NASA/TM-20220008923

Machine Learning to Increase the Quality and Repeatability of 3D Printing - Workflow

Kevin A. Hunt
New College of Florida, Sarasota, Florida

Austin B. Ebel
College of William and Mary, Williamsburg, Virginia

Evan S. Rose and Sean C. Zylich
Governor's School for Science and Technology, Hampton, Virginia

Benjamin D. Jensen, Kristopher E. Wise, Emilie J. Siochi, John M. Gardner, and Godfrey Sauti
NASA Langley Research Center, Hampton, Virginia

August 2023

NASA/TM-20220008923

Machine Learning to Increase the Quality and Repeatability of 3D Printing - Workflow

Kevin A. Hunt
New College of Florida, Sarasota, Florida

Austin B. Ebel
College of William and Mary, Williamsburg, Virginia

Evan S. Rose and Sean C. Zylich
Governor's School for Science and Technology, Hampton, Virginia

Benjamin D. Jensen, Kristopher E. Wise, Emilie J. Siochi, John M. Gardner, and Godfrey Sauti
NASA Langley Research Center, Hampton, Virginia

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23681-2199

August 2023

Acknowledgments

The authors would like to thank Chris Stelter, David Anderegg, Jae-Woo Kim, Christine Linsinbigler, Christine Dillard, Valerie Ellis, Karen Taminger, Forrest Barber, and Jacob Gissinger for their help and support with this project.

Available from:

NASA STI Program / Mail Stop 148
NASA Langley Research Center
Hampton, VA 23681-2199
Fax: 757-864-6500

Table of Contents

Introduction

The imprecise nature of three-dimensional (3D) printing limits the use of the technology beyond prototyping. For production of end-use parts, such as those for aerospace applications, improvements are needed to enhance quality and repeatability [1]. Much of the difficulty in obtaining high quality printed parts lies in finding optimum printing parameters. Currently, optimization requires trial and error performed by an expert [2]. Finding the optimum printing parameters is also obfuscated by the variation in optimum parameters throughout the part due to part geometry and printer effects. To allow for locally optimized printing parameters, one can envision a machine learning algorithm that could view an object, predict the best printing parameters, and communicate these parameters to a printer. With this scenario in mind, a tool was developed that can predict and implement locally optimized printing parameters in 3D printing. This tool consists of elements designed to detect errors in a printed part, predict the probability of local flaws occurring at each point in the part, and select the optimal local parameters for the highest quality part given hardware limitations. The results of this work were highlighted in *Advanced Materials Technologies* [3]. This paper includes an in-depth discussion of the workflow and algorithms involved with this tool that were not detailed in the journal publication. This information may provide insight on lessons learned that led to the culmination of the approach used in the journal paper.

The research presented in this Technical Memorandum (TM) will cover the steps used to create the machine learning tool in chronological order as developed. It will begin with building the master dataset covering printing test parts, collecting geometric data, error detection using image classification, and combining the data to create the master dataset. Then, using this master dataset to build a machine learning model to predict errors in the parts will be discussed. These discussions will include algorithms tested, use of the model to predict errors in parts, and translating these predictions into optimized parameter instructions the 3D printer is physically capable of performing. Finally, the results of using this tool to perform printing optimization on simple parts will be described.

Disclaimer on Figure Text Formatting

Due to many of the figures included in this document being taken directly from running program code, there will be unavoidable inconsistencies in the fonts and text formatting. Despite this, the authors deem there to be sufficient benefit to the reader to having these for a fuller appreciation of the work and thus their inclusion. The figures and their text can be readily resized and seen in detail in digital versions of this document.

Glossary

(I) Additive Manufacturing

AM	Additive manufacturing
Blob	An AM defect classification
CAD	Computer Aided Design
CURA	Open-source slicing and toolpath planning software for 3D printing
Delamination	An AM defect classification
NetFabb	Toolset used in CAD for additive manufacturing
OpenSCAD	Open-source CAD software
Pronterface	User interface for monitoring print and machine status

STL	Stereolithography CAD file format
Warp	An AM defect classification
Z-axis	Direction perpendicular to the build surface on a 3D printer

(II) Commercial and Open Source Data Analysis/Machine Learning Tools

Adam	Optimization algorithm for Deep Learning
AlexNet	A pre-trained convolutional neural network
K-d tree	Space partitioning data structure for organizing data points.
Libigl	C++ geometry processing library
MATLAB	A programming language primarily used for numeric computation
Pandas	Python data analysis library
Python	Interpreted high-level and general-purpose programming language
R	Programming language and free software environment for statistical computing
Scikit-learn	Machine learning library for the Python

(III) Terms Introduced in this Paper

3 sides of a cube	Simple geometry created for this work
Image Classifier	Model for identifying features in an image
Image Training Dataset	Dataset that includes segmented images used to train the image classifier
Master Dataset	Dataset which includes all information known about printed parts segments including: printing parameters, geometry, XYZ, and errors
"None"	Error classification of an image the does not contain any other error tracked in this work (e.g., blob, warp, delamination).
Prediction Model	Model used to calculate the probability of errors occurring in the part
XYZ	Data representing the X, Y, and Z coordinates of some point on a printed part

Building the Master Dataset

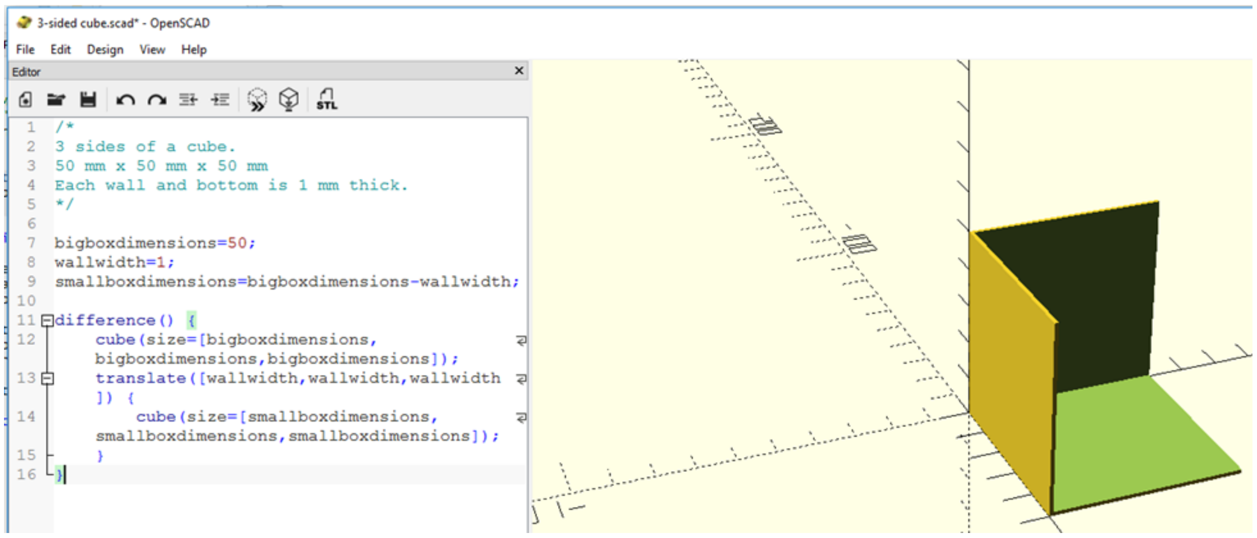
This section covers the work to create a master dataset used to train the tool. Steps include printing parts using a variety of parameters, creating a point cloud representation of the geometry, obtaining geometric information at those points, identifying errors at those points using an image classification network, and combining this information to create a master dataset.

Printing Test Parts

To create the master dataset, parts were printed using various print parameter combinations. This involved choosing the part geometry, material, print parameters of interest and printing the object with the various print parameter combinations.

CAD Model

The part geometry used is “3 sides of a cube”. This simple 3D geometry was prone to the types of errors that are visually identifiable. Only three sides were printed for speed, material savings, and stackability. The object was designed in OpenSCAD and exported as a STL file (Figure 1). The STL file was then sliced into machine readable G-code by Cura Lulzbot Edition for the Lulzbot Mini.



```
/*
3 sides of a cube.
50 mm x 50 mm x 50 mm
Each wall and bottom is 1 mm thick.
*/

bigboxdimensions=50;
wallwidth=1;
smallboxdimensions=bigboxdimensions-wallwidth;

difference() {
    cube(size=[bigboxdimensions,
    bigboxdimensions,bigboxdimensions]);
    translate([wallwidth,wallwidth,wallwidth
    ]) {
        cube(size=[smallboxdimensions,
    smallboxdimensions,smallboxdimensions]);
    }
}
```

Figure 1. OpenSCAD used to create the model.

Material

Each part was printed using Lulzbot Natural ABS 3 mm filament (SKU: 1775201312) procured from FAME 3D (Fargo, ND). Three reels of filament were required to print all test components.

Print Parameter Selection

The parameters of interest were the nozzle temperature, bed temperature, print speed, extrusion multiplier, and fan speed. These ranges were selected based on prior work [4-5].

- Bed temperature: 100°C, 110°C, 120°C.
- Print speed: 20 mm/s, 40 mm/s, 60 mm/s, 80 mm/s.
- Extrusion multiplier: 90%, 100%, 110%.
- Fan speed: 30%, 45%, or 80%.
- Nozzle temperature: 225°C, 235°C, 245°C.

The combinations of parameters to be tested were selected randomly to yield a consistent number of “good” and “poor” parameter combinations. To do this, R code was created to build a “list of prints” consisting of randomly ordered parameter combinations for printing.

Writing G-code and Printing

There were 324 possible combinations of the print parameters and ranges selected. To streamline G-code writing and ensure consistency, a base G-code file of the selected geometry was created using Cura Lulzbot Edition for the Lulzbot Mini (with extrusion multiplier at 100%). A Python script was written to parse through this base G-code and modify the print parameters to those in the list of prints. Parts were then printed using a Lulzbot Taz Mini 3D printer using the order specified in the list of prints. At the time of printing, the ambient room temperature and ambient humidity were recorded. A thermocouple was used to determine the actual nozzle temperature for comparison with the nozzle thermistor used by the printer control system. These readings confirmed the reliability of the Pronterface for reading nozzle temperature. Prints were sent to the Lulzbot Mini printer using Pronterface. A total of 141 prints were completed.

Creating a Point Cloud

Data at specific areas on the part were required to find local optimum parameters along the toolpath using a machine learning model. Therefore, a point cloud model of the geometry was created to facilitate matching of specific locations with print parameters and error types. To do this, the G-code was segmented into 2-mm sections along the toolpath to create a denser population of XYZ points. A Python script was written to pull the extrusion commands out of a base G-code (skipping the skirt), segment them into evenly divided sections, and rewrite the G-code as the segmented version.

The motivation for rewriting the G-code in this way was three-fold: (1) these segments may have different optimum parameters picked by the model, the G-code will need to be segmented in this way in the future; (2) to ensure that the segmenting alone (without changing any of the parameters from segment to segment) did not negatively impact the print; (3) to test the impact of varying the parameters from segment to segment on part quality.

Collecting Geometric Data

Geometric information at the XYZ points needed to be correlated with the other information required to train the model. To do this, geometric data was gathered from a refined STL file at points corresponding to those in the G-code. Details of this process are discussed below.

Refine the STL

The original 3 sides of a cube STL file did not contain enough vertices to provide accurate geometric information at every point along the segmented G-code. Therefore, the model was imported into NetFabb and refined to obtain more vertices (Figure 2).

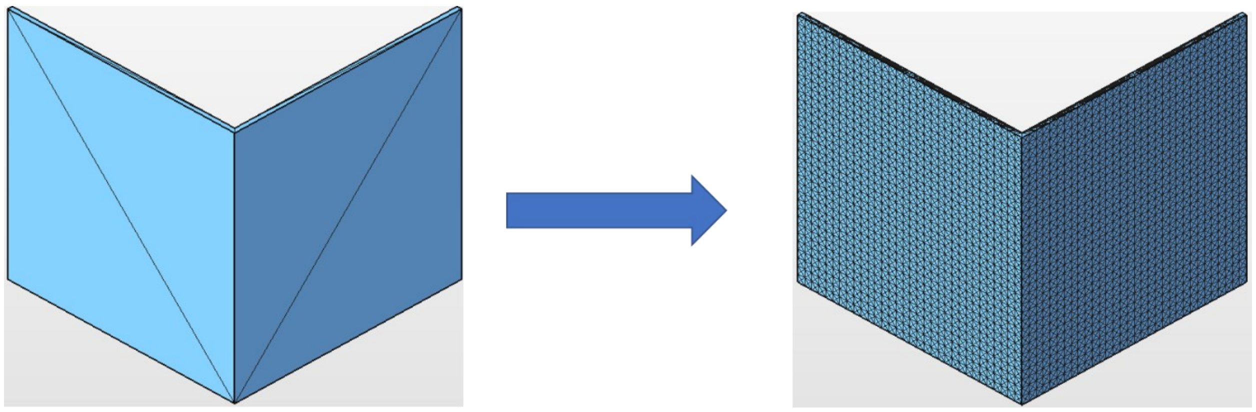


Figure 2. Model refined with NetFabb.

The following process was completed to refine the mesh:

- File → Open Project
- Prepare → Repair Part

Under the new menu select:

- Mesh Edit → Refine Triangle Mesh

Set the first value, Max. Edge Length, to 0.1 mm (or the desired maximum) and click "Apply repair."

Follow:

- File → Export Path → as STL

Using a 0.1 mm maximum edge length ensures that every G-code segment maps almost exactly to an STL vertex.

Libigl

Libigl, a C++ geometry processing library, was used to obtain geometric information for each vertex of the STL file [6]. This information included the vertex normals, Gaussian Curvature, and Discrete Mean Curvature, and X, Y, Z coordinates. Libigl has bindings in Python, which helped streamline integration of these values into the master dataset.

Other Geometric Information

In addition to the geometric data obtained from Libigl, other factors such as angle between the z-axis and print direction were also calculated. It was anticipated that the angle with the z-axis may provide more insight to overhangs, while print direction (defined as a vector facing out of the front of the printer) might give information about orientation. The angles relative to the z-axis are illustrated in Figure 3. Vertex normals found through Libigl were used to find both θ and ϕ and write functions that return these angles for every vertex in the STL file. The print angle was found by using the standard angle between two vectors equation:

$$\alpha = \cos^{-1}\left(\frac{u \cdot v}{|u||v|}\right)$$

Where v is the vector (0,1,0) facing out of the print bed. A function computed θ and ϕ for each vertex. After this process, all the geometric data calculation functions were contained in one file.

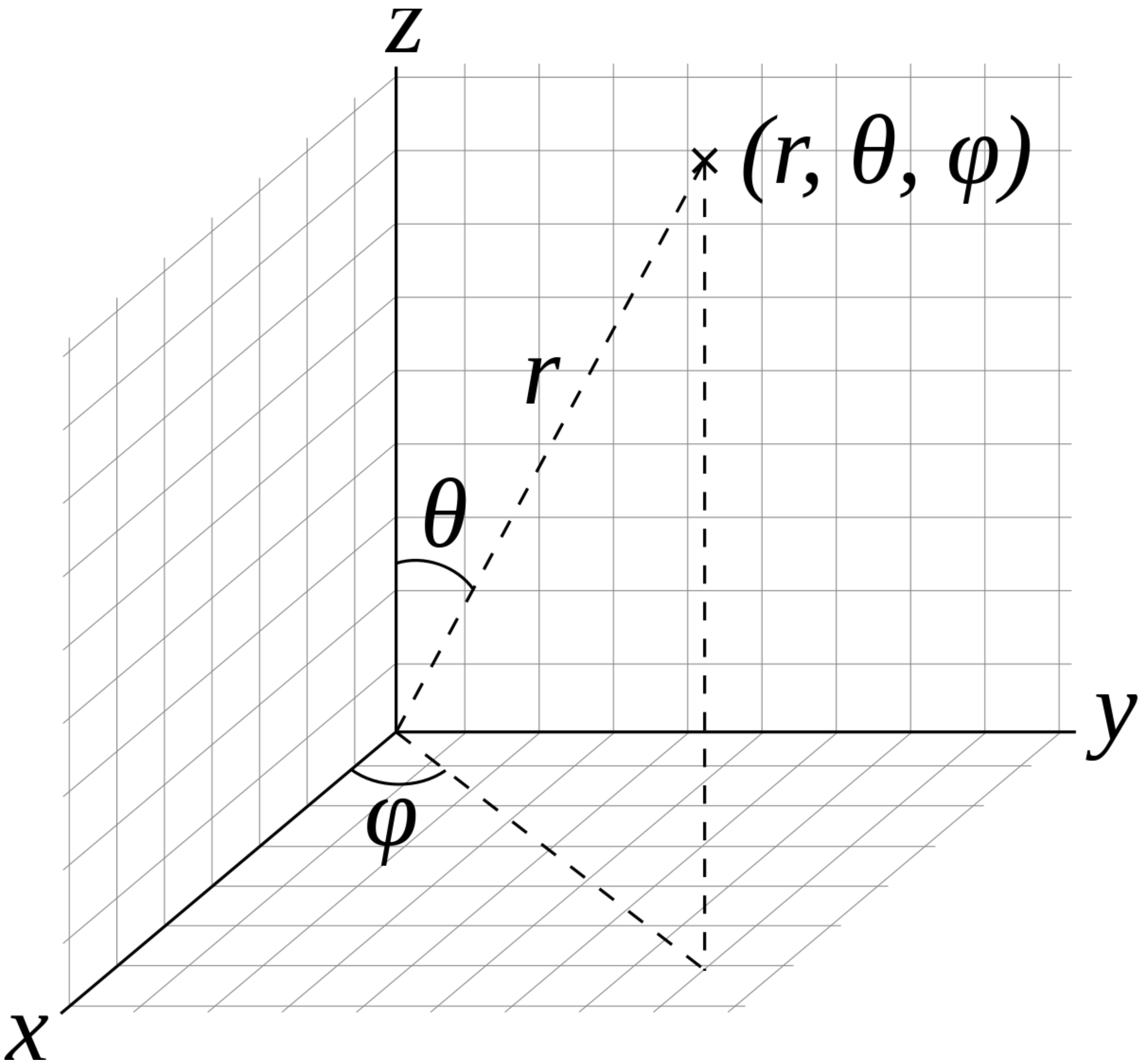


Figure 3. Illustration of vertex normals.

Align G-code and STL Points

Previously, geometric data about the vertices of the STL file were gathered, but the model needed to be trained to the geometric information of the G-code points. Thus, an efficient way to convert STL vertex information into G-code segment information was required and is discussed below.

The first challenge was the misalignment between the G-code toolpath and the STL vertices at the edge of the part as shown in Figure 4. The G-code toolpath is outlined in light blue and black dots represent the refined STL vertices. Note that STL vertices on the top and right side go beyond that of the toolpath. In Cura, the toolpath was adjusted to take the nozzle width of 0.5 mm and material deposition into account. Therefore, the dimensions of the toolpath were 49.5 mm x 49.5 mm x 49.4 mm. In addition, the walls of the original STL file were 1-mm wide; however, Cura created a toolpath along the walls of only 0.5 mm. G-code segments on the corner may not have received the correct ‘corner-like’ information from Libigl. Therefore, correction was required to align the points. Initially, XYZ coordinates were simply scaled down to meet the G-code toolpath, however this did not appropriately scale the wall widths. Therefore, the CAD model was adjusted to reflect the dimensions of the G-code path. This solution was not generalizable, however, and other solutions will be considered in the future.

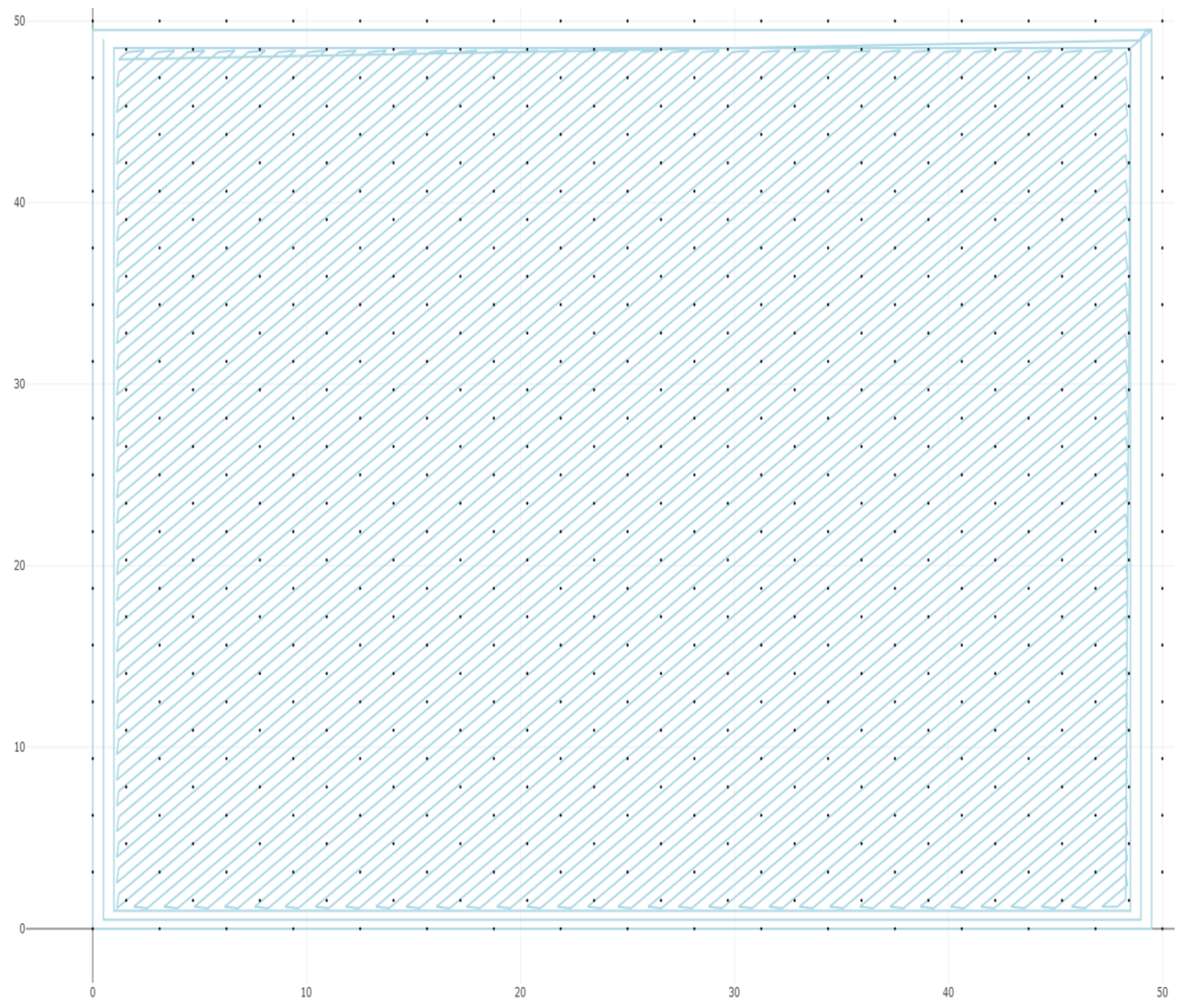


Figure 4. Overlay of the printer toolpath (blue) and STL points. The STL points must be scaled to match the toolpath.

Once the STL and G-code points were aligned, their XYZ values were correlated to associate geometric information from the STL file with G-code segments. Correlation was done by finding the minimum X, Y, Z coordinates of both the STL and G-code files and calculating the difference between each of them. Then, that difference was added to the STL file to obtain the same coordinate system.

Map G-code to Nearest STL

Once the STL and G-code were aligned, the G-code points needed to be mapped to the closest STL point to gather geometric information. A visual representation of the G-code cut up into 2-mm segments and mapped to the nearest vertex refined to 0.1 mm is shown in Figure 5. The G-code is colored by its associated vertex. Solid colors surrounding the vertices indicates that the mapping algorithm worked as expected. Every STL point does not directly correspond to every other G-code point since the STL file is segmented only every 0.1 mm.

Code was created for using the k-d tree to associate STL vertices with G-code segments. A function first removed the duplicate vertices in the STL file that were a result of multiple triangles containing the same vertex. Then, another function added the offset described in the ‘Align the G-code and STL Points’ section. Finally, the k-d tree was created for ~3,000,000 vertices, taking around 15 minutes. However, once created that tree could be saved and loaded into the program. The actual act of assigning vertices to G-code segments was optimized using the k-d tree, and therefore took around 20 s. Larger part sizes may require different methods to ensure the problem does not become computationally expensive.

Including all segments, the average distance to the STL vertex was 0.04 mm. The geometric data of each G-code point was plotted to position to ensure correct alignment, scale, and to verify mapping functionality (Figure 6). G-code points are colored by geometric properties from the associated STL points. The final training data ultimately dropped the vertex information and distances between G-code segments and STL vertices.

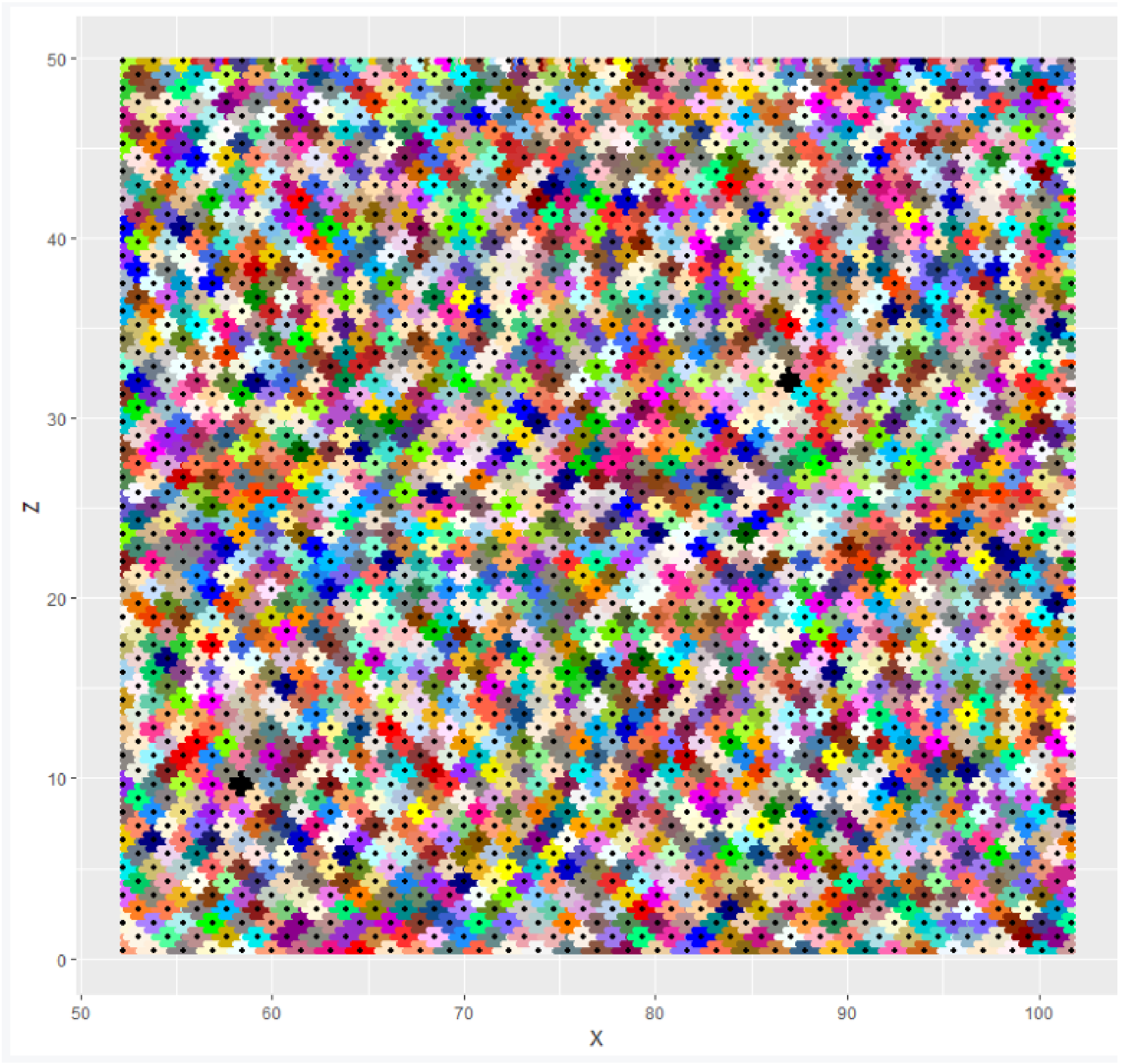


Figure 5. G-code toolpath points (colored blocks) are matched to their associated STL points (black dots). Solid colors surround the vertices indicating that the algorithm correctly corresponds G-code points to STL vertices.

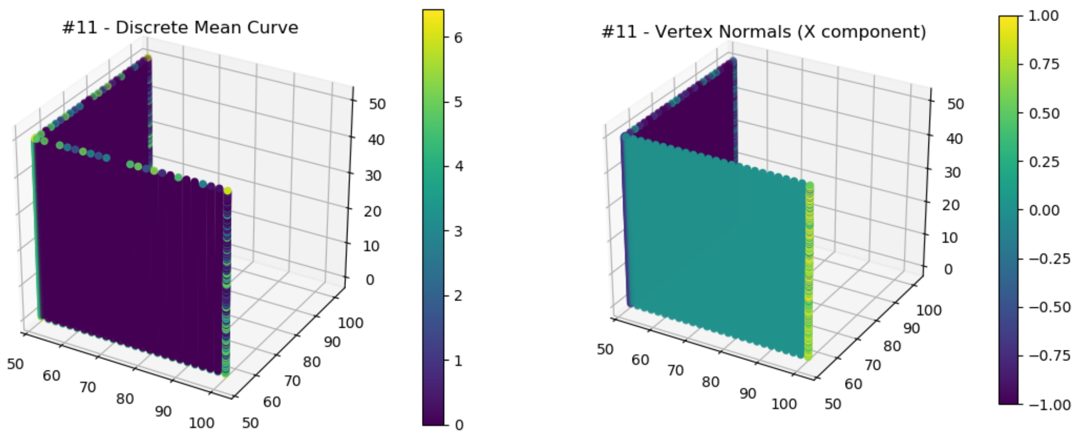


Figure 6. G-code points colored by geometric properties from associated STL points.

Collecting Error Data with an Image Classifier

Information on G-code segments and their geometric properties now needed to be combined with data on printing parameters and errors in the printed part. The workflow for this section involved training an image classification model to classify error types and then using the model to scan images of the printed part and identify the error type at each G-code point.

The Image Classifier was created using transfer learning with AlexNet in MATLAB to automatically determine the likelihood of an error type based on pictures of the walls [7]. The pixel coordinates of each picture were translated into the necessary G-code segments for our model.

Collecting Images

The Image Classifier used an edge detection method to find the printed part within the image. To aid the edge detection, a black background was chosen for maximum contrast against the natural ABS. Pictures of the printed part were taken in the order of the bottom with label, then front (XZ), then left (YZ). This order was essential for using a script to prepare the data for classification. Great care was taken to ensure the images were in focus and bright.

Organizing Images

The images needed to be organized in a specific way for the MATLAB script to know which print number and face it was processing. Each print number received its own folder named with the Print Number. Inside the folder were the two faces named *PrintNumber_XZ*, *PrintNumber_YZ* shown in Figure 7.

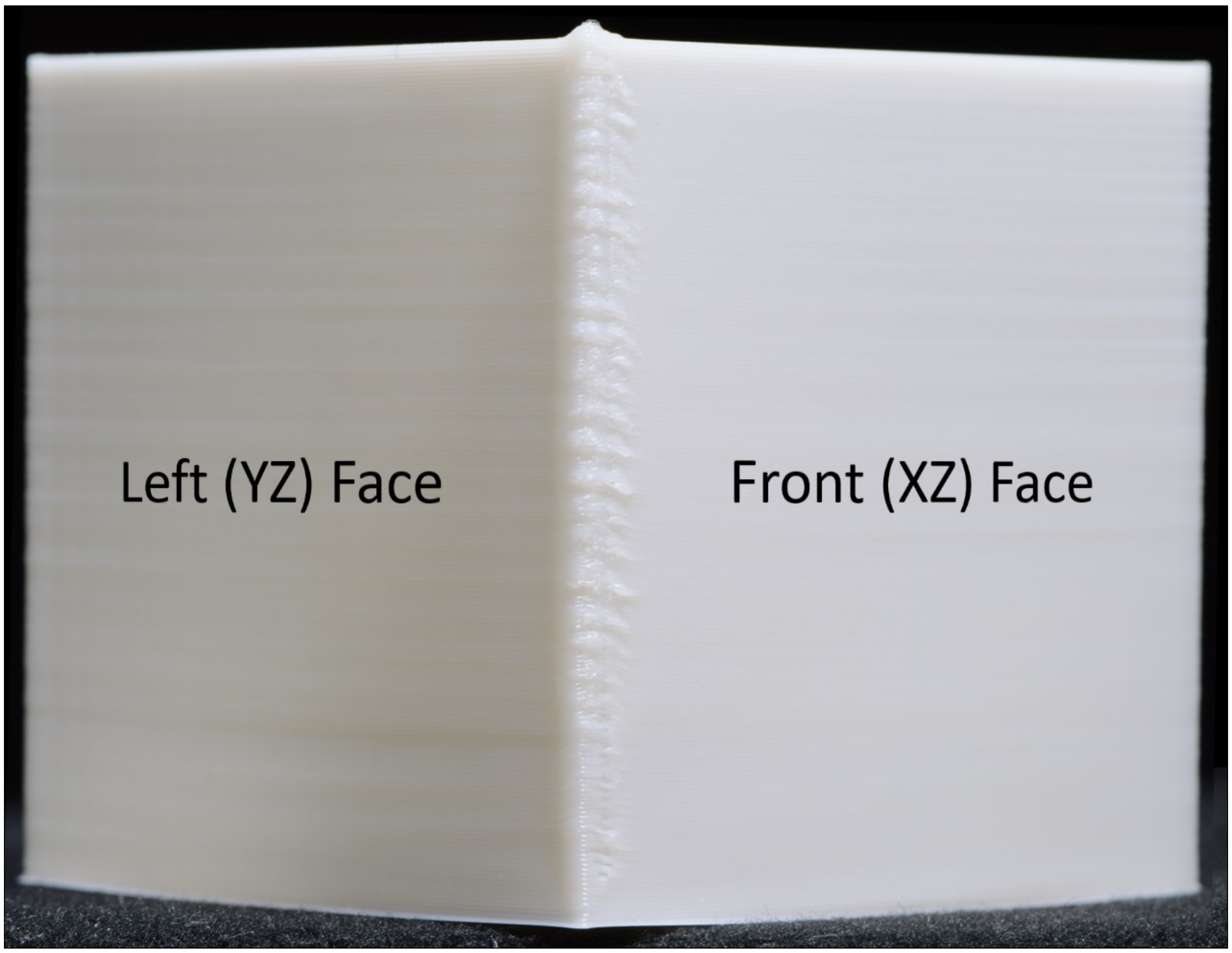


Figure 7. The two faces imaged on the 3-sides-of-a-cube printed part.

A Python script was developed that looped over one folder of images direct from the camera and renamed and organized the images in this way. The first step was to ensure that the images from the camera were in a single folder and in the order of label, front (XZ), left (YZ), next label, etc. A list of the print numbers in the order they are in the folder was manually created at the beginning of the script. The script then looped through the folder of images and renamed the images as *PrintNumber_Label*, *PrintNumber_XZ*, and *PrintNumber_YZ*. All the label images were consolidated into a single folder. The two face images were placed into their own folder named with the *PrintNumber*. After organization, these images were ready to feed into the image classification model.

Training Image Classifier

Transfer learning is a useful image classification technique that modifies a neural network pre-trained to classify a much larger data set. The first layers of the neural network are tuned and trained to extract basic features from an image (e.g., lines and textures) while the last layer performs the classification work. If the layer of the neural net is replaced with the new classifications, the model can be trained on a new dataset and achieve high accuracy while using fewer total data points. For this work, the MATLAB tutorial for using AlexNet for transfer learning was implemented to classify images with blob, delamination, warp, or “none” [8]. The classifications of each error can be subjective. Examples of each error type are shown in Figure 8. The image classification model was trained on ~430 segmented pictures of each error.

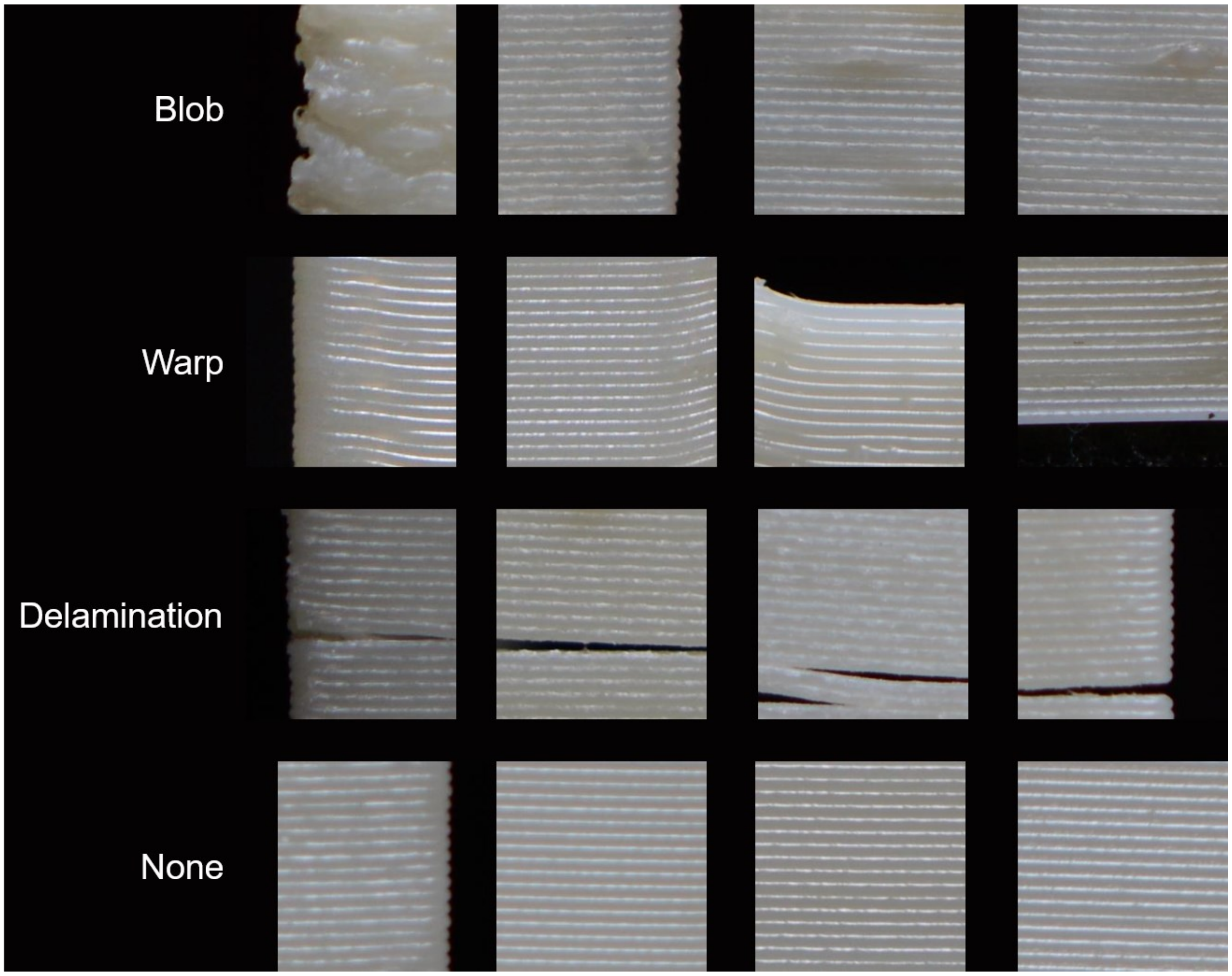


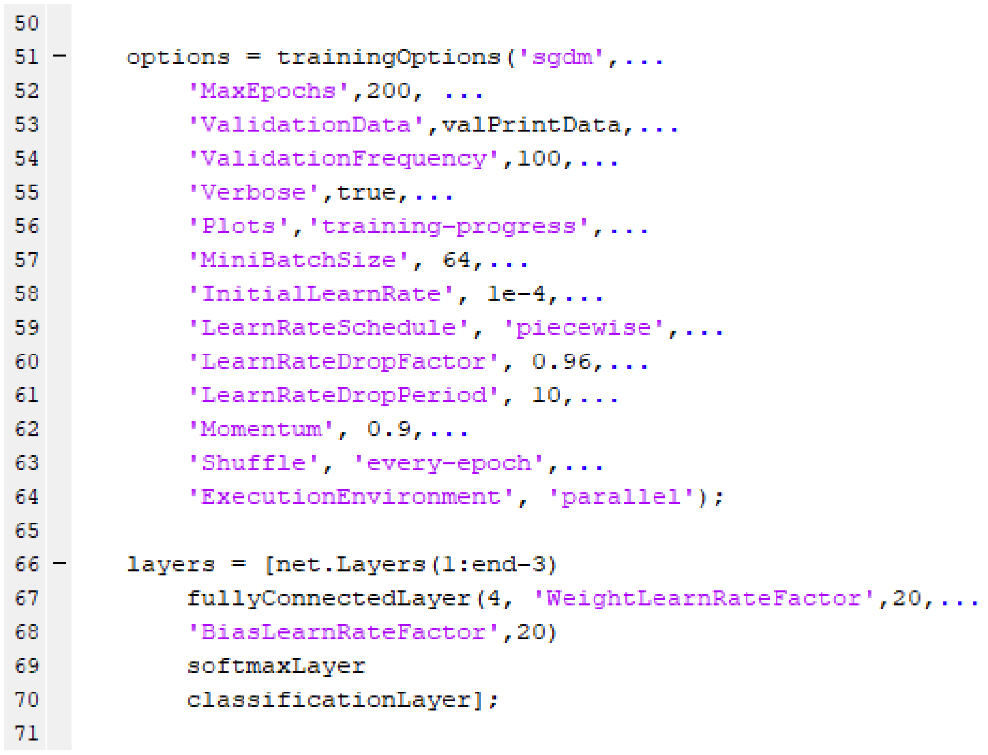
Figure 8. Examples of error types used for training the image classifier.

An image training dataset for the neural network to learn from was first created. To identify the precise location of errors in the print, images of printed walls were segmented into > 600 smaller squares representing 2-mm sections. These smaller images were hand classified by error type. Not all the error types occurred at the same rate with approximately 95% of the images classified as “none”. When an unbalanced image training dataset was used, the network learned to classify every image as “none” because it was achieving 95% accuracy. The image training dataset was rebuilt to include an equal number of each error type to correct this issue. As a result, the image training dataset was limited by the small number of delamination samples (200) with the total size of the image training dataset being 1000 images. However, even with this limitation, the model attained 93.8% training accuracy and 88% test accuracy.

A modified version of semi-supervised learning, an approach of using a model to classify and then using the results of that as new training data, was used to train the model. The modification included manually checking and reclassifying model results that were incorrect. Adjustments were also made to the image training dataset to make it more representative of real scenarios.

For example, the network trained with >98% test accuracy, but when used to classify a larger image dataset, often miscategorized blobs and “nones”. Adding examples of small blobs to the image training dataset and replacing poorly lit “nones” resulted in a model with 98.26% test accuracy that also showed high accuracy when manually checking the classifications of the image training dataset.

Training time of AlexNet was optimized using various techniques including increasing momentum from 0 to the default of 0.9 for stochastic gradient descent (SGD) and using Adam optimizer. The parameters used for trainingOptions and layers are shown in Figure 9. The Adam optimizer yielded the best results with a training time of 20 minutes. The final model was created and saved as “98.26_after_modifications.mat”. Results are shown in Figures 10 and 11. There was some error in the model caused by a lack of k-fold cross-validation, subjective errors during manual classification, and presence of additional categories not considered.



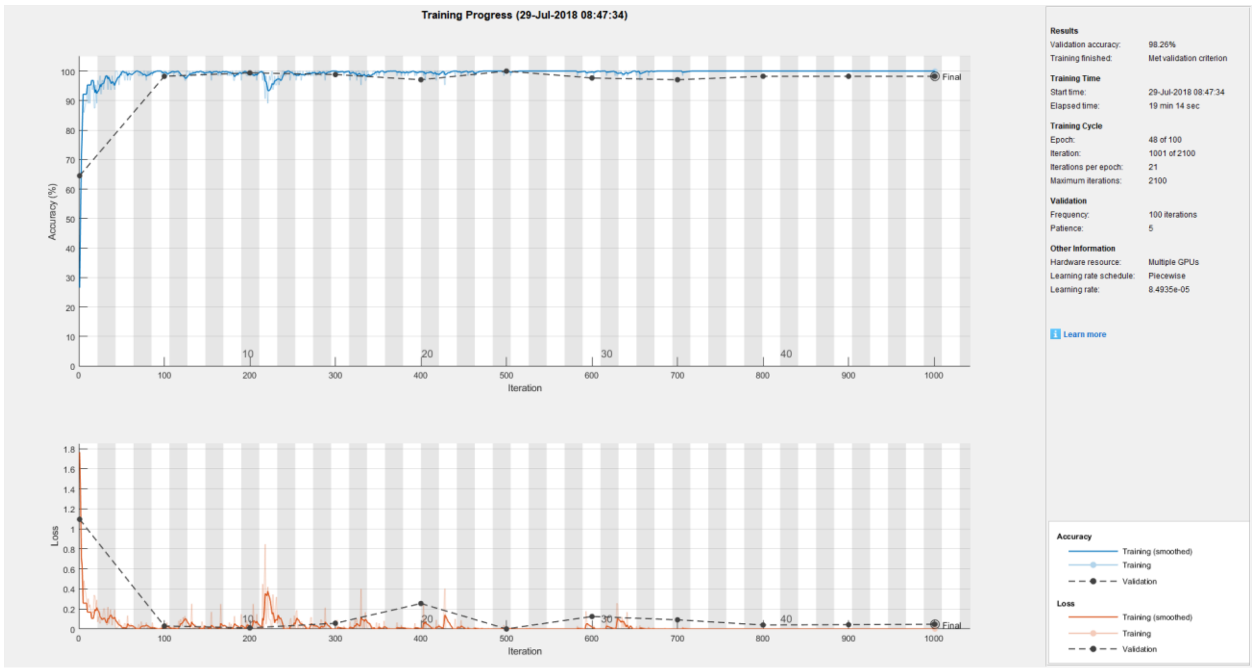
```

options = trainingOptions('sgdm',...
    'MaxEpochs',200, ...
    'ValidationData',valPrintData,...
    'ValidationFrequency',100,...
    'Verbose',true,...
    'Plots','training-progress',...
    'MiniBatchSize', 64,...
    'InitialLearnRate', 1e-4,...
    'LearnRateSchedule', 'piecewise',...
    'LearnRateDropFactor', 0.96,...
    'LearnRateDropPeriod', 10,...
    'Momentum', 0.9,...
    'Shuffle', 'every-epoch',...
    'ExecutionEnvironment', 'parallel');

layers = [net.Layers(1:end-3)
    fullyConnectedLayer(4, 'WeightLearnRateFactor',20,...
    'BiasLearnRateFactor',20)
    softmaxLayer
    classificationLayer];

```

Figure 9. The parameters used for trainingOptions and layers in the AlexNet MATLAB tutorial.



Results

Validation accuracy:	98.26%
Training finished:	Met validation criterion

Training Time

Start time:	29-Jul-2018 08:47:34
Elapsed time:	19 min 14 sec

Training Cycle

Epoch:	48 of 100
Iteration:	1001 of 2100
Iterations per epoch:	21
Maximum iterations:	2100

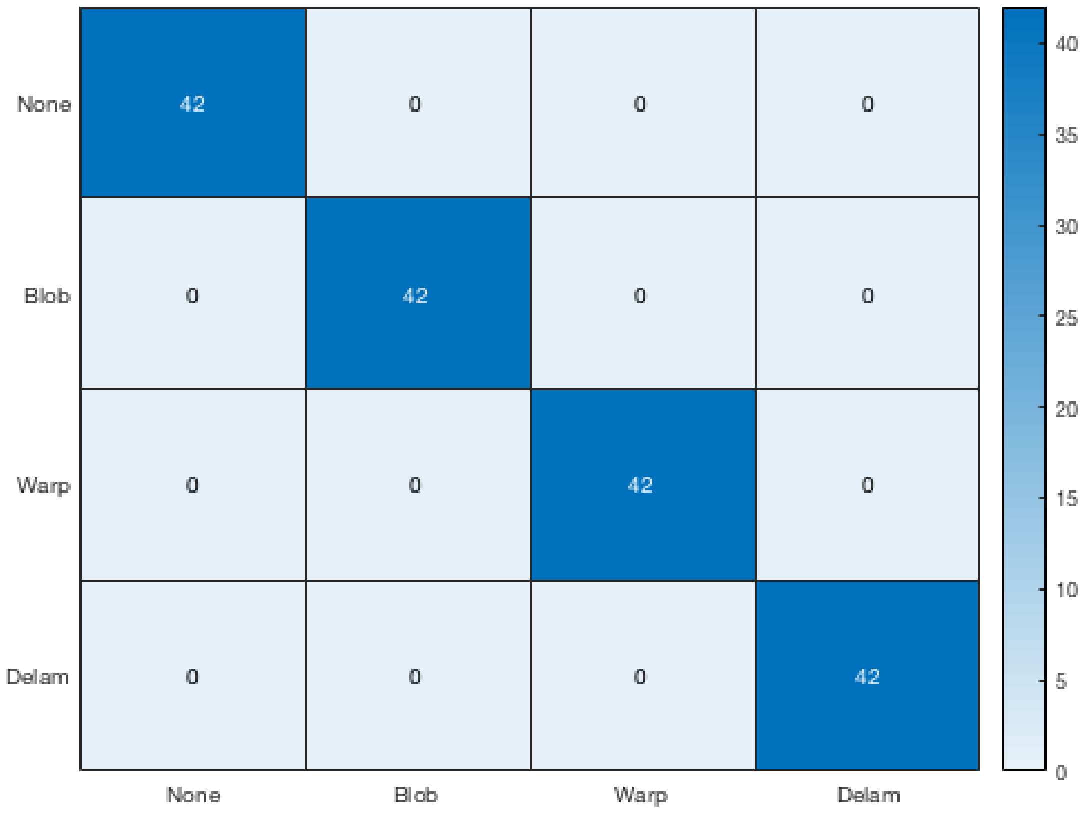
Validation

Frequency:	100 iterations
Patience:	5

Other Information

Hardware resource:	Multiple GPUs
Learning rate schedule:	Piecewise
Learning rate:	8.4935e-05

Figure 10. The image classifier achieved 98.26% accuracy after training.



	None	Blob	Warp	Delam
None	42	0	0	0
Blob	0	42	0	0
Warp	0	0	42	0
Delam	0	0	0	42

Figure 11. Image classifier confusion matrix showing 100% test accuracy.

Classifying Errors

Once the Image Classifier was trained and verified, a MATLAB script was used to sort through all the image data collected and classify errors in each. In operation, the script prompted the user to input both the folder of organized images and an output folder to save the segmented and classified images. The algorithm for this is as follows:

1. Use edge detection to identify the pixel location of the printed part in the image.
2. Segment the face into >600 square images.
3. Use the Image Classifier to identify the error type.
4. Save these classified images in a folder named for the error type.
5. Use the edge detection to identify the G-code points contained in the image.
6. Write these G-code points, print numbers, and error type probabilities to a text file (Table 1).

Table 1. Results of the image classifier that determined the location of the error and the error type probability.

gx	gy	gz	None	Blob	Delamination	Warp	Print Number
52.25	52.25	7.424	0.9984384	4.40E-06	0.0010829	0.0004744	1
52.25	52.25	7.674	0.9984384	4.40E-06	0.0010829	0.0004744	1
52.25	52.25	7.924	0.9984384	4.40E-06	0.0010829	0.0004744	1
52.25	52.25	8.174	0.9984384	4.40E-06	0.0010829	0.0004744	1
52.25	52.25	8.424	0.9984384	4.40E-06	0.0010829	0.0004744	1
52.25	52.25	8.674	0.9984384	4.40E-06	0.0010829	0.0004744	1
52.25	52.25	8.924	0.9984384	4.40E-06	0.0010829	0.0004744	1
52.25	52.25	9.174	0.9984384	4.40E-06	0.0010829	0.0004744	1
52.25	52.25	9.424	0.99953	5.00E-07	0.0000442	0.0004253	1
52.25	52.25	9.674	0.99953	5.00E-07	0.0000442	0.0004253	1

The results of running this image classifier on print 11 are shown in Figures 12-14. These figures are evidence that the classification of and mapping to the G-code points functioned correctly.

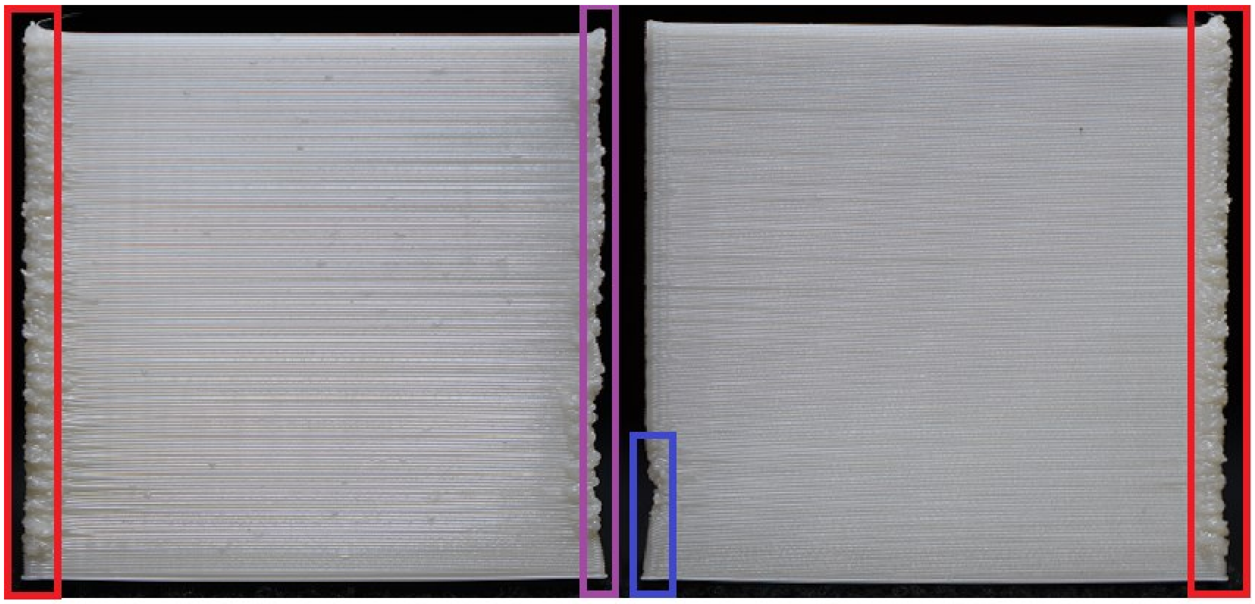


Figure 12. Resulting part from parameters selected for print 11. Boxes highlight areas where blobs are present. Colors coordinate with those found in Figure 13.

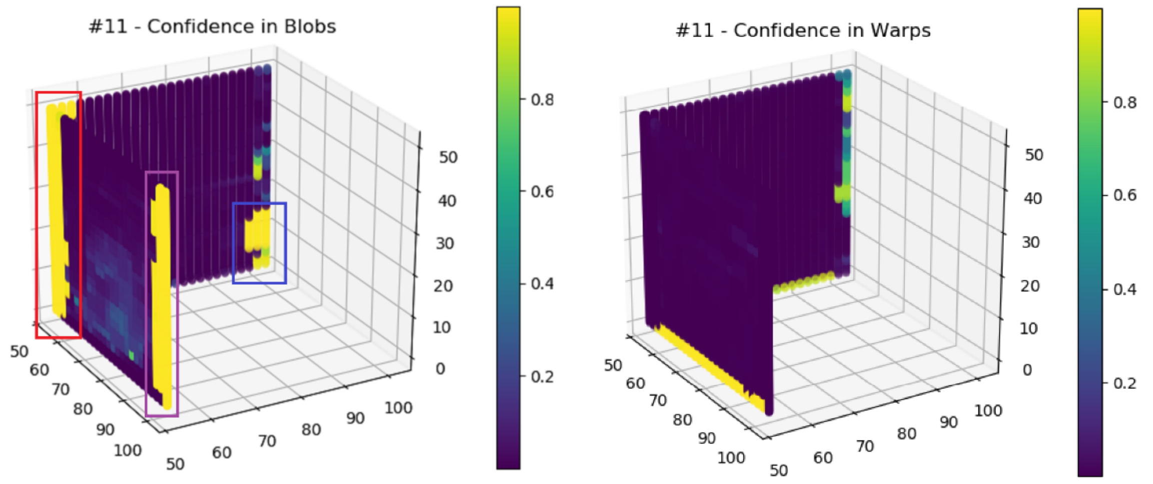


Figure 13. Confidence of error in blobs and warps identified in print 11 by the image classifier.

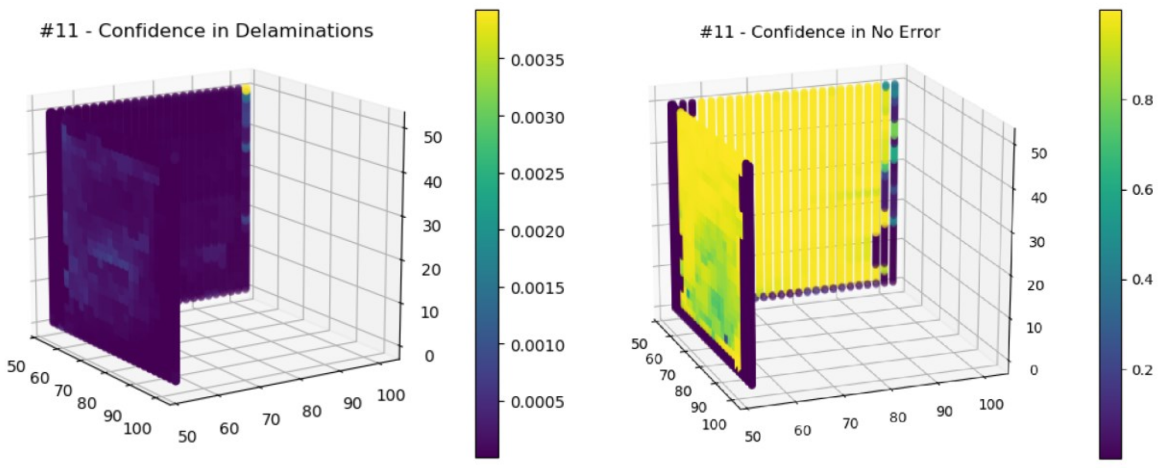


Figure 14. Confidence in delamination and "none" identified in print 11 by the image classifier.

Combining the Data

The data were output to a file that contained each G-code segment of the outer wall, their respective error probabilities, and the print number associated with those segments. This data was merged with the previously collected data (e.g., geometry, etc.) to create a master dataset that will be used in subsequent work. G-code segments from the bottom and inner wall were removed using an 'inner merge' in Pandas (a data analysis tool) since images were only collected for the outer walls.

Completed Master Dataset

Print Number and Segment Number were dropped to complete the master dataset. Steps to accomplish this included printing parts with varying parameters, segmenting the G-code to obtain a point cloud, calculating the geometric properties of the G-code points (through mapping to STL information), training an image classification neural network, using the image classification model on the printed parts, mapping the errors to the G-code points, and combining these data together into one master dataset.

Creating the Error Prediction Model

A model was created to determine of the optimum printing parameters for each point on the part using the master dataset. There were many design options for this model including multi-output, single-output, regression, or classification models. This section discusses the model types explored and trail-and-error process used to select the most appropriate model.

Selecting Model Output

One approach to the problem explored was using a model to predict multiple values simultaneously in what is known as a Multi-Output, Multi-Target, Multi-Variate, or Multi-Response regression. This approach is more complicated than fitting a separate model to each output because there are expected relationships between the outputs. The machine learning techniques best used for this approach are support vector regression, regression trees and neural networks. A review by Borchani outlines this problem and recent attempts to improve the techniques [9].

The Python machine learning library Scikit-learn has a few models that natively support multi-output regression:

- neighbors.KNeighborsRegressor
- neighbors.RadiusNeighborsRegressor
- tree.DecisionTreeRegressor
- tree.ExtraTreeRegressor
- ensemble.ExtraTreesRegressor
- ensemble.RandomForestRegressor
- neural_network.MLPRegressor

A machine learning model that would intake XYZ and geometric properties and predict the best printing parameters (e.g., print speed, temperature, etc.) for a given point would be a multi-output model. This approach was not selected due to the difficulties in training a multi-output model and the output of only one good parameter combination per point. The predicted parameter changes from segment to segment may not be possible with the physical limitations of the printer.

The problem was simplified to avoid the issue discussed above by recasting it. XYZ, geometric properties, and printing parameters were used to predict the error at a given point, instead of using the model to predict the print parameters. This simplification resulted in a single-output model for which there were various Scikit-learn packages available. This approach assessed all possible parameters combinations for a particular point. The predicted errors for parameter combinations at each point would then guide selection of the optimal print parameters at that point. For example, the model would yield different probabilities for “none” output for a given point (X=1, Y=1, Z=1) with different printing parameter combinations (Table 2). The table shows the best predicted printing parameters as the first combination, but the second combination may also print well. The option of selecting between multiple parameter combinations that fall above a threshold prediction enabled parameter selections that were within the physical limitations of the printer.

Table 2. Probability of "none" error using different parameter combinations.

Nozzle Temp	**Bed Temp**	**Print Speed**	**Extrusion Multiplier**	**Fan Speed (%)**	**Probability of “none” Error**
245	120	20	110	80	0.98
245	110	40	110	80	0.90
235	120	60	110	45	0.65
225	110	60	100	45	0.54
245	100	60	110	30	0.12
245	110	40	100	80	0.00

Models/Techniques Attempted

Since the image classifier provided the probabilities and error type, the problem could be posed as a multi-output regression (predict all the error type probabilities as an output), single-output regression (predict the probability of a “none” type error), or a type of classification (binary being error or no error or multi-class to predict which of all the errors).

The models tested are listed below:

- Decision Tree Classifiers
- Gradient Boosting Classifier
- K-Nearest Neighbors
- MLP Classifier
- Naive Bayes Classifier
- Quadratic Discriminant Analysis
- Random Forest Classifier
- Support Vector Classifier
- Gaussian Process Classifier
- Decision Tree Regressor
- MLP Regressor
- Gradient Boosting Regressor

Various approaches using these models were also attempted including:

- Balancing the classes (up, down, and mixed sampling)
- Not balancing the classes
- Using models while supplying prior information (the ratio of each of the classes in the master dataset)
- Error vs. No Error Binary Classification
- Multi-Class classification
- Ensemble Regression Chaining

The resulting best model was the Gradient Boosting Classifier.

Gradient Boosting Classifier

This model was trained by down-sampling the data so each error type was represented equally in the data. This was done with care because the prediction accuracy could be artificially inflated if information of a single print was in both the training and testing data. After balancing the classes, the final model used was trained with 12,291 data points of each error type. The amount of data used to train the model was significantly smaller than the amount of data available, due to the limiting factor of delaminations in the data set. The final model dropped prints that contained points with 121% Extrusion Multiplier (discussed below): prints 1, 2, 3, and 167. All the default parameters for the model were used since their performance was adequate. In the future, the parameters could be tweaked to obtain a higher accuracy model, however care must be taken to avoid overfitting. Note that typical models are trained with 5- or 10-fold cross validation to determine the generalizability of the model to outside data. This was not done here because of the practical limitations of performing this while down-sampling the data by hand. However, cross validation can and should be done.

False positive and false negative rates that occur in our model are shown in a confusion matrix (Figure 15). A false positive occurs when the model predicts there to be an error when there is “none”. A false negative occurs when the model predicts no error, but one is present. For this model:

- ~24.2% of the true “none” were predicted to be an error
- ~2.87% of predicted “none” were an error (from image classification model)

Thus, the model was much more likely to assign errors where there are none than vice versa.

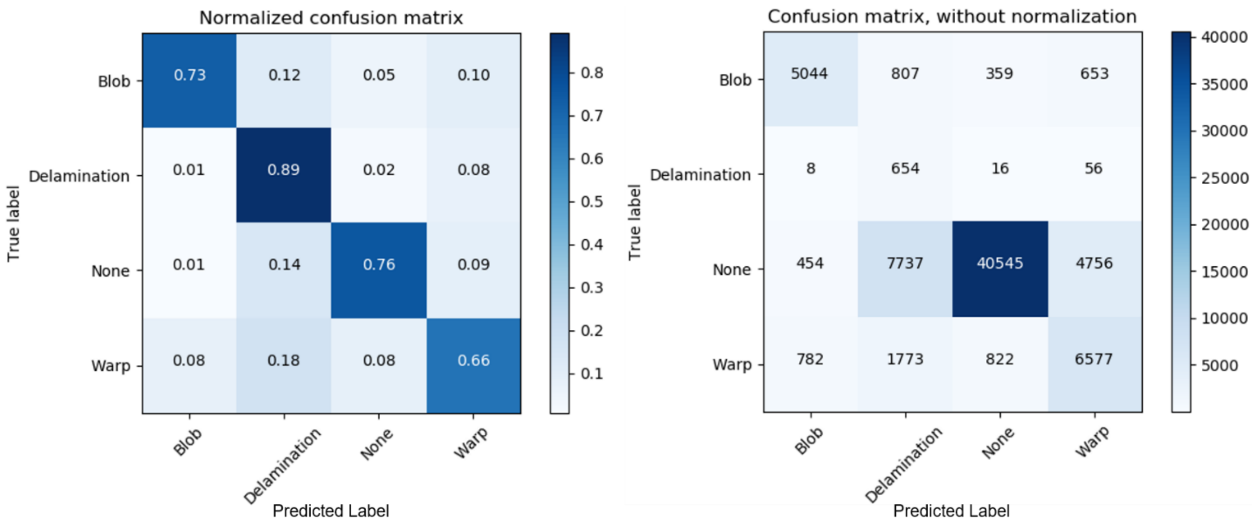
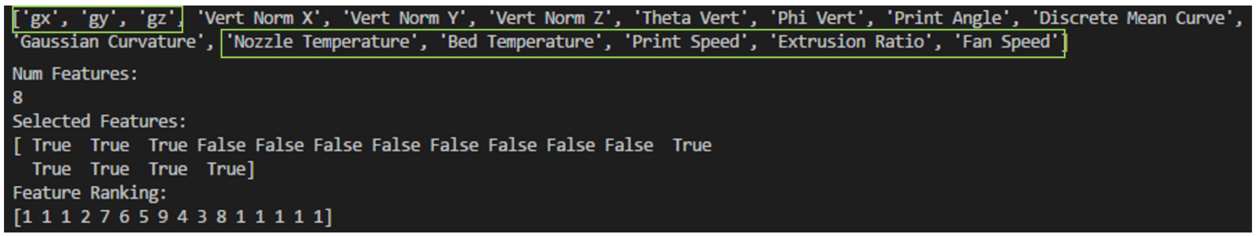


Figure 15. Confusion matrices for the error prediction model.

Caveats

This model was a proof of concept that was trained on a very narrow range of XYZ and geometric properties. Therefore, the model, as trained, should only be used for the three sides of a cube part and is not generalizable to other geometries. This assertion is supported by the results of feature ranking using recursive feature elimination (RFE). This method assigned the importance of the features used to train a model. The default was to select half of the features by assigning them an importance of 1. Performing this with the Gradient Boosting Classifier yielded the results shown in Figure 16. The most important features during model training were the XYZ and printing parameters. This was probably due to the geometric properties being intrinsic to the XYZ since only one shape was used for training. With a model trained on more shapes, future predictions may be applicable to more geometries.



```
['gx', 'gy', 'gz', 'Vert Norm X', 'Vert Norm Y', 'Vert Norm Z', 'Theta Vert', 'Phi Vert', 'Print Angle', 'Discrete Mean Curve',
'Gaussian Curvature', 'Nozzle Temperature', 'Bed Temperature', 'Print Speed', 'Extrusion Ratio', 'Fan Speed']
Num Features:
8
Selected Features:
[ True  True  True False False False False False False False False  True
  True  True  True  True]
Feature Ranking:
[1 1 1 2 7 6 5 9 4 3 8 1 1 1 1 1]
```

Figure 16. Results of using recursive feature elimination with the gradient boosting classifier.

Verifying the Prediction Model

Several experiments were performed to verify the behavior of the model. These include predicting errors in future prints (ones that the prediction model had not seen), forcing each of the error types, optimizing for print speed while reducing errors, and printing the highest predicted quality print. The experiments are detailed in the next section.

Getting the Predictions

Each experiment required the use of all 10,149 points and 432 parameter combinations (there are more than 324 because Fan Speed 0 was included in the training data). A Python script was used to load in the points, do a Cartesian join with the possible parameter combinations, normalize the data, and send it through the model.

Predicting Future Prints

Four prints were selected at random that were not included in the training data. The error predictions for these prints were obtained from the model. After printing, each was photographed and run through the Image Classifier to obtain the "true" errors at each segment. This could then be compared with the error prediction model output for accuracy.

The prints chosen at random were:

- Print 161:
    - Nozzle temperature: 235°C
    - Bed temperature: 100°C
    - Print speed: 60 mm/s
    - Extrusion multiplier: 110%
    - Fan speed: 80%
- Print 173
    - Nozzle temperature: 245°C
    - Bed temperature: 120°C
    - Print speed: 60 mm/s
    - Extrusion multiplier: 100%
    - Fan speed: 30%
- Print 239
    - Nozzle temperature: 245°C
    - Bed temperature: 100°C
    - Print speed: 40 mm/s
    - Extrusion multiplier: 100%
    - Fan speed: 30%
- Print 274
    - Nozzle temperature: 245°C
    - Bed temperature: 100°C
    - Print speed: 20 mm/s
    - Extrusion multiplier: 90%
    - Fan speed: 45%

The accuracy score for the prediction was determined by calculating how many of the XYZ errors were accurately predicted compared to the image classification.

The predicted errors of each print, the associated photographs, and the AlexNet predictions are shown in Figures 18-25. The key for the error prediction and image classification images is shown in Figure 17. The average accuracy of these four prints is 79.83%. With more prints, this accuracy is expected to fall to the 73.69%. The model was the least accurate at predicting prints with a 20 mm/s print speed. This is likely due to both the randomness in delaminations compared with other errors, as well as the tendency of the Image Classifier to incorrectly classify delaminations. These two, in combination, meant the model made correlations between delaminations and low print speed, but was less accurate at predicting where they occur.

🟪 Blob 🟨 Warp 🟦 Delamination 🟩 None

Figure 17. Color code used for different error types.

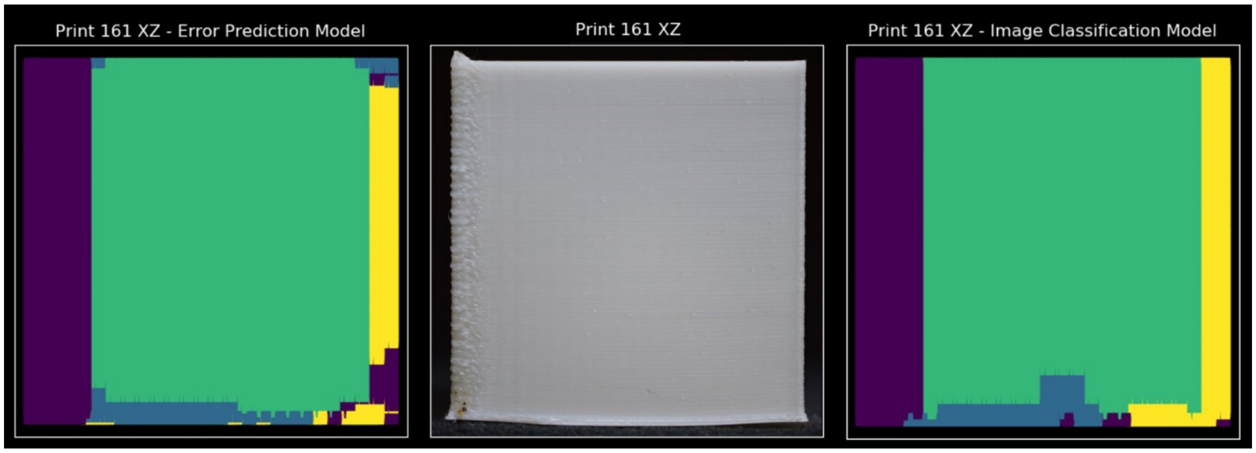


Figure 18. Error prediction model (left), printed part (center), and image classifier errors on XZ face for print 161.

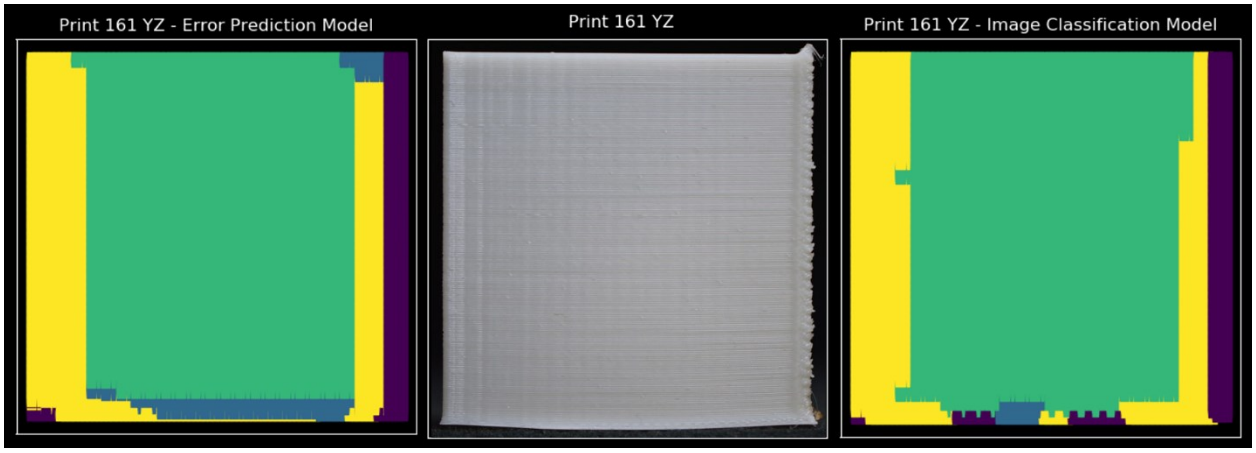


Figure 19. Error prediction model (left), printed part (center), and image classifier errors on YZ face for print 161.

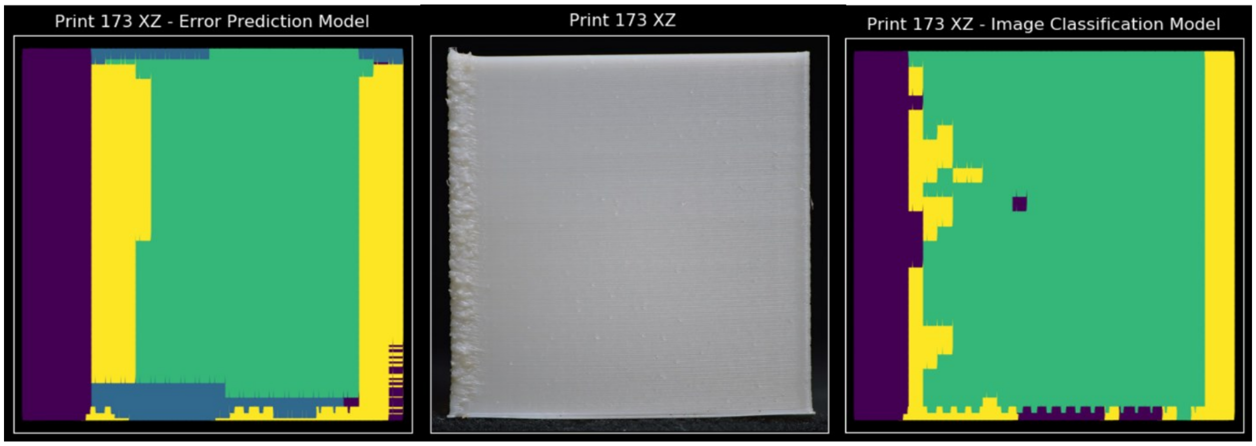


Figure 20. Error prediction model (left), printed part (center), and image classifier errors on XZ face for print 173.

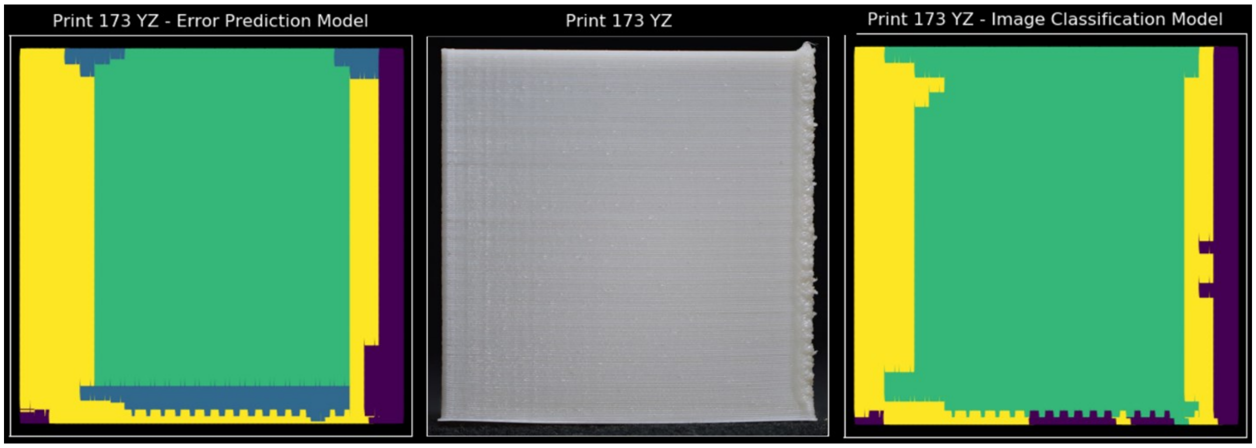


Figure 21. Error prediction model (left), printed part (center), and image classifier errors on YZ face for print 173.

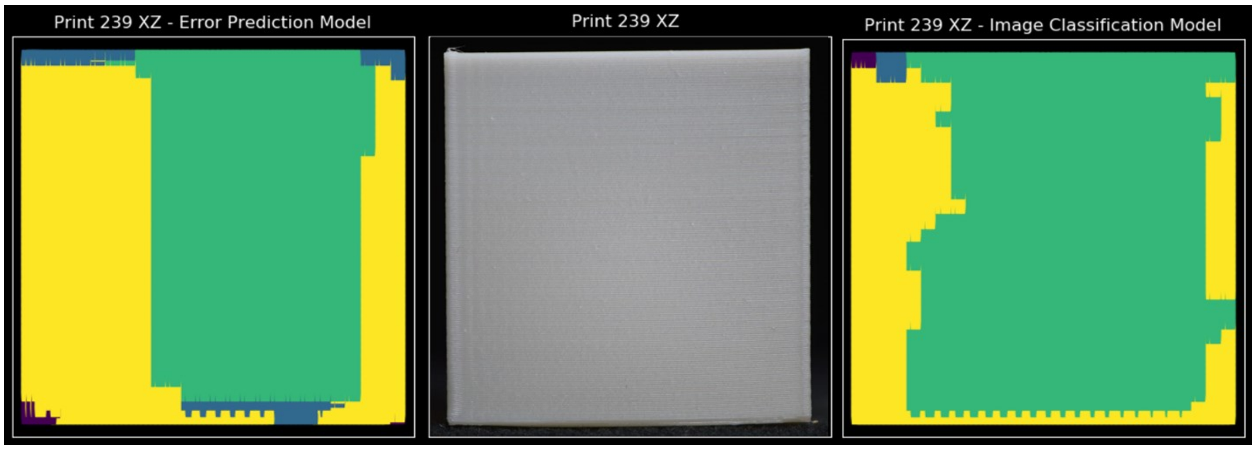


Figure 22. Error prediction model (left), printed part (center), and image classifier errors on XZ face for print 239.

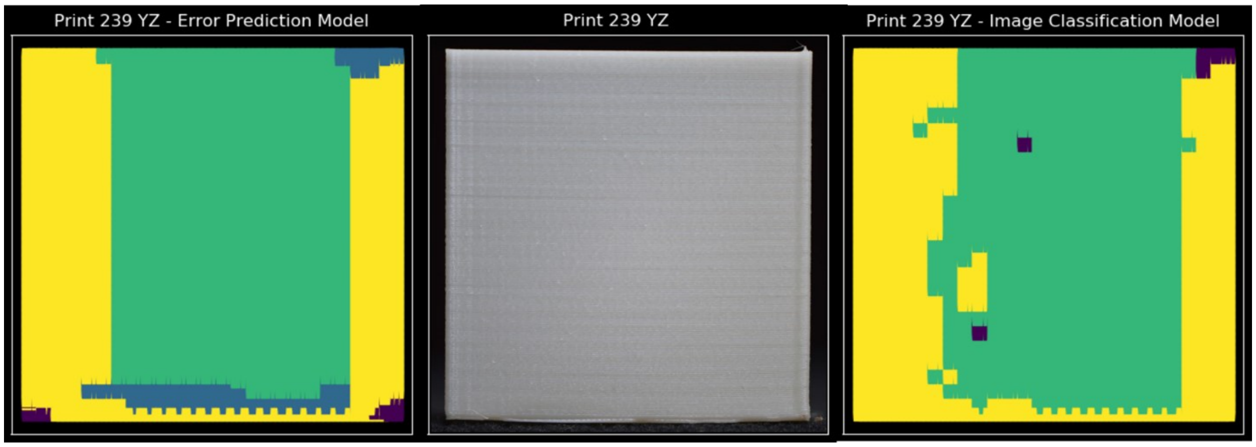


Figure 23. Error prediction model (left), printed part (center), and image classifier errors on YZ face for print 239.

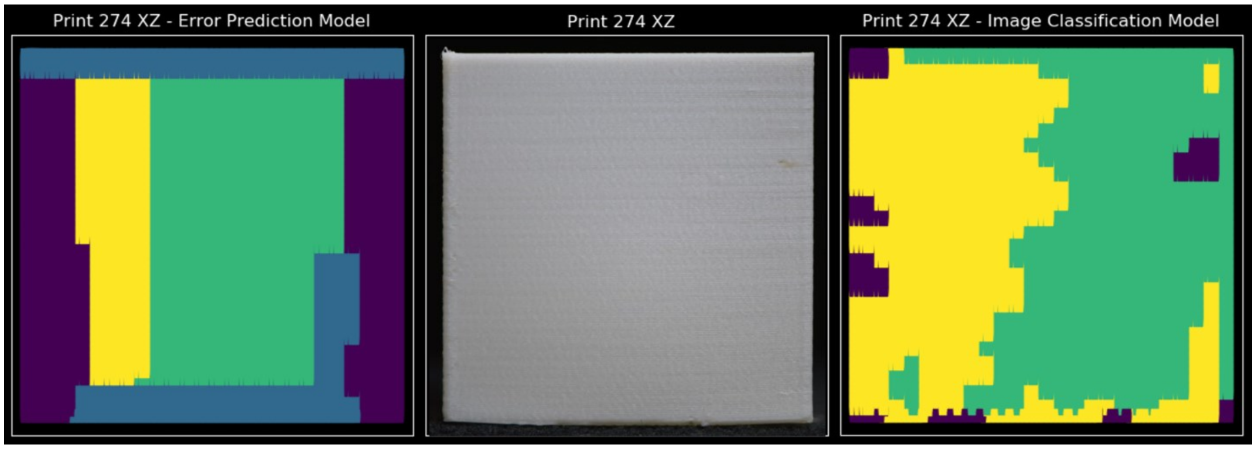


Figure 24. Error prediction model (left), printed part (center), and image classifier errors on XZ face for print 274.

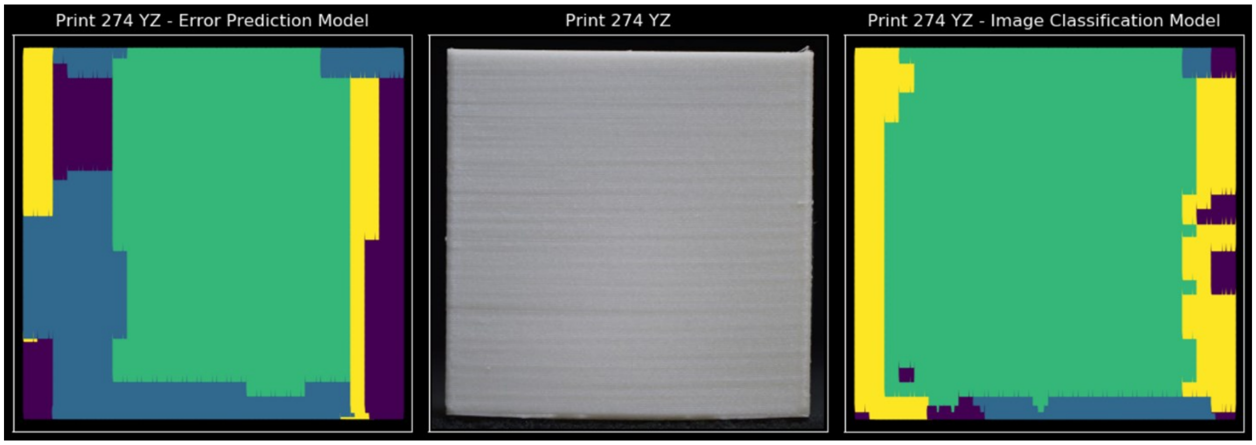


Figure 25. Error prediction model (left), printed part (center), and image classifier errors on YZ face for print 274.

Optimizing Prints Using Predicted Errors

The tool was tested for optimization of various parameters including quality, speed, and specific errors. In all the optimizations below, the bed temperature and nozzle temperatures were held constant due to the global effect of bed temperature and physical limitations of nozzle heating. It takes ~42 s to ramp up 10 ºC under the worst printing conditions for nozzle heating (Figure 26). At the slowest printing conditions, 20 mm/s, one face layer was completed in 2.5 s. A time delay from segment to segment for even a second drastically affected the print quality. However, it was outside the capabilities of the printer to change the nozzle temperature within a single layer. Holding the nozzle temperature constant throughout the print did not lower the probability of “none” error (Figure 27). In the case of the optimizing for speed, there was no loss in the print speeds predicted (Figure 28).

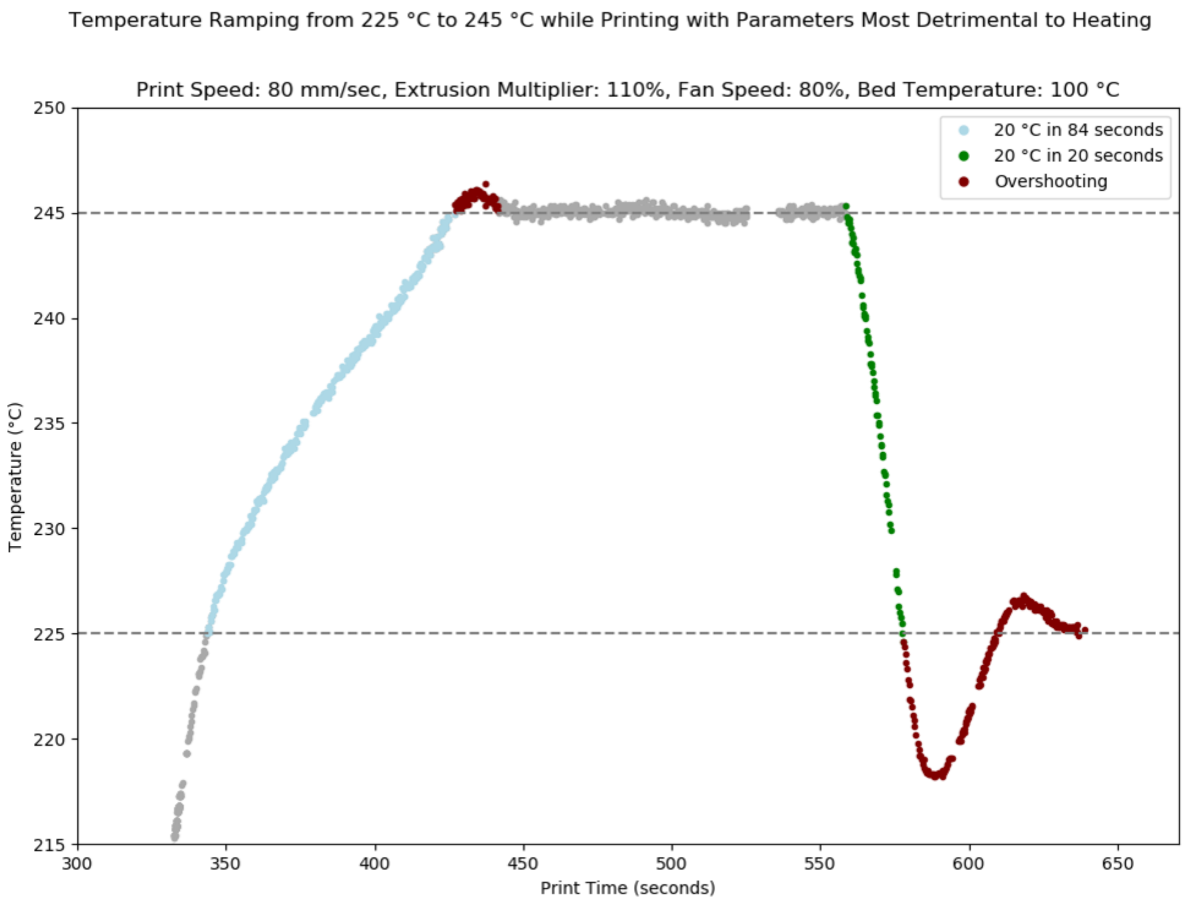


Figure 26. Thermal response of the nozzle to heat input while printing under worst-case parameters.

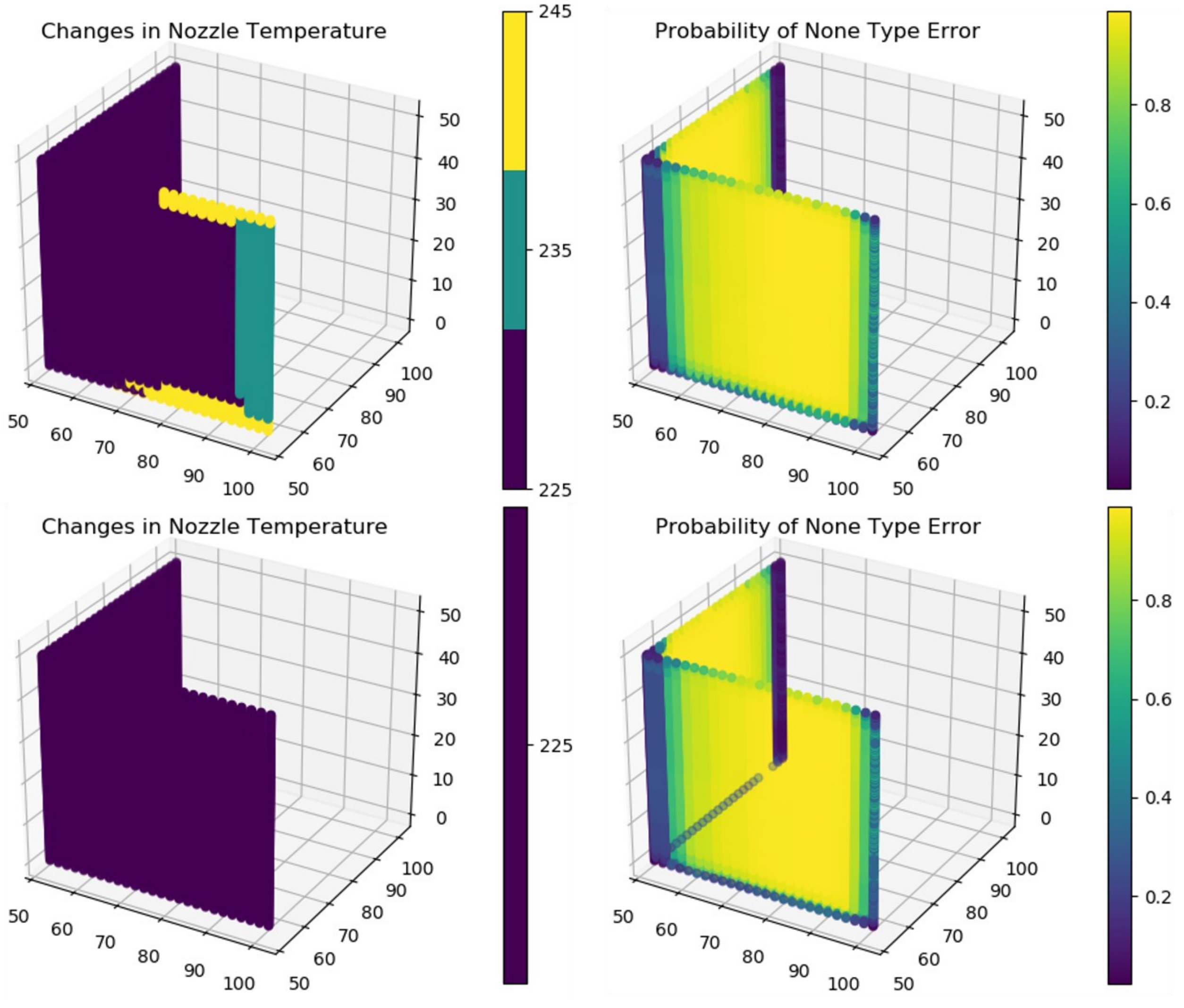


Figure 27. Results of the tool showing that holding the nozzle temperature constant does not reduce the probability of “none” error when optimizing for quality.

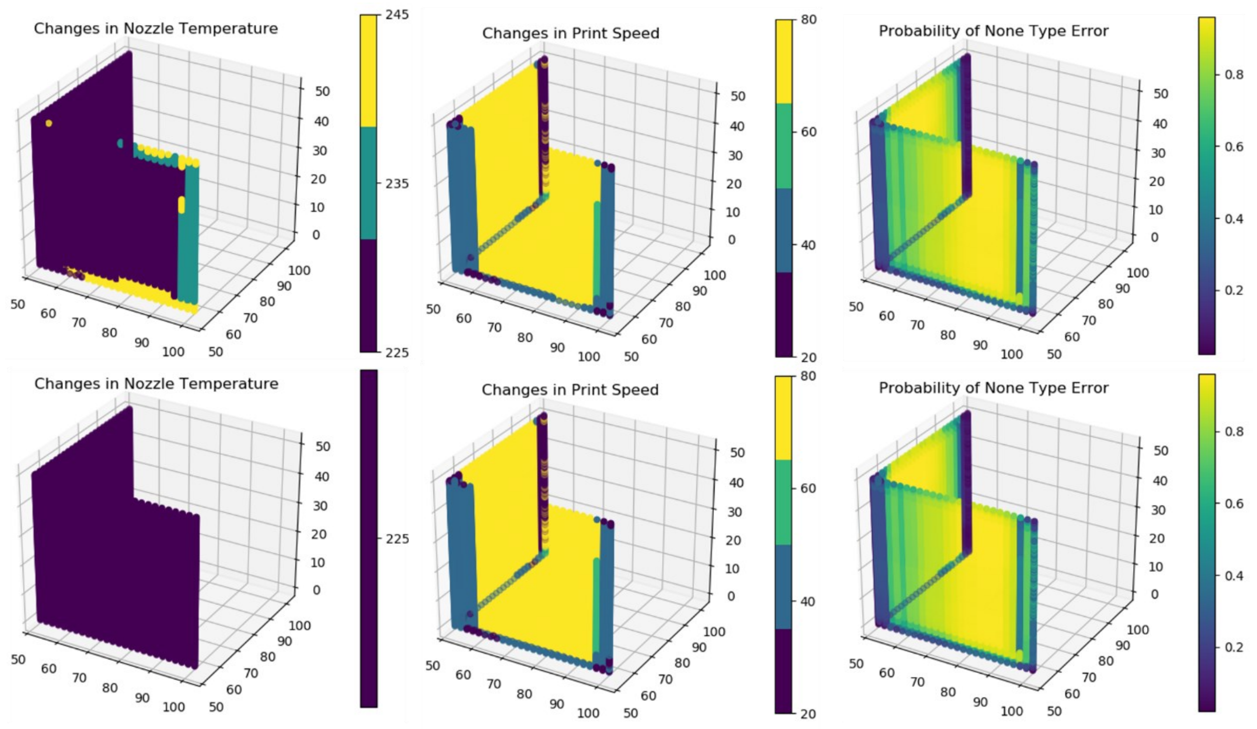


Figure 28. Results of the tool showing that holding the nozzle temperature constant does not reduce the probability of “none” error when optimizing for speed.

These optimizations used a simple approach for selecting parameters (smoothing). In the future, more advanced optimizations may provide a better print and allow for the inclusion of advanced printer capabilities.

Optimizations Tested

Quality

The best overall bed temperature and nozzle temperature were first selected. “Best” was determined by finding the bed temperature and nozzle temperature combination that contained the most “none” predictions of the 10,149 points. Points where “none” error was not the highest ranked probability for any combination of parameters at the selected bed and nozzle temperature had their print speed, extrusion ratio, and fan speed selected for the lowest probability of blobs. Points that did not fit these criteria had the remaining parameters selected for highest probability of “none”. The selected parameters for quality optimization and probability of “none” error using those parameters are shown in Figure 29.

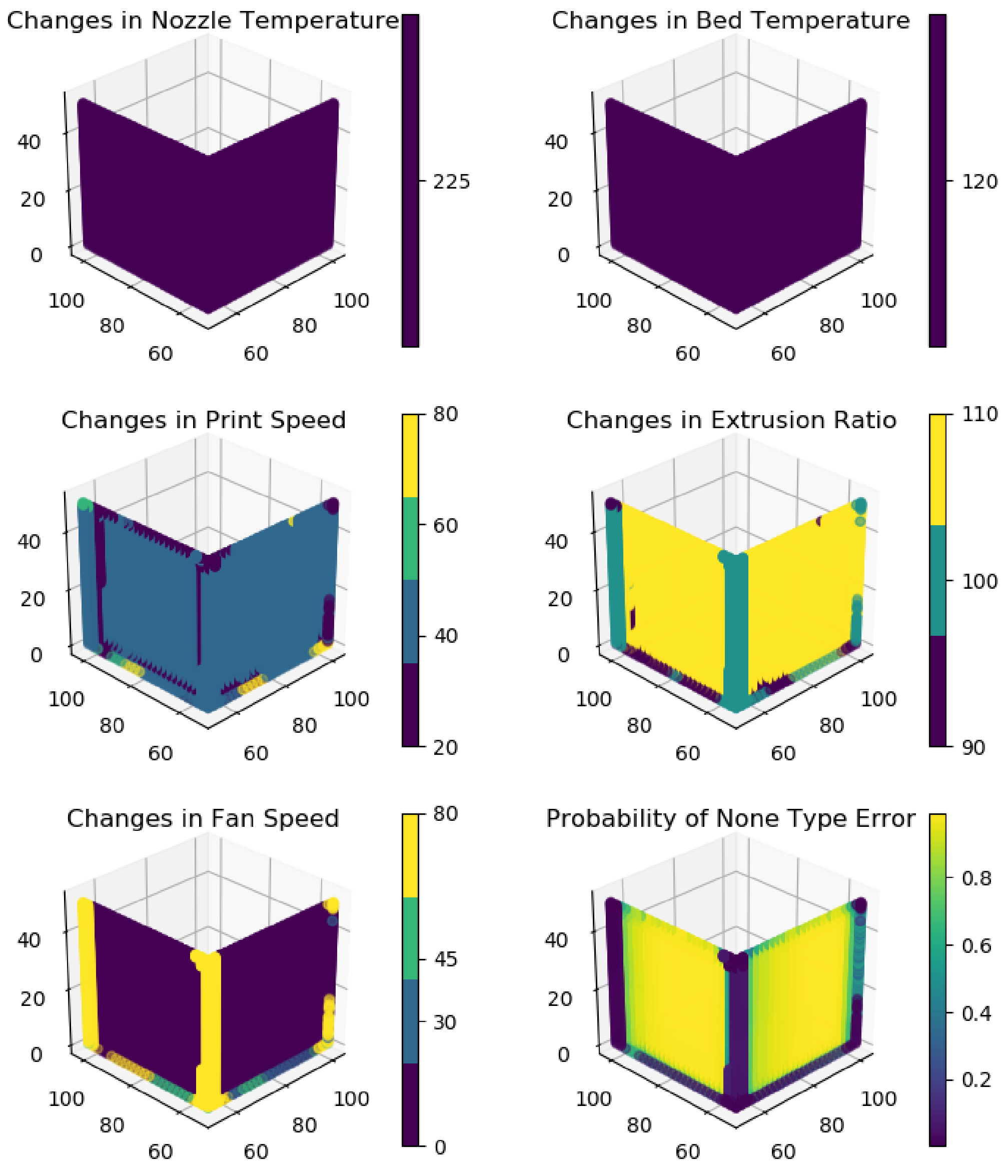


Figure 29. Selected parameters and probability of “none” error when optimizing for quality.

Speed

To optimize for speed, the best overall bed temperature and nozzle temperature with a print speed of 80 mm/s were found. As before, “best” was determined by finding the bed temperature and nozzle temperature combination that contained the most “none” predictions of the 10,149 points. From this, all points where “none” was the most likely error were grabbed with fan speed and extrusion ratio selected to maximize the probability of “none” error. Other points where no combination of the remaining parameters resulted in “none” being the most common error underwent the same procedure at 60 mm/s. If “none” error was still not the most likely, the speed was again reduced, and the same procedure performed. Points where no combination of parameters resulted in a most likely error of “none” had print speed, extrusion ratio, and fan speed selected for the highest probability of “none”. The locations and parameters chosen to minimize the probability of blobs are shown in Figure 30. The selected parameters and probability of “none” error are shown in Figure 31.

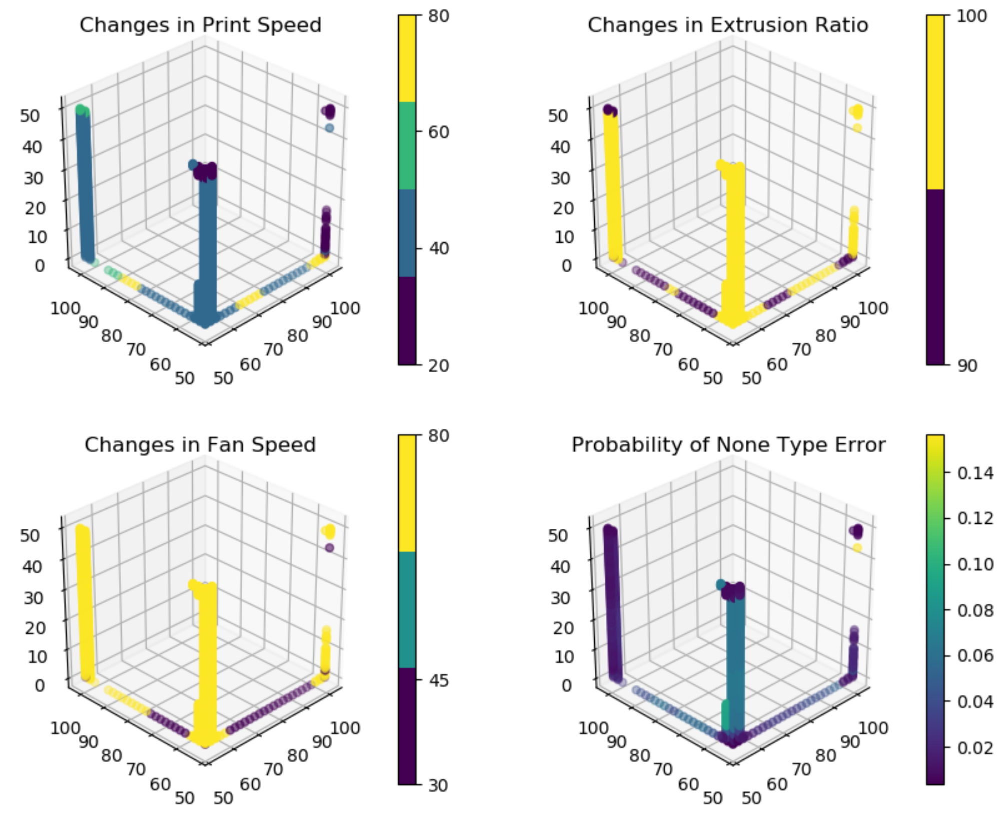


Figure 30. Parameters chosen to reduce the probability of blobs in locations when no combination of parameters yields a most likely error of “none”.

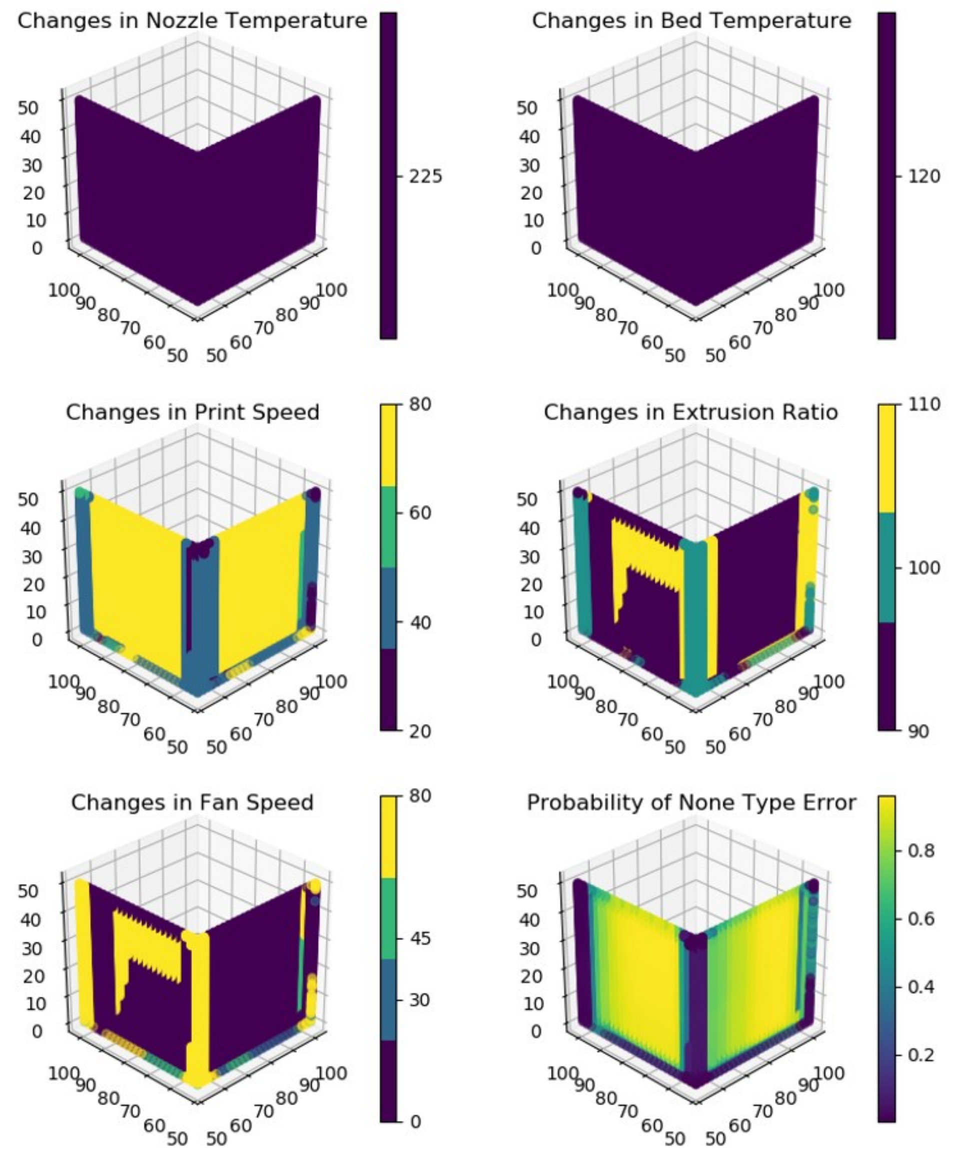


Figure 31. Selected parameters and probability of “none” error when optimizing for speed.

Forcing Errors

To force errors, the highest probability of blobs, warps, or delaminations is selected at each point.

Smoothing

Randomly varying Extrusion Ratio and Print Speed every 2 mm revealed physical limitations of the system (Figure 32). Changes to print speed and extrusion multiplier compromised the printing quality as observed in the optimization tests shown in Figure 33. Here, clear demarcations are present where print speed and extrusion multiplier have changed during the print. Note that print speed and extrusion multiplier tend to change together. The printing path is from the left to the right in the picture of the printed part. As print speed and extrusion multiplier decreased, there was an over-extrusion of material for roughly 1 cm (5 segments). Near the right edge of this printed face, print speed and extrusion multiplier were increased. Here, an under-extrusion of material for roughly 1 cm exists.

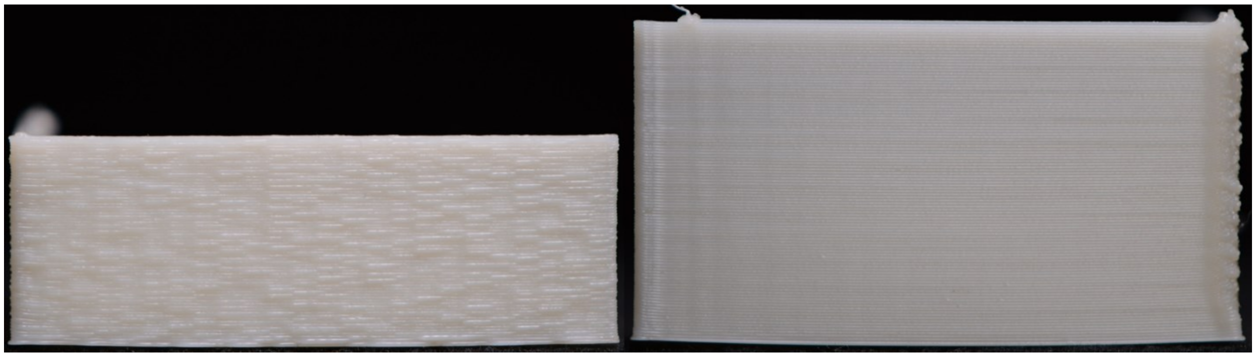


Figure 32. Part printed with randomized print speeds (left) and extrusion multiplier (right).

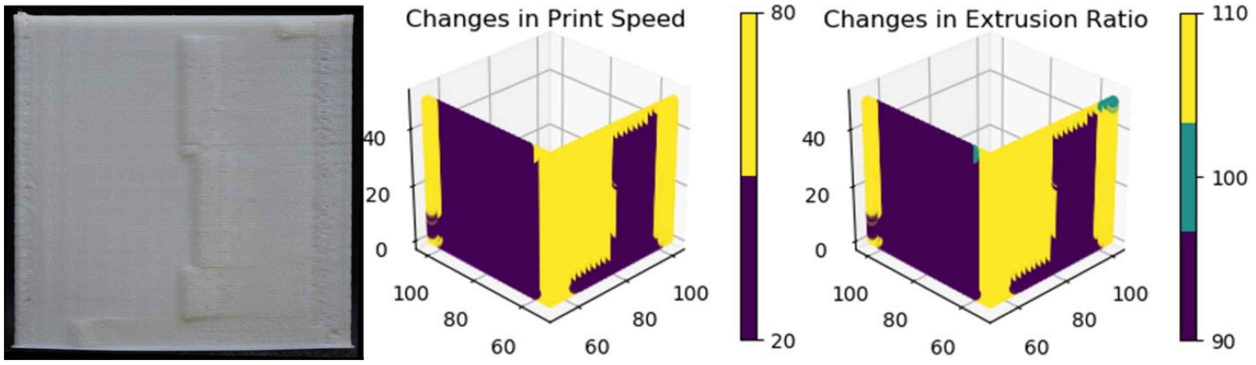


Figure 33. Printed part with changes to print speed and extrusion multiplier without any smoothing techniques implemented.

A linear smoothing technique was developed to overcome these issues. This code looks at a window of segments for a single parameter; if the last value is different from the second to last value, the whole window takes incremental steps from the first value of the window to the last. So, for a window size of 5:

$$[20, 20, 20, 20, 80] \rightarrow [20, 35, 50, 65, 80]$$

Future work could improve the smoothing algorithm; however, the performance was adequate as seen in Figure 34. Smoothing was applied to print speed, extrusion multiplier, and fan speed for all print optimizations. Note that all fan speed values less than 30 will be realized as Fan Speed 0 due to the physical limitations of the fan. The results of applying the smoothing algorithm to the selected parameters for optimizing speed and quality are shown for a window size of 10 (Figure 35) and a window size of 7 (Figure 36).

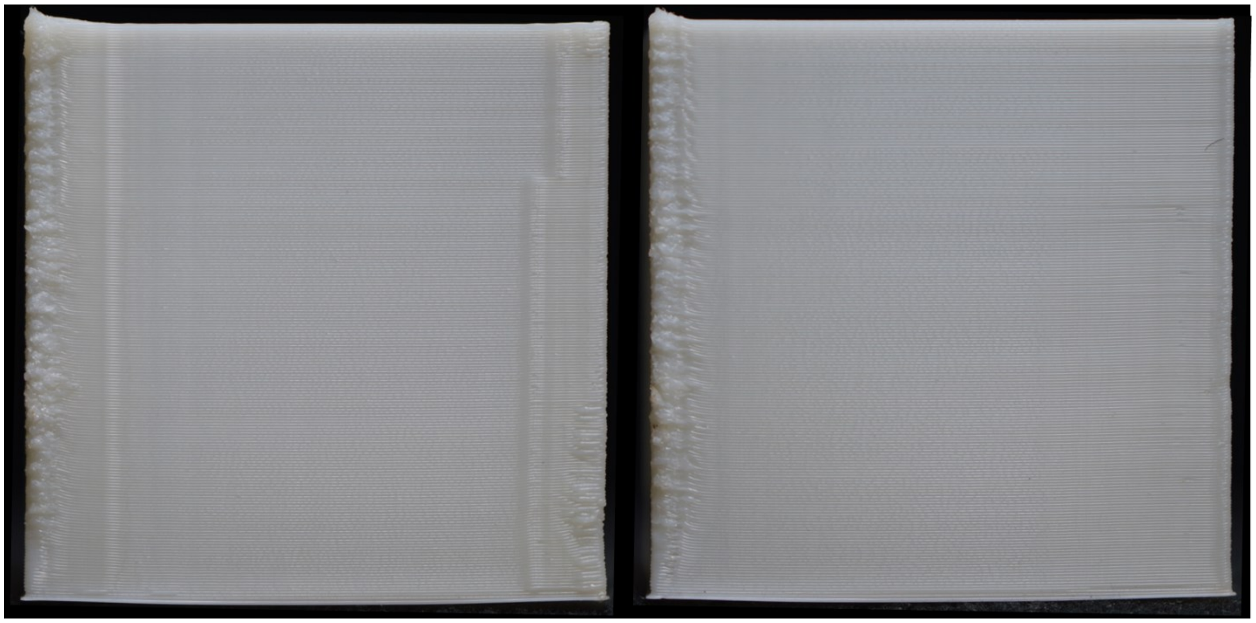


Figure 34. Parts printed with a window size 5 (left) and window size 7 (right) smoothing algorithm implemented.

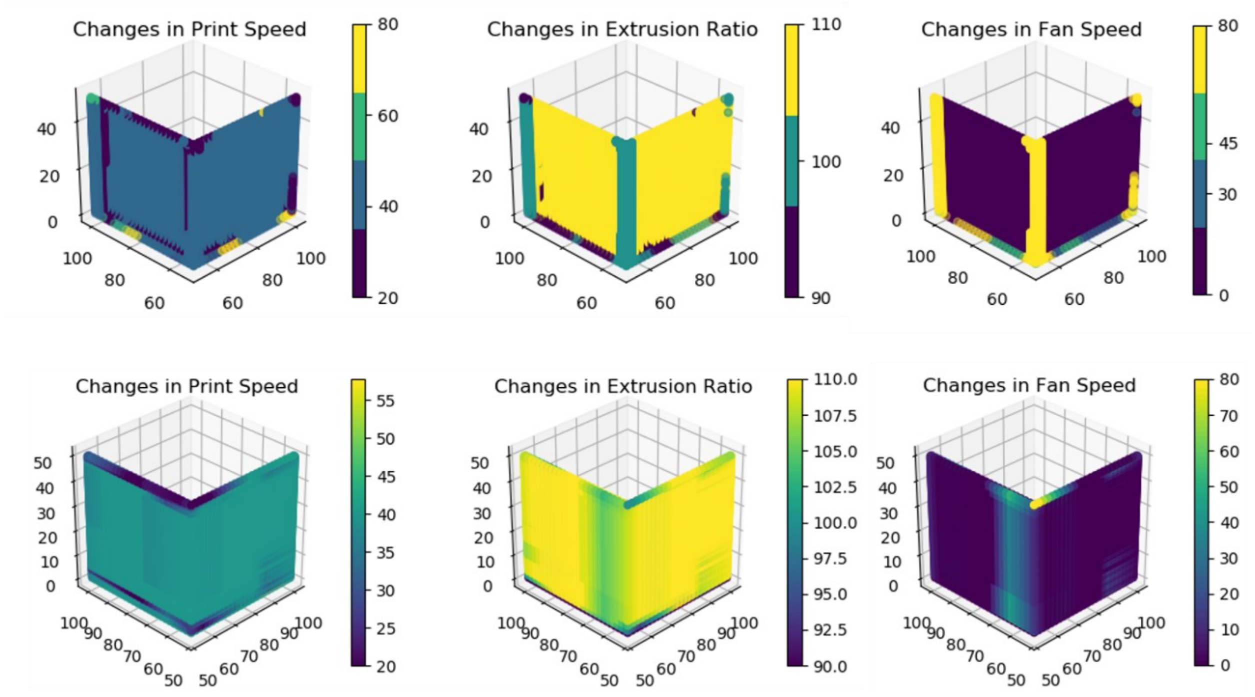


Figure 35. Predictions for quality optimized part (top) and smoothed predictions using window size 10 (bottom).

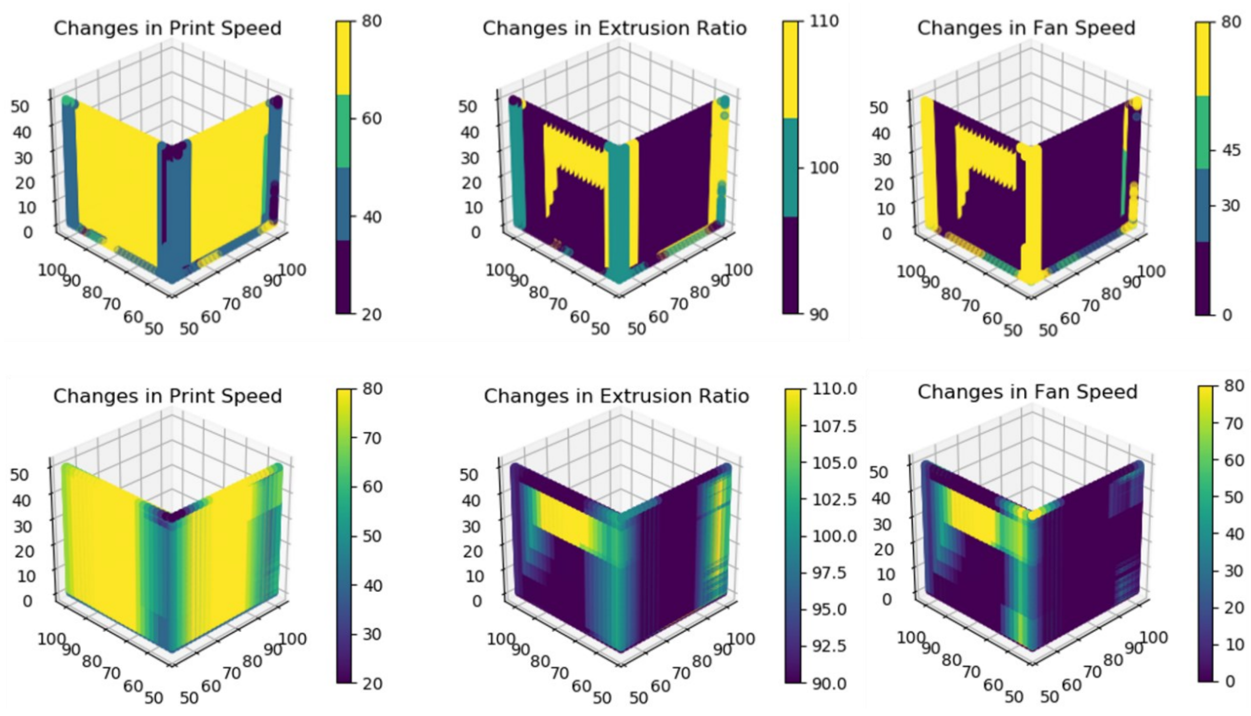


Figure 36. Predictions for speed optimized part (top) and smoothed predictions using window size 7 (bottom).

Communicating to G-code

Outer Points to All G-code Points

The optimal printing parameters for the outer walls were used for the entire print due to the limitations with capturing errors on the inner walls. To associate the outer wall segments with the inner wall segments, a k-d tree was used to find the nearest outer wall segment to each inner wall segment. Then information about printing parameters and the associated G-code segments were merged using Pandas. For the bottom three layers, parameters were generalized based on the most common optimal parameter combinations predicted for those areas. Before and after images of the bottom layer optimization are shown in Figure 37. These pictures show that the algorithm worked as designed, providing a way to generalize the predictions of the tool on the outer wall to every segment seen in the G-code. Using Pandas to merge the data preserved the original order of G-code segments in the print and made communicating the parameter combinations to G-code simpler.

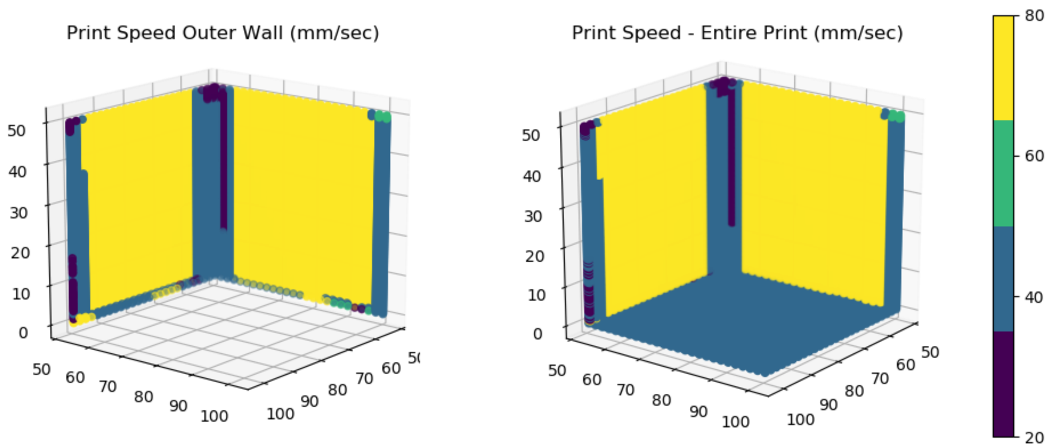


Figure 37. Results from using code to optimize bottom layer parameters.

Communicate to G-code

Next, the printing parameters for each segment needed to be communicated to the printer via G-code. A script was created to intake the original G-code, cut up the lines, and rewrite the segmented G-code with the new printing parameters for each G-code segment. To run the script, a mapped, segmented G-code from the above sections was placed into a folder and the path to that folder given to the script. The script wrote the new, optimized G-code for all the files in that folder.

Optimization Results

Quality

The results of optimizing for quality are shown in Figures 38-39. The part was printed twice and ranked 6th and 12th out of 146 prints in terms of print quality according to the Image Classifier. As mentioned previously, the Image Classifier was dependent on human interpretation of errors. Outputs of these rankings should only be used to get a rough estimate of where the prints fall. These prints took 24 minutes 13 seconds to complete. The quality optimized print time as compared to each baseline print speed was:

- 20 mm/s: 45.45 min
- 40 mm/s: 23.52 min
- 60 mm/s: 16.05 min
- 80 mm/s: 12.53 min
- Quality Optimizations: 24.22 min

In addition, average percentage of "none" error for each print (Figure 40) was compared with the quality optimized print:

- 20 mm/s: 74.21%

- 40 mm/s: 76.04%
- 60 mm/s: 72.20%
- 80 mm/s: 69.59%
- Quality optimization: 86.41%

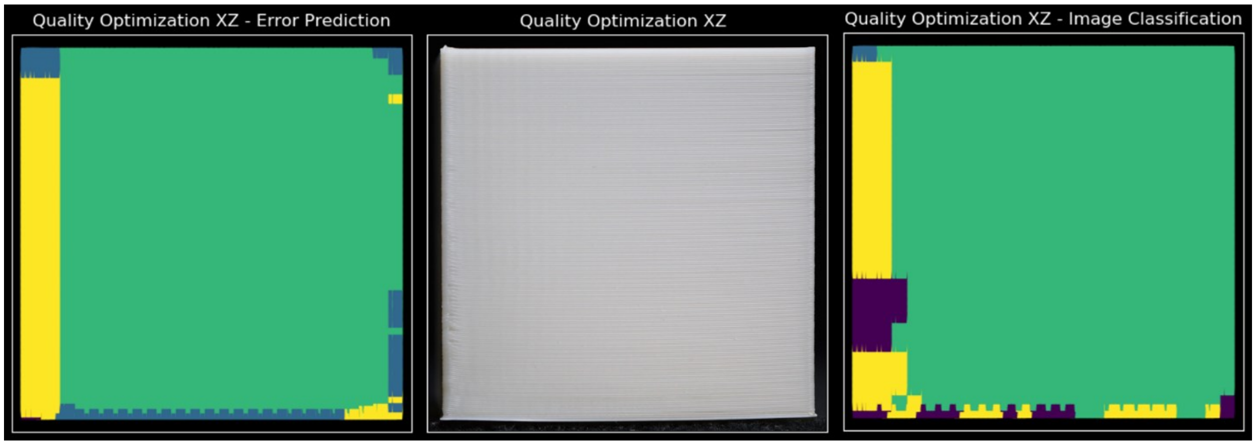


Figure 38. Error prediction model (left), printed part (center), and image classifier errors on XZ face for quality optimized part.

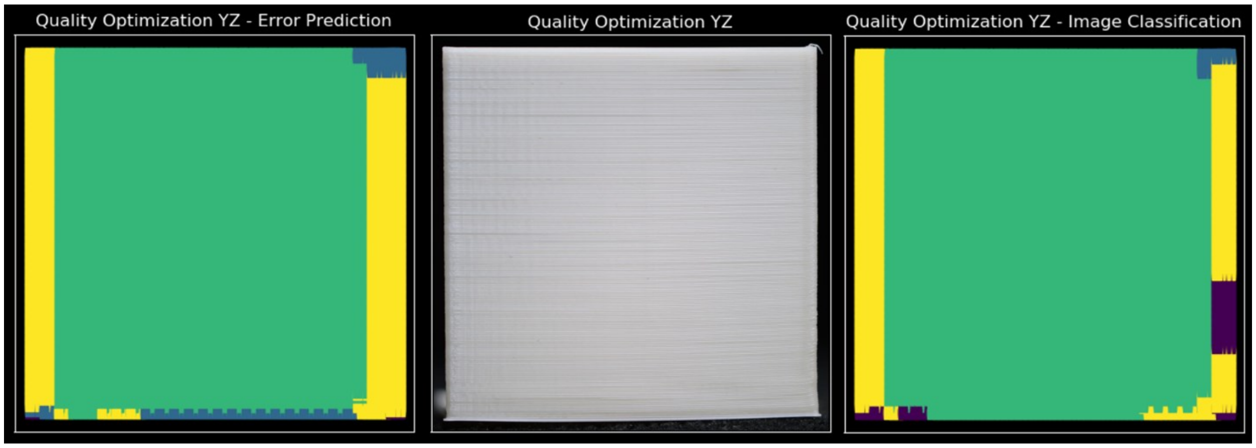


Figure 39. Error prediction model (left), printed part (center), and image classifier errors on YZ face for quality optimized part.

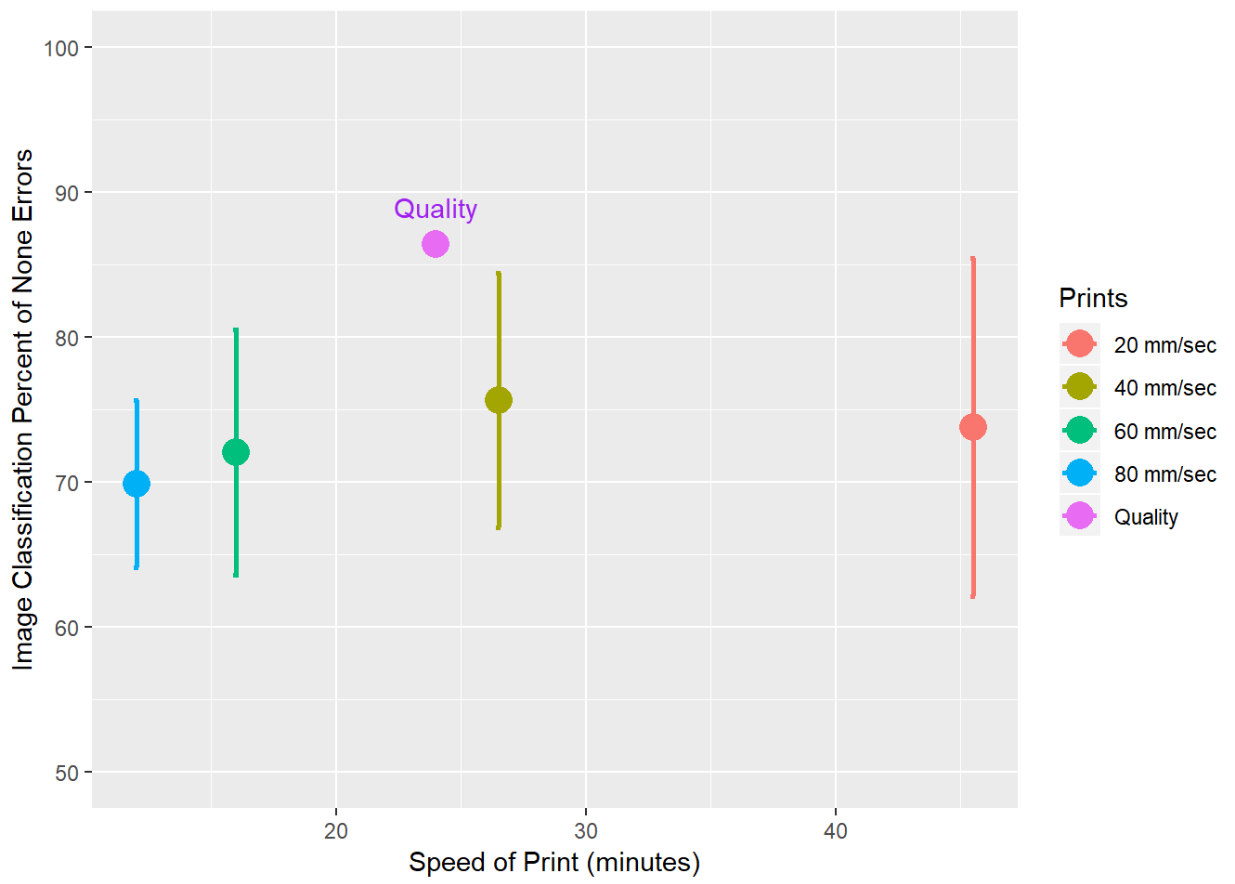


Figure 40. Print time vs. percent of “none” errors for quality optimized part and parts printed with constant speeds.

Speed

The goal for speed optimization was to keep print speed as high as possible while simultaneously limiting errors. Results are shown in Figures 41-42. The print with local speed optimizations took 16 minutes 56 seconds to complete. The speed optimized print time as compared to each baseline print speed was:

- 20 mm/s: 45.45 min
- 40 mm/s: 23.52 min
- 60 mm/s: 16.05 min
- 80 mm/s: 12.53 min
- Speed optimization: 16.93 min

Comparing the average percentages of “none” error at each print speed with the speed optimized print was (Figure 43):

- 20 mm/s: 74.21%
- 40 mm/s: 76.04%
- 60 mm/s: 72.20%
- 80 mm/s: 69.59%

- Speed optimization: 76.61%

The constant print speed percentages were in line with what was expected – there was some trade-off of speed to ensure quality. The speed optimized print outranked the average quality of each constant speed print.

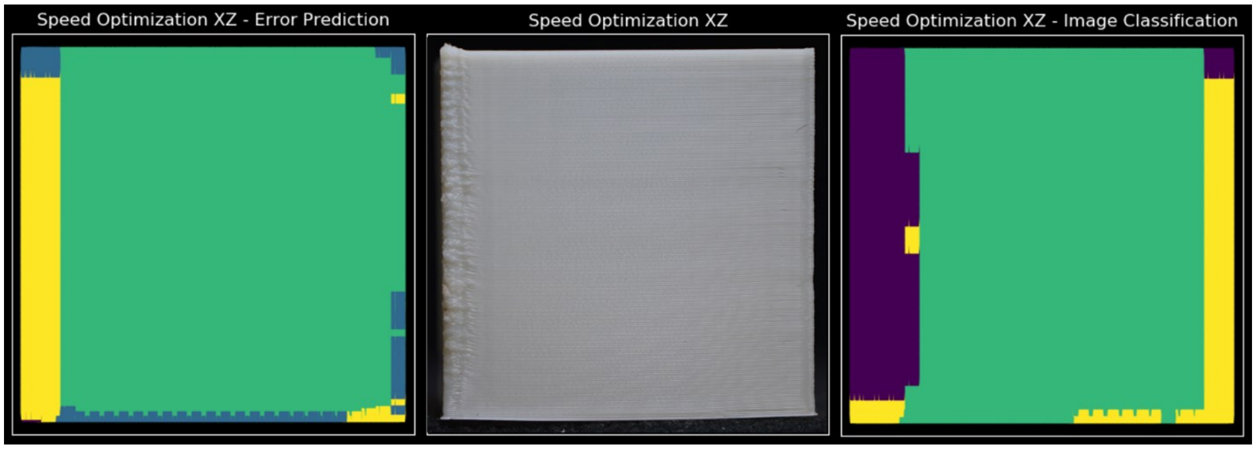


Figure 41. Error prediction model (left), printed part (center), and image classifier errors on XZ face for speed optimized part.

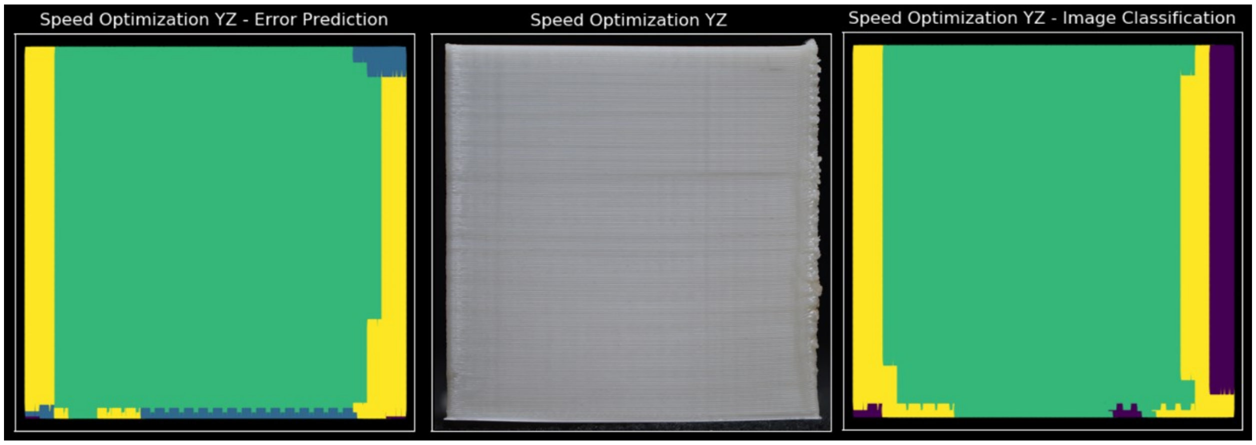


Figure 42. Error prediction model (left), printed part (center), and image classifier errors on YZ face for speed optimized part.

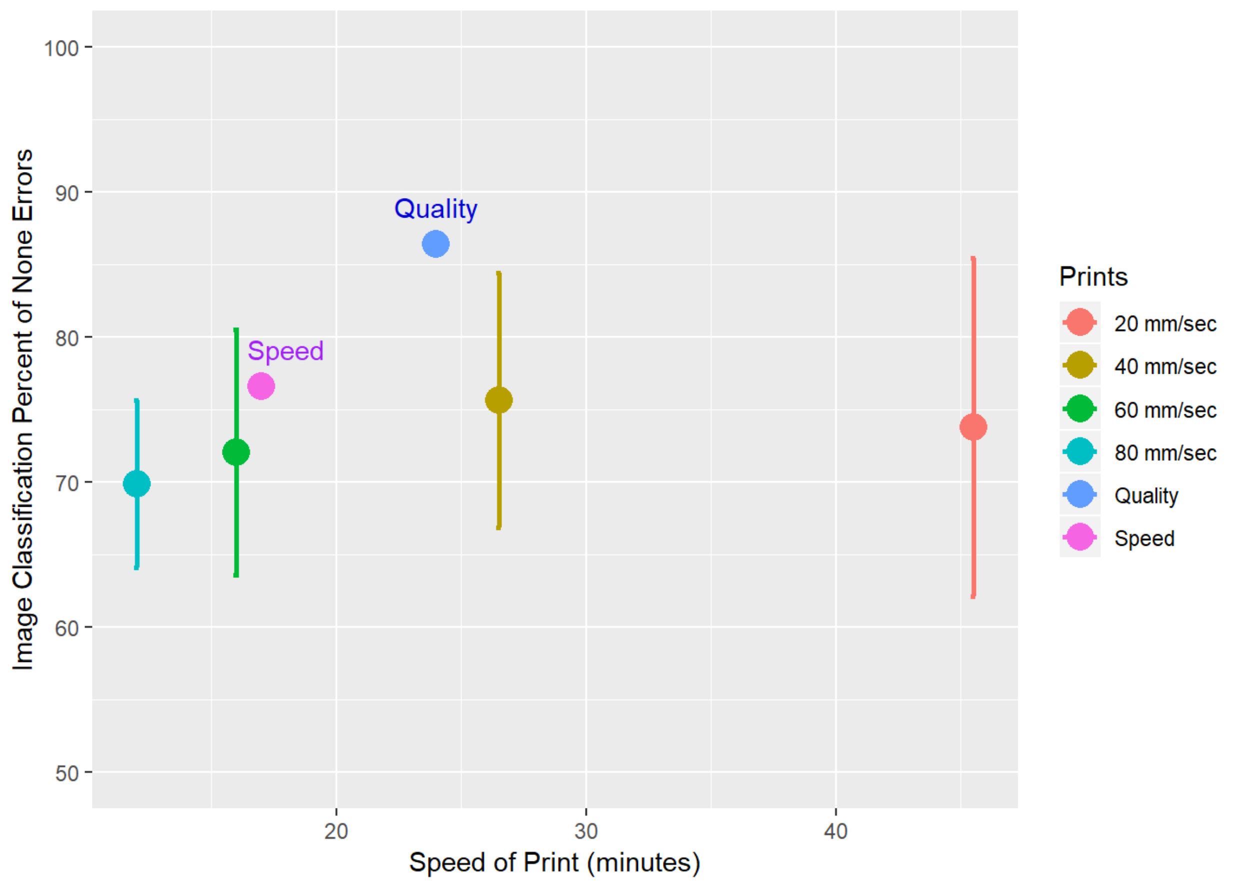


Figure 43. Print time vs. percent of “none” errors for speed optimized part, quality optimized part, and parts printed with constant speeds.

Errors

Forcing each error type was used to demonstrate the predictive power of the model. The parameters selected by the model to create blobs in a part are shown in Figure 44. While this print looked worse, it did not have the error-filled corners expected from a model trained on so many corners with large blobs. When the model attempted to force errors, it ramped up print speed and extrusion multiplier while decreasing fan speed. These were the conditions where blobs are normally present. However, blobs appeared more consistently when these parameters were kept constant throughout the print, unlike this example where the parameters were changed only near the intended location of the error. Global factors (e.g., overall part temperature, time between layers) and physical limitations of the printer and material that would affect the development of localized errors were not considered. In this print, going quickly from low print speeds to high print speeds around the corners had a negative impact on forcing blobs since the quick print speed increase resulted in less material extruded from the nozzle (a product of the physical limitations of the system).

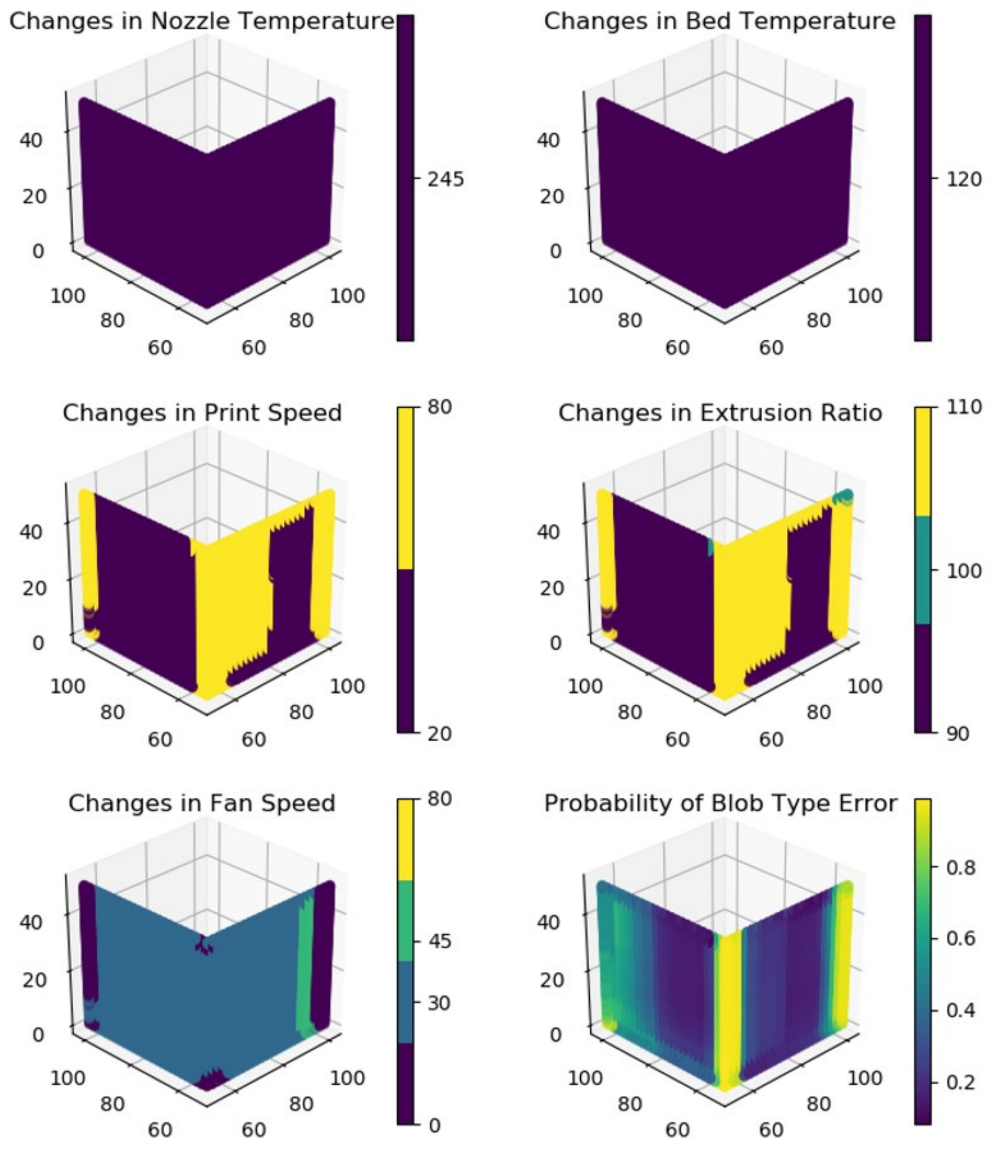


Figure 44. Print parameters and predicted errors for parts optimized for blobs.

Conclusions

The objective of the work reported here was to lay the groundwork for an end-to-end system that increases the quality and repeatability of 3D printing using machine learning. The approach taken was to develop code and models that take a CAD design and optimize the local printing parameters throughout a print. The machine learning models used here were trained for the "3 sides of a cube" geometry only, so that work could focus on developing and evaluating the overall approach. Future work will expand on tool capabilities for handling a wide range of part geometries. The intent of this paper is to explain the concepts and code used to create the data, train models, and predict the optimum printing parameters. The tool developed in this work was able to predict future print quality with ~80% accuracy and select local optimum printing parameters throughout the part to optimize for quality, obtaining “none” error rates in the top 10% of the data. Additionally, the tool optimized for print speed resulting in a 30% reduction in print time, while maintaining a high “none” error rate. The result of this project is a step towards using machine learning to improve repeatability and enable certification of printed parts in aerospace applications.

References

[1] Thomas-Seale, L. E. J., Kirkman-Brown, J. C., Attallah, M. M., Espino, D. M., and Shepherd, D. E. T., “The Barriers to the Progression of Additive Manufacture: Perspectives from UK Industry,” *International Journal of Production Economics*, Vol. 198, Apr. 2018, pp. 104–118.
https://doi.org/10.1016/j.ijpe.2018.02.003

[2] Abdollahi, S., Davis, A., Miller, J. H., and Feinberg, A. W., “Expert-Guided Optimization for 3D Printing of Soft and Liquid Materials,” *PLOS ONE*, Vol. 13, No. 4, Apr. 2018.
https://doi.org/10.1371/journal.pone.0194890

[3] Gardner, J. M., Hunt, K. A., Ebel, A. B., Rose, E. S., Zylich, S. C., Jensen B. D., Wise, K. E., Siochi, E. J., and Sauti, G., “Machines as Craftsmen: Localized Parameter Setting Optimization for Fused Filament Fabrication 3D Printing,” *Advanced Materials Technologies*, Vol. 4, No. 3, Jan. 2019.
https://doi.org/10.1002/admt.201800653

[4] Rao, P. K., Liu, J., Roberson, D., Kong, Z., and Williams, C., “Online Real-Time Quality Monitoring in Additive Manufacturing Processes Using Heterogeneous Sensors,” *Journal of Manufacturing Science and Engineering*, Vol. 137, No. 6, Sep. 2015.
https://doi.org/10.1115/1.4029823

[5] Aranda, S., “Complete ABS Profile Print Settings,” Dec. 2015.
https://www.sd3d.com/abs-settings/

[6] Panozzo, D., Jacobson, A., and others, “A Simple C++ Geometry Processing Library,” 2018.
https://libigl.github.io/

[7] Krizhevsky, A., Sutskever, I., and Hinton, G., "ImageNet Classification with Deep Convolutional Neural Networks," *Advances in Neural Information Processing Systems*, Vol. 25, 2012.
https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf

[8] "Transfer Learning Using AlexNet," *MATLAB & Simulink,* retrieved 18 Nov. 2022.
https://www.mathworks.com/help/deeplearning/ug/transfer-learning-using-alexnet.html

[9] Borchani, H., Varando, G., Bielza, C., Larrañaga, P., "A Survey on Multi-Output Regression," *Data Mining and Knowledge Discovery*, Vol. 5, No. 5, Sep. 2015.
https://doi.org/10.1002/widm.1157