry(RVE_Dim, masterList, modelName, matrixName, matList)

        dbf.close()

        # Save the abaqus model
        mdb.saveAs(pathName='CGG_Test.cae')
else:
        print 'Aborting Geometry Genesis'
```