Solid Freeform Fabrication 2021: Proceedings of the 32nd Annual International
Solid Freeform Fabrication Symposium – An Additive Manufacturing Conference
Reviewed Paper

Using Medial Surfaces to Produce Graded Voronoi Cell Infill Structures for 3D Printed Objects

T. Williams, Prof. D. Storti, Prof. M. Ganter
Dept. Mechanical Engineering, University of Washington, Seattle, WA, 98195

Abstract

Many methods of additive manufacturing rely on infill structures to decrease part mass and print time. However, standard infill patterns generally use a uniform density or require time-consuming analysis to generate a density field tailored to part geometry. We propose a Voronoi cell based infill structure which uses the medial surfaces of the object to locate thin regions and increase local material density. The Voronoi cell structure reduces transition points within the infill, producing a more even gradient in density, while the weighting scheme ensures that traditionally weaker portions of the model receive adequate internal support.

Introduction

When an object is created with a 3D printer, there are many decisions beyond just the external geometry that must be made in accordance with the requirements of the application. Variables such as layer thickness, infill density, infill pattern, print material, and print temperature can have an influence on the mechanical properties of the part and the amount of time and material it takes to produce it [1-2]. The infill density and infill pattern variables are an active field of study within the additive manufacturing community [1-6]. To summarize, the infill of a 3D printed component can be described as a structure within the model that allows the interior of the component to remain partially hollow, and is often used to help hold the shell in place. For material extrusion processes, the infill structure used allows the model to be printed considerably faster with lower material use [2].

As the field of additive manufacturing has matured, infill structures have become more and more functional. Many of the commercially available infill generation technologies are geometry-agnostic and have few control variables, such as the infill pattern itself (grid, hexagon, cat, etc.), the scale, and the orientation [1-2]. Some infill generation algorithms have been published that allow for a manually weighted infill structure, with higher and lower densities chosen based on user input, desired mechanical properties, or the geometry of the part [3-5].

Weighted infill structures are an active scientific field, as they allow for a high degree of control over the mechanical properties of the printed components [3-5]. With a weighted infill generation algorithm, additional infill material can be added at locations of stress concentration or on weaker portions of the model, while lower density of infill can be used elsewhere to save on material usage, part mass, and print time. However, many of the weighted schemes that are currently available require a trained user's input to implement them successfully. Weighted infill structures that are based on stress concentrations require that the model is analyzed via finite element analysis (FEA), which is often computationally expensive and depends on the user to implement the correct loading parameters.

This paper proposes an alternative approach to weighted infill that reinforces thinner areas of the object while maintaining lower infill density in thicker, bulkier areas. Although it can be computationally expensive, it can be done without access to an external FEA package and maps the model's density without requiring user input.

The suggested medial-surface-based approach to generating infill structures requires the usage of several geometric constructs: Signed Distance Fields (SDFs), Medial Surfaces, and Voronoi Diagrams.

An SDF is a mathematical field where every point in space is assigned a value to denote its distance to the nearest surface of an object [7]. If the point in space is inside of the object, the value is negative. If the point is outside the object, the value is positive. If the point is directly on the surface of the object, then the value is zero. Mathematically, this field can be continuous and defined by a single equation, but such equations tend to be cumbersome to produce and unwieldy to use. For the purposes of this paper, the SDF will be discretely sampled in space and stored in a 3D array. The SDF of a rectangle (Side lengths 240 by 160 voxels) is shown in Figure 1.

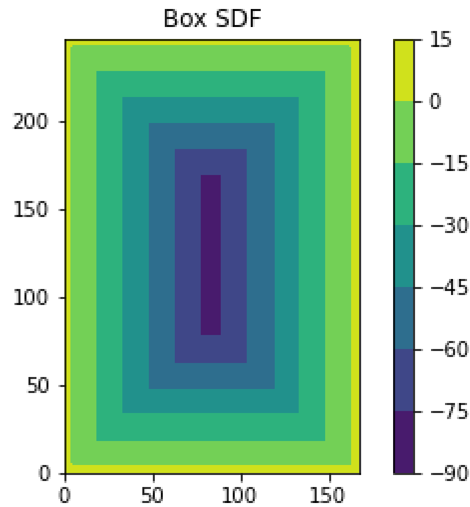

Figure 1: SDF of a rectangle

The medial surface of an object can be described in multiple ways, with one of the most elegant being that the medial surface is the locus of the centers of all the maximal spheres of an object [8-15]. As such, the medial surface lies directly in the center of an object. Every point on a medial surface is the centerpoint of a maximal sphere, meaning that the sphere centered on that point must be touching at least two points on the surface of the object without passing through the object's surface or being completely enclosed by any other maximal sphere. The medial surface, when combined with the radii of the maximal spheres that originate from each point on the medial surface, can be used to recreate the object in a process called a medial surface transform [14]. The medial surface of the rectangle from Figure 1 is shown in Figure 2.

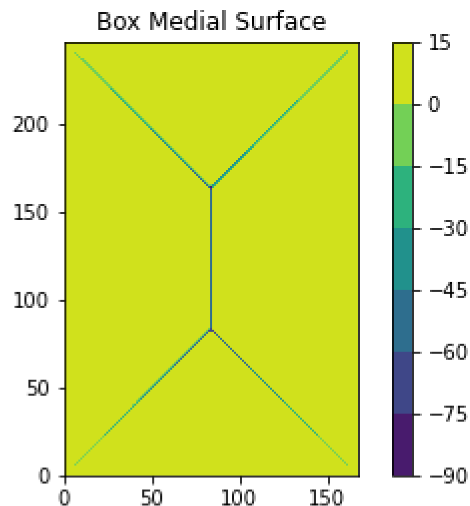

Figure 2: Medial Surface of a rectangle, where the color scale represents the corresponding radial data. Positive values are outside of the medial surface.

Voronoi diagrams are an interesting construct as they mimic geometry that often shows up in nature. A Voronoi diagram is based on a set of seed points scattered across a 2D plane or in a 3D space [16-19]. For each point, a 'cell' is grown such that any point within the cell is closer to its own seed point than any other seed point. For the purposes of this paper, the walls of these Voronoi Cells provide an excellent structure for 3D printed infill, as they can be generated relatively quickly and the material density is linked to the volumetric density of the seed points. Figure 3 shows an example of these structures.

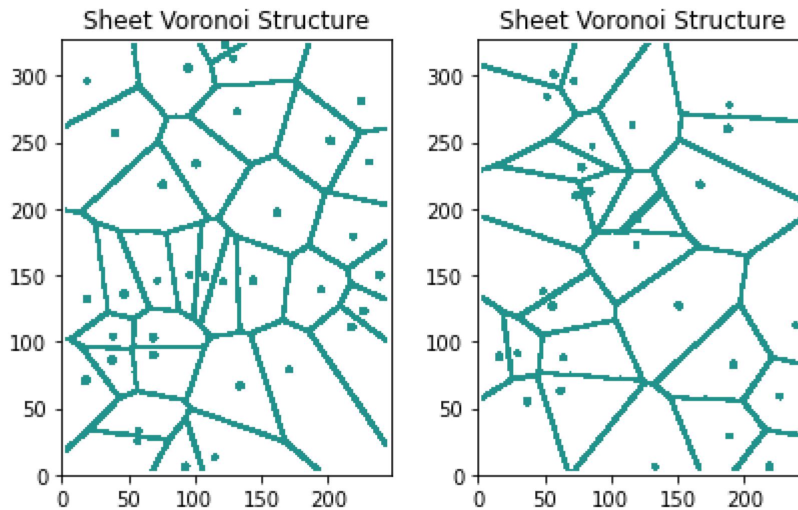

Figure 3: Slice of Voronoi cell structure generated within a 2D rectangle with seed points shown. Left image has 35 points, the right has 28. These are generated via an unweighted random distribution.

One of the advantages of Voronoi cells is that the seed point density field can be determined by any number of methods. Previous studies have used unweighted point density fields, have based the point density on distance from the surface of the object [17], or have manually modified the point density field to have more direct control over the resulting infill's geometry and its mechanical properties [3-5].

Proposed Alternative

The infill style that we propose would alter the weighting scheme for the distribution of the Voronoi seed points by automatically increasing the chances that seed points would be generated in thinner areas of the model. Point density distribution would be done through the usage of the medial surfaces of the object, which run through the central axes of each portion of the model. Under even weighting or surface-distance weighting, proportionally thinner portions of the model often have scattered and disconnected sheets of Voronoi cell infill, meaning that

their mechanical properties are often more reliant on the perimeters of the print than the infill structure. By weighting the point density field to favor areas where the medial surface has lower magnitude radial data, the density of seed points will increase to boost the local strength of the object, ensuring that the thinner portions are properly supported. An example of the results from a distance weighting algorithm is shown on the left of Figure 4, with our comparison model on the right.

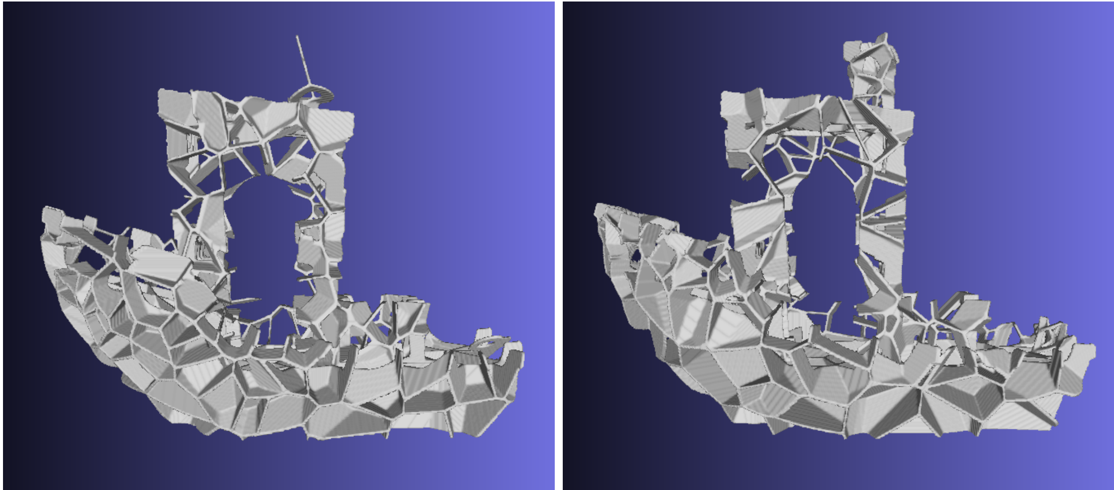

Figure 4: Example of Voronoi cell infill generated for the 3D Benchy model [20] via surface distance weighting methods (left) and medial surface weighting methods (right). Both models have approximately 250 seed points. As can be seen, the distance weighting method has a lower infill density in the smokestack and a higher density at the bow of the ship.

Methods of Implementation

In broad strokes, the algorithm first converts the model into a discrete form (also known as a voxel representation) and stores it in a 3D array. From the discrete array, the algorithm then finds the medial surface. Wavefront propagation is employed to recreate the original voxelized object while ensuring that every voxel within the array retains the size of the largest maximal sphere it resides within [16]. After the radii have been distributed by the wavefront algorithm, a weighting field is then produced by setting every voxel within the object to instead contain the reciprocal of its value, ensuring that voxels in the thinner portions of the object have a larger value, while voxels in the thicker portions of the object have a smaller value. A random number is then generated for each voxel in the array and is compared to the value of the weighting field at those coordinates. If the random number is smaller than the value, then that voxel is set to be a seed point for the Voronoi cells. Once the seed points have been generated, Voronoi cells can be generated and trimmed to remain within the object. The surface of the object is wrapped over the resulting cellular structure and converted back to a surface model via the marching cubes algorithm.

Details on the steps of this process are listed below, and many of the more complex algorithms are also described in the appendix. A complete copy of the code used throughout this paper is also available at the github repository linked at the end of the paper.

Finding the Medial Surface

The medial surface can be described simply, but finding it is often more complex. A method that has proven successful is to first find the SDF of the object, then find the points at which the continuous SDF would be non-differentiable [16]. As we are instead working in discrete space, we will instead locate the points at which the magnitude of the gradient field of the SDF strays considerably from one [16]. The method of using the gradient function to find the medial surface works because the continuous SDF becomes non-differentiable at points that are equidistant from multiple surfaces of the object. At these equidistant locations, there are multiple directions that share the steepest descent and the SDF becomes non-differentiable while working in continuous space. In discrete space, these points will instead have a gradient vector with a magnitude less than one. The non-differentiable sections are colored red in Figure 5.

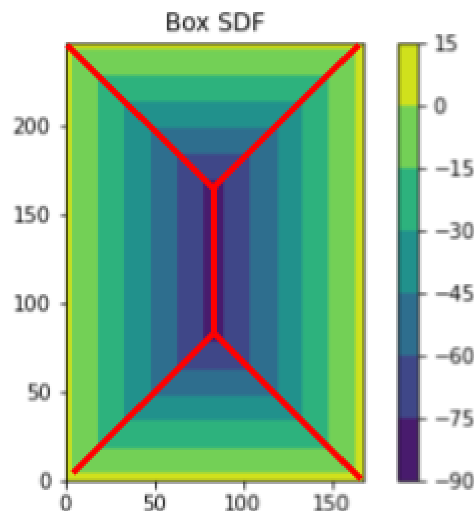

Figure 5: SDF of a rectangle with non-differentiable sections colored red

The combination of each voxel where the magnitude of the gradient field is below our set threshold provides us with the geometry of the medial surface, at which point all that is needed is the information for the radii of the maximal spheres at each medial surface point. Fortunately, the maximal sphere radii can be easily obtained by having each voxel of the medial surface geometry check the value stored in the SDF of the original object at its indices. The values retrieved from the SDF are then stored in the corresponding voxels of the medial surface

geometry, kept at a negative value to differentiate the medial surface voxels from the voxels that are not within the medial surface [16].

To improve the resulting medial surface, voxels that contain low-magnitude maximal radii are trimmed from the geometry to reduce the effects of the surface noise produced by the voxelization process and distance field calculations. When the process is complete, the algorithm returns a 3D array of the same size as the input array, where positive values represent voxels outside of the medial surface and negative values represent voxels that do belong to the medial surface, an example of which was shown earlier in Figure 2. The algorithm is also described in pseudocode in the appendix.

Finding the Weighting Field

The next step to generate the Voronoi-style infill pattern is to use the medial surface to generate a probability field. As mentioned in the overview, this field is generated using a wavefront propagation algorithm, meaning that the values of each voxel in the medial surface 'ripple' outwards, propagating to fill the volume of the model. By employing parallel computing, the wavefront propagation algorithm can be calculated by iterating the same algorithm on each cell multiple times.

First, a new array is formed where each index is now updated to contain another array of length 4. These sub-arrays are initially filled with negative values to show that the voxels have not yet been initialized. If the sub-array is for an index that lies within the medial surface, the sub-array will instead list its own indices (X, Y, Z) and the magnitude of the radial sphere for its location (R). Once this array has been produced, each voxel checks the values of each adjacent voxel and calculates the distance between its own indices and the indices stored in the adjacent voxel. If the calculated distance is less than the checked voxel's stored radial value and the checked radial value is greater than the voxel's own radial value, the current voxel stores the new indices and the new radial value. If either of the tests fail, the current voxel's values remain unchanged. The voxel continues until it has checked the value of each neighboring voxel, at which point each voxel returns its current values and the process repeats.

The process of having each voxel check the values of all of its neighbors is repeated a number of times equal to the magnitude of the most negative value in the array, as this gives each voxel that lies on the medial surface enough iterations for its value to propagate out to a distance equal to its radial value. The dependence of the iteration count on the most negative value means that the total processing time for this step is going to be dependent on the model, but it will always be at least as fast as iterating a number of times equal to half the magnitude of the smallest array dimension.

The algorithm now has a 4D array where the first three indices locate the voxel in space, and the fourth index stores the sub-array that contains the nearest medial surface coordinate and its radial information. Voxels that would be outside of the original object have negative values in each of the coordinate and radial data indices. The 4D array can be mapped to a 3D array by producing a new empty 3D array of a matching size and having each voxel in the new array determine its value by selecting the radial value from the corresponding index in the 4D array. To match with the convention of negative values being inside of the object and positive values being outside of the object, the entire new array is multiplied by -1. The field described by the new array can be seen in Figure 6, and the algorithm is described as pseudocode in the appendix.

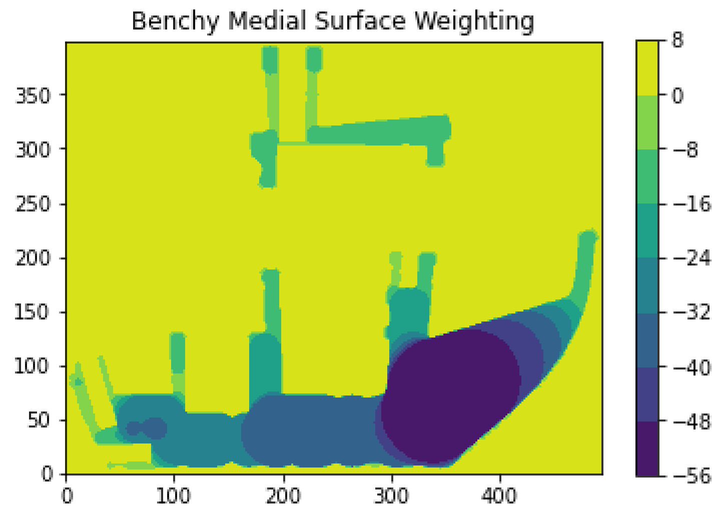

Figure 6: Weighting field for a benchy model, darker regions are less likely to generate seed points. Yellow regions are external to the model.

Distributing the Seed Points

The previous step has left us a 3D array where each cell contains either a positive value (meaning that it is outside of the object) or a negative value with a magnitude that matches the radial value of the nearest medial surface voxel. In order to generate our seed points, random numbers between 0 and 1 must be generated for each voxel. If the random number is less than a user-input threshold value times the square of the reciprocal of the value stored in the weighting array (an arbitrary equation chosen due to desirable material distribution), that index is set to zero, making it a seed point. Otherwise, the index is set to 1, ensuring that it is not a seed point. After the process of setting the seed points has been completed, the algorithm will return an array of equal dimensions to the input model's array with random points scattered inside, weighted such that all points will be within the original model with a varying volumetric density. Currently, the points are being distributed probabilistically within the object as a placeholder,

though a properly tuned deterministic point distribution algorithm would likely be just as effective, if not moreso.

Producing the Voronoi Cells

Producing Voronoi cells from a point field in discrete space is a process that has been described in previous papers, so this section will be brief [17]. First, the array is once again expanded to a 4D array, where the first three indices are the coordinates, while the last index will contain the X, Y, and Z indices of the nearest seed point and the distance from the current voxel to that seed point. The seed point coordinates are then filled in using the Jump Flood algorithm [7, 17]. From here, another pass is done where each voxel checks its neighbors to determine if any of the adjacent voxels are pointing to different seed points. If the voxel finds that at least one of its neighbors has a different set of stored coordinates, then it sets itself to zero, otherwise it sets itself to 1. The array is then collapsed back to a 3D array, and the distance field is calculated once again. To set the thickness of the Voronoi cell walls, a constant is subtracted from each voxel of the SDF.

Finishing Touches

From the previous step, there is a 3D array where the stored geometry represents a set of Voronoi cells, but the cellular structure currently fills the entire volume. The cellular structure can be trimmed via the usage of boolean operators that are commonly used in functional representation modeling [17, 22-23]. In this case, the intersection operator will be used. The Voronoi array is compared with the original voxelized model's array. In order to trim the shape, the desired geometry must be on the inside of both models. This can be performed by taking the maximum value for each index, meaning that only voxels that have a negative value in both arrays would be negative in the resulting array, leaving only the portion of the Voronoi cells that lie within the bounds of the starting object.

Next, the outer surface of the model must be added to the Voxel model. The process of adding the shell to the Voronoi infill has been discussed in our previous work, but an overview will be provided here [17]. First, the SDF of the original model is retrieved and a positive value that represents the desired thickness of the shell (in voxel units) is added to each index. The SDF thresholding process leaves a resulting model that is thinner than the original object by the input offset. The thinner model can then be subtracted from the original model via boolean subtraction (the same as boolean intersection, but the signs of the thinner model are inverted), and the resulting shell is then combined with the Voronoi structure via boolean union (similar to intersection, but the minimum value is taken instead of the maximum).

At this point, the desired infill structure has been properly generated, and the object can be exported to an intermediary file format for slicing. The conversion from a voxel model to STL can be performed via a marching cubes algorithm or the geometry can be exported to an image stack for DLP style printers.

Discussion

The Voronoi-style infill structure, when combined with the medial surface weighting scheme, provides a solution to the issue of thinner portions of the model being under-supported when seed points are distributed randomly or weighted to favor the surface of the model. The medial surface based algorithm provides a probabilistic increase in infill density at thinner regions and a decrease in infill density in larger, bulkier regions when compared to the distance field weighting method described in our previous work [17]. This comparison can be seen in Figure 7. Qualitatively, the distinction in point density can also be seen in the infill structure itself, shown in Figure 8.

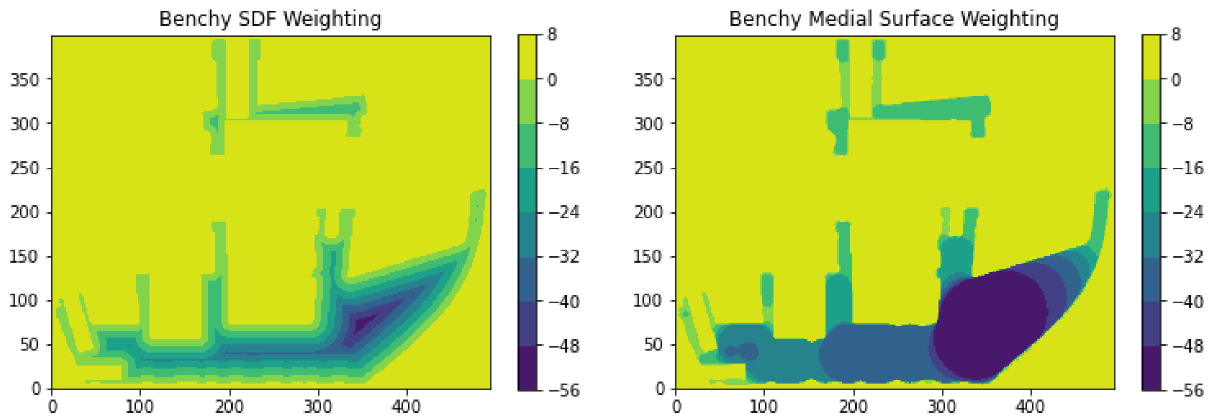

Figure 7: Comparison of SDF weighting field (left) vs. medial surface weighting field (right). Darker colors represent a lower probability of containing a seed point. It can be seen that the medial surface field would generate fewer seed points in the bulkier sections of the model and proportionally more points in the thinner sections of the model.

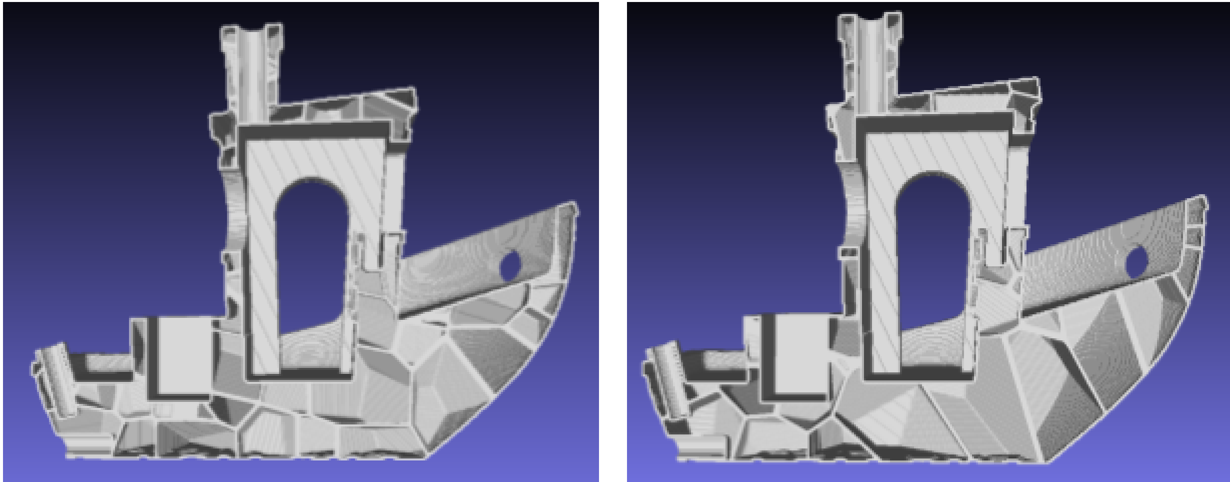

Figure 8: Comparison between SDF weighting field (Left) and Medial Surface Weighting Field (Right). Note larger cells in the bow of the ship in the right image.

Due to the extra steps taken to generate the medial surface weighting field, the processing time required to produce the medial surface weighting field is nearly twice as long as it takes to produce the SDF-based weighting field, with data shown for varying array sizes in Table 1. As the medial surface weighting field requires that you would first produce the SDF, this makes sense. However, the added complexity produces an infill structure that is differently tuned to the object's geometry and requires minimal user input, so we believe it to be a worthwhile tradeoff.

Table 1: Processing Time Comparison for Benchy Model using an NVidia GeForce GTX970

Array Sizes (Voxels)	SDF Weighting Field (sec)	Medial Surface Field (sec)
160x192x104	3.4	6.3
240x296x160	9.4	16.5
320x392x208	20.0	38.0
400x496x256	37.7	77.5

Many of the additional benefits and drawbacks of this approach are shared with previous work [3-5, 17]. Voronoi-style infill takes longer to generate than the industry standard grids or hexagons, and the time required to print the models is also increased. However, the benefits of having a graded infill structure can outweigh the drawbacks depending on the application and geometry of the part. As with most infill structures, the weighted Voronoi-style infill is best applied to material extrusion style printers, though it could be applied to SLA machines if drainage holes are left for excess resin to drain from the model.

Portions of the algorithms used within this paper take advantage of CUDA, an NVIDIA-based parallel computing toolkit [24]. The algorithms used within the paper rely heavily on being able to utilize a CUDA-capable graphics card to keep computation times low. Future improvements in the code could be done to allow for efficient implementation on serial processors, but they are not currently in development. Previous work has shown that parallel implementation was able to speed up processing time for many similar algorithms by multiple orders of magnitude, so no direct time comparison was performed for this algorithm [17].

Conclusion

The Voronoi-style infill structures performed as intended when their seed points were weighted based on the medial surface of the input model. Cellular density is increased at thinner portions of the model and decreased elsewhere to reduce material usage. The algorithms used allow several adjustments to control specifics of the infill structure, such as the cell wall thickness and the overall quantity of Voronoi cells. After being generated, the resulting structure can be exported in the form of a standard mesh file that's compatible with most slicing software. The full benefits of the algorithm and its detailed comparisons to other infill structures are still being explored, but the current results are promising.

References

[1] Mishra, Pradeep, Senthil, P., Adarsh, S., Anoop, M.S. "An investigation to study the combined effect of different infill pattern and infill density on the impact strength of 3D printed polylactic acid parts," *Composites Communications*, Vol. 24 (2021), https://doi.org/10.1016/j.coco.2020.100605

[2] Pandzic, A., Hodzic, D., & Milovanovic, A. "Effect of infill type and density on tensile properties of PLA material for FDM process." *30th Daaam International Symposium on Intelligent Manufacturing and Automation*, Zadar Croatia., Vol. 30 No. 1 (2019) pp. 0545. https://www.daaam.info/Downloads/Pdfs/proceedings/proceedings_2019/074.pdf

[3] Martínez, Jonàs, Dumas, Jérémie, and Lefebvre, Sylvain. "Procedural Voronoi Foams for Additive Manufacturing." *ACM Transactions on Graphics* Vol. 35 No. 4 (2016): DOI 10.1145/2897824.2925922. https://dl.acm.org/citation.cfm?doid=2897824.2925922.

[4] Martínez, Jonàs, Hornus, Samuel, Song, Haichuan, and Lefebvre, Sylvain. "Polyhedral Voronoi diagrams for additive manufacturing." *ACM Transactions on Graphics* Vol. 37 No. 4 (2018): DOI 10.1145/3197517.3201343. https://dl.acm.org/citation.cfm?doid=3197517.3201343.

[5] Martínez, Jonàs, Song, Haichuan, Dumas, Jérémie, and Lefebvre, Sylvain. “Orthotropic k-nearest foams for additive manufacturing.” *ACM Transactions on Graphics* Vol. 36 No. 4 (2017): DOI 10.1145/3072959.3073638. https://dl.acm.org/citation.cfm?id=3073638

[6] Lu, Lin, Sharf, Andrei, Zhao, Haisen, Wei, Yuan, Fan, Qingnan, Chen, Xuelin, Savoye, Yann, Tu, Changhe, Cohen-Or, Daniel, and Chen, Baoquan. “Built-to-Last: Strength to Weight 3D Printed Objects.” *ACM Transactions on Graphics* Vol. 33 No. 4 (2016): DOI 10.1145/2601097.2601168. https://dl.acm.org/citation.cfm?doid=2601097.2601168.

[7] Rong, Guodong and Tan, Tiow-Seng. “Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform.” *Symposium on Interactive 3D Graphics and Games.* Pp 109-116. Redwood City, CA, March 14-17, 2006. DOI 10.1145/1111411.1111431. https://dl.acm.org/citation.cfm?id=1111431

[8] Blanding, R., Brooking, C., Ganter, M., & Storti, D., “A Skeletal-Based Solid Editor.” *Proceedings of the fifth ACM symposium on Solid modeling and applications* (1999): pp. 141-50

[9] Cornea, N.D., Silver, D., & Min, P., “Curve-Skeleton Properties, Applications and Algorithms.” *IEEE Trans Vis Comput. Graph., 13*(3) (2007): pp.530-48

[10] Huang, H., Wu, S., Cohen-Or, D., Gong, M., Zhang, H., Li, G., & Chen, B., “L_1-Medial Skeleton of Point Cloud.” *ACM Trans. Graph.*, *32*(4) (2013): pp.65-1

[11] Attali, D. & Montanvert, A., “Computing and Simplifying 2D and 3D Continuous Skeletons.” *Computer Vision and Image Understanding 67*(3) (1997): pp.261-273

[12] Lee, Y. & Lee, K., “Computing the medial surface of a 3-D boundary representation model.” *Advances in Engineering Software 28* (1997): pp.593-605

[13] Dey, T.K. & Zhao, W., “Approximate Medial Axis as a Voronoi Subcomplex.” *Proceedings of the seventh ACM symposium on Solid Modeling and Applications* (2002): pp. 356-366

[14] Puig Puig, Anna, “Discrete Medial Axis Transform for Discrete Objects.” (1997)

[15] Delamé, T., Roudet, C., & Faudot, D., “From A Medial Surface To A Mesh” *Eurographics Symposium on Geometry Processing 31*(5) (2012): pp.1637-46

[16] Williams, T.; Storti, D.; Ganter, M.; “Voxelized Skeletal Modeling Techniques via Complemental Skeletons.” *SAMPE Virtual Series* (2020)

[17] Williams, T., Langehennig, S., Ganter, M., & Storti, D., "Using Parallel Computing Techniques to Algorithmically Generate Voronoi Support and Infill Structures for 3D Printed Objects." *Solid Freeform Fabrication Symposium Proceedings 30* (2019): pp.1830-52

[18] Hoff, Kenneth, Keyser, Joh, Lin, Ming, Manocha, Dinesh, and Culver, Tim. "Fast computation of generalized Voronoi diagrams using graphics hardware." *Proceedings of the 26th annual conference on Computer graphics and interactive techniques.* pp. 277-286 Los Angeles, CA, August 8-13 1999. DOI. 10.1145/311535.311567

[19] Rong, G., & Tan, T., "Jump Flooding in GPU with Applications to Voronoi Diagram and Distance Transform." *Symposium on Interactive 3D graphics and Games* (2006): pp.109-16

[20] Creative Tools, Sweden, 2016, "#3DBenchy - the jolly 3D printing torture-test" http://www.3dbenchy.com/

[21] Pederkoff, Christian, 2019 "Turn STL files into voxels, images, and videos" https://github.com/cpederkoff/stl-to-voxel

[22] Ensz, Mark, Storti, Duane, and Ganter, Mark. "Implicit Methods for Geometry Creation." *International Journal of Computational Geometry & Applications.* Vol. 8 No. 3 (1998): pp. 509-536 https://www.worldscientific.com/doi/abs/10.1142/S0218195998000266.

[23] Zhang, Di, "*A GPU Accelerated Signed Distance Voxel Modeling System*". 2016. University of Washington Seattle, PhD dissertation. <http://hdl.handle.net/1773/38175>

[24] NVIDIA, Vingelmann, P. & Fitzek, F.H.P., 2020. *CUDA, release: 10.2.89*, Available at: https://developer.nvidia.com/cuda-toolkit.

The code described in this paper will be freely available for download at:
https://github.com/tjwill95/SkeleVoronizer

Appendix

Several algorithms are mentioned in the Methods section that could use further explanation. In order of appearance, here is a description of the process for each of the referenced algorithms in the following format:

```
Operation (Inputs: A, B, C; Outputs: X, Y, Z):
PSEUDOCODE
```

I will be using Python as a baseline for my pseudocode, meaning that the index of an array is dictated by square brackets (u[i,j,k] = element stored at index i,j,k), a colon selects all values within that index, (u[i,:,:] = 2D array), and the # symbol is used to dictate comments.

Some algorithms require several functions. The main algorithm will be **in bold**, sub-algorithms will be underlined, and there will be a break indicated by --- between algorithms.

```
Gradient Magnitude Field (Inputs: SDF of Object;
                          Outputs: Gradient Magnitude Field):
Gradient Magnitude Field = Copy of SDF of Object
Gradient Kernel (SDF of Object, Gradient Magnitude Field)
#Done in parallel, exact setup depends on the language/platform
Return Gradient Magnitude Field

CFD (Inputs: P, N, H; Outputs: Slope):
#P = value one step forward from our coordinate
#N = value one step backward from our coordinate
#H = step size
return ((P-N)/(2*H))^2
#Squared central difference equation for first derivative

Gradient Kernel (Output, Input):
i, j, k = thread indices
#1D derivatives
dx = CFD(Input[i-1,j,k],Input[i+1,j,k], 1)
dy = CFD(Input[i,j-1,k],Input[i,j+1,k], 1)
dz = CFD(Input[i,j,k-1],input[i,j,k+1], 1)
#Diagonal (2D) derivatives
dPxPy = CFD(Input[i-1,j-1,k], Input[i+1,j+1,k], 2^(0.5))
dNxPy = CFD(Input[i-1,j+1,k], Input[i+1,j-1,k], 2^(0.5))
dPyPz = CFD(Input[i,j-1,k-1], Input[i,j+1,k+1], 2^(0.5))
```

```
dNyPz = CFD(Input[i,j-1,k+1], Input[i,j+1,k-1], 2^(0.5))
dPxPz = CFD(Input[i-1,j,k-1], Input[i+1,j,k+1], 2^(0.5))
dNxPz = CFD(Input[i-1,j,k+1], Input[i+1,j,k-1], 2^(0.5))
#Finding Magnitudes
d1 = (dx+dy+dz)
d2 = (dPxPy+dNxPy+dz)
d3 = (dx+dPyPz+dNyPz)
d4 = (dPxPz+dy+dNxPz)
Output[i,j,k] = min(d1,d2,d3,d4)^(0.5)
#Finds the squares of the magnitudes, then compares them to find
#the minimum and takes the square root, storing the output in
#the corresponding cell of the output matrix.
```

```
Medial Surface(Input: SDF of Model, Grad Magnitude Field, Grad
               Threshold, SDF Threshold; Output: Med Surface):
#Gradient threshold is usually 0.85, as a magnitude below this
#indicates that the gradient function would return an 'invalid'
#value.
#SDF Threshold helps get rid of false positives, if the SDF
#value is greater than the #Threshold (ie, positive or low
#magnitude negative), the voxel can be ignored with minimal
#impact on the resulting skeleton, usually set to -1.
Med Surface = Copy of SDF of Model
Med Surface Kernel(Med Surface, Grad Magnitude Field, Grad
                   Threshold, SDF Threshold)
#Done in parallel, exact setup depends on the language/platform
Return Med Surface

Med Surface Kernel(Med Surface, Grad Field, Grad Thresh,
                   SDF Thresh):
i, j, k = thread indices
if (Med Surface[i,j,k]>SDF Thresh) or
     (Grad[i,j,k]>Grad Thresh):
   Med Surface[i,j,k] = 1
   #If the gradient is above the threshold, we are necessarily
   #outside of the med surface, so any positive value would
   #remove us from the object. If the med surface value at that
   #point (currently set to match the SDF value) is above the
   #threshold, we are either outside the object, so any positive
   #value maintains that, or we are inside with a low magnitude
   #and can be safely ignored.
```

```
Weight Field(Input: Med Surface; Output: VMO):
#Med Surface is the voxelized medial surface produced earlier
#VMO is the voxel model output
X,Y,Z = Lengths of the first, second, and third indices of Med
        Surface
VMOR = New array of dimensions (X,Y,Z,4), values set to 5000
VMOW = New array of dimensions (X,Y,Z,4), values set to 5000
#These act as read and write matrices, so that the Weight Field
#Kernel has a stable array to read data from, and eliminates
#the possibility of a racing condition.
Weight Field Setup Kernel(Med Surface, VMOR)
#Done in parallel, exact setup depends on the language
Iterations = roundup(abs(min(Med Surface)))
#Finds the largest magnitude voxel on the medial surface.
While Iterations > 0:
     Weight Field Kernel(VMOR,VMOW)
     #Done in parallel, exact setup depends on the language
     VMOW, VMOR = VMOR, VMOW
     Iterations = Iterations-1
Weight Field Unpack Kernel(VMOR,Med Surface)
#Done in parallel, exact setup depends on the language
Return Med Surface

Weight Field Setup Kernel(Med Surface, VMOR):
i,j,k = thread indices
if Med Surface[i,j,k]<0:
     VMOR[i,j,k,:]=i,j,k,-Med Surface[i,j,k]
Else:
     VMOR[i,j,k,:]=-1,-1,-1,0

Weight Field Kernel(VMOR,VMOW):
i,j,k = thread indices
CX = VMOR[i,j,k,0] #Current X
CY = VMOR[i,j,k,1] #Current Y
CZ = VMOR[i,j,k,2] #Current Z
CV = VMOR[i,j,k,3] #Current Medial Surface Value
CCI = 0 #Checked Cell Index
```

```
While CCI < 27:
     CC = (i+((CCI//9)%3-1),j+((CCI//3)%3-1),k+(CCI%3-1))
     #Checked Coordinate, cycles through all adjacent cells
     NX = VMOR[CC,0] #New X
     NY = VMOR[CC,1] #New Y
     NZ = VMOR[CC,2] #New Z
     NV = VMOR[CC,3] #New Medial Surface Value
     New Distance = Distance(CX,NX,CY,NY,CZ,NZ)
     if New Distance ≤ NV and NV > CV:
          CX,CY,CZ,CV = NX,NY,NZ,NV
     CCI = CCI+1
VMOW[i,j,k,:] = CX,CY,CZ,CV

Distance(x1,x2,y1,y2,z1,z2):
Return sqrt((x1-x2)^2+(y1-y2)^2+(z1-z2)^2)

Weight Field Unpack Kernel(VMOR,Med Surface):
i,j,k = thread indices
if VMOR[i,j,k,3]==0:
     Med Surface[i,j,k]=1
     #Means that the index is not within any medial spheres
else:
     Med Surface[i,j,k]=-VMOR[i,j,k,3]
```

```
Point Generation(Input: VMI, Threshold; Output: VMO):
#VMI is a distance/depth field of the object, with low magnitude
#values near the surface.
Random = Array of the same size as VMI with random numbers
between 1 and 0 for each cell.
VMO = Copy of VMI
#This keeps from accidentally overwriting anything in the input
Point Generation Kernel(VMI, VMO, Random, Threshold)
#Done in parallel, exact setup depends on the language
Return VMO

Point Generation Kernel(VMI, VMO, Random, Threshold):
i,j,k = thread indices
```

```
if VMI[i,j,k] < 0 and Random[i,j,k] < -1*Threshold/VMI[i,j,k]^2:
     VMO[i,j,k] = -1
else:
     VMO[i,j,k] = 1
END
```

Boolean Operations:

All input matrices must have the same dimensions. All boolean operations function on a very similar idea, so we can construct a single kernel and modify the inputs as needed. For our purposes, let's design the kernel for the Boolean Union. As this kernel is going to be applied to all cells, and none of the cells have any interdependence on the matrices that they're in, this can be applied simultaneously via parallel computational methods.

```
Boolean(VM1,VM2,VMO):
i,j,k = thread indices
VMO[i,j,k] = min(VM1[i,j,k],VM2[i,j,k])
END

Union(Input: VM1,VM2; Output: VMO):
VMO = Boolean(VM1,-1*VM2)
#Done in parallel, exact setup depends on the language
Return VMO

Intersect(Input: VM1,VM2; Output: VMO):
VMO = -1*Boolean(-1*VM1,-1*VM2)
#Done in parallel, exact setup depends on the language
Return VMO

Subtract(Input: VM1,VM2; Output: VMO):
#VM1 is the base, VM2 is the tool
VMO = -1*Boolean(-1*VM1,VM2)
#Done in parallel, exact setup depends on the language
Return VMO
```