Technical Solution for Accurate Printability Assessment of 3D Models

PodoPrinter — Sole Filter

Fontys UAS, Eindhoven, The Netherlands
Veronika Valeva - 4090349

Table of Contents

Introduction

SOLE by PodoPrinter is an innovator in the podiatric market. Through PodoPrinter's workflow, they offer the podiatrist the possibility to print insoles. PodoPrinter does this on a printer (that they have patented) that prints on a conveyor belt; this makes it possible to print 24/7. To make it run in the best possible way, the podiatrists design the insole with the PodoCAD program.

In order to connect other CAD programs to the PodoPrinter workflow, PodoPrinter wants to look at an extra input. This input must meet several conditions, security, and feedback. This would allow PodoPrinter to help even more people mobile with the right insoles.

For the software solution we are developing for SOLE-Filter by PodoPrinter. One of the main focuses of this project, and also the topic of this research document, is the validation of 3d files. Specifically, my research aims to develop a technical solution that can accurately assess the printability of 3D models across multiple file formats.

When importing a file, the user has to be informed on the printability of the 3d model he/she uploaded. If the 3d model does not fall under the verification – it is not advisable that they continue with the printing process. The imported 3D files need to be checked regarding the specific design requirements. These are certain requirements given to use by the client - that have to be considered when building the solution.

This research and implementation could not be possible without some client input. Apart from the file validation requirements, the client has provided couple of 3d files that will help test what has been done. The file formats provided are **STL** and **3dm**. Thus, this research will be focused around those two file formats.

At the end of the research, I intend to deliver a technical solution that can accurately assess the printability of 3D models. The solution will cover the key criteria for evaluating the printability of a 3D model and the best practices for analyzing the geometry, topology, and other features of a 3D model to determine its printability. The proposed solution will be implemented and used for the group project.

Research questions

The main question revolves around the issue we are facing. There the problem can have different interpretations and hence there may be multiple solutions. The solution will be based on the requirements of the project.

Main research question:

How can a technical solution be created to accurately assess the printability of 3D models across multiple file formats?

The sub questions are:

What are the technical requirements for creating a system that can accurately assess the printability of 3D models?

What are the key criteria for evaluating the printability of a 3D model, and how can they be quantified?

What are the best practices for analyzing the geometry, topology, and other features of a 3D model to determine its printability?

Research methods

To answer the research questions, I will be using the following research methods:

Library research strategy

- **Available product analysis**:
  To identify existing solutions and products related to 3D model printability assessment.
- **Community research**:
  To review relevant literature and publications related to 3D model assessment and 3D printing. I'll be looking at what people say about the different LIBRARIES to have a critical view of what will be best for this project.
- **Competitive analysis**:
  As the different libraries have different things to offer it is good to look at whichever has what we need for the project and analyze what is the best option to go with.

Field research strategy

- **Document analysis**:
  To analyze the existing documentation provided by the stakeholder on the existing solutions for printability assessment.
- **Explore user requirements**:
  Looking into what is required for this project is essential to make the best choice, for this I would have to take a deeper look into the structure of 3d files and consider the requirements from the client.

Lab research strategy

- **System test**:
  Testing the different components as a greater whole and seeing all the individual parts work together in different scenarios.

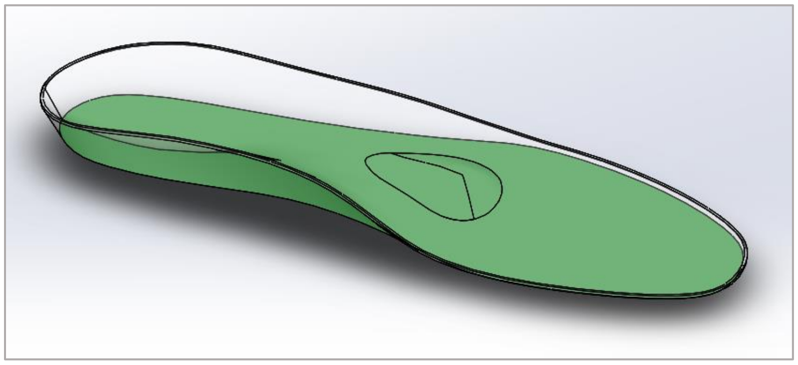
Showroom research

- **Peer review**:
  Having a conversation with the technical teacher and discussing the choices that have been made, along with their reasons, is important. Using their feedback can make the solution better. To get feedback from the other group members – also involved with the group project.

Requirements - Verify the 3D data.

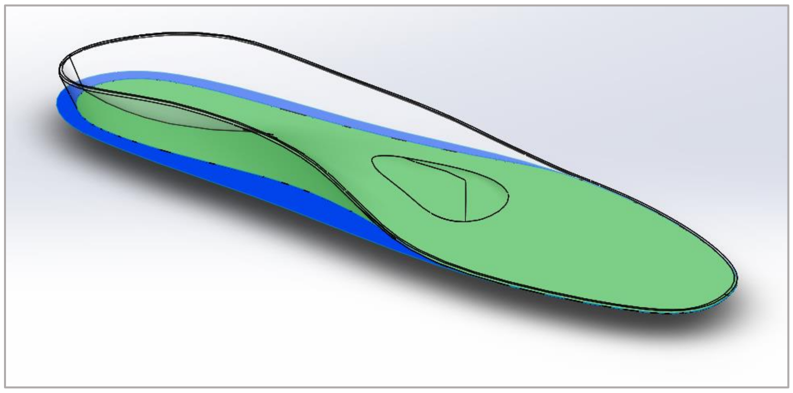
The bottom surface of the insole needs to be flat.

The bottom surface of the insole comes in direct contact with the conveyor belt during the 3D printing process. It needs to be perfectly flat to ensure a reliable printing outcome. Any irregularities in flatness greater than ±0.1mm are unacceptable and would lead to the rejection of the design. The image provided below depicts a transparent insole with the highlighted flat underside surface in green.



The flat underside surface needs to be large enough for a reliable 3D printing process.

If the bottom surface area of the digital 3D insole is less than 70% of its footprint, the design should be rejected. To illustrate this, the image below displays a transparent insole where the green color represents the bottom surface and the combined green and blue colors represent the footprint of the insole.

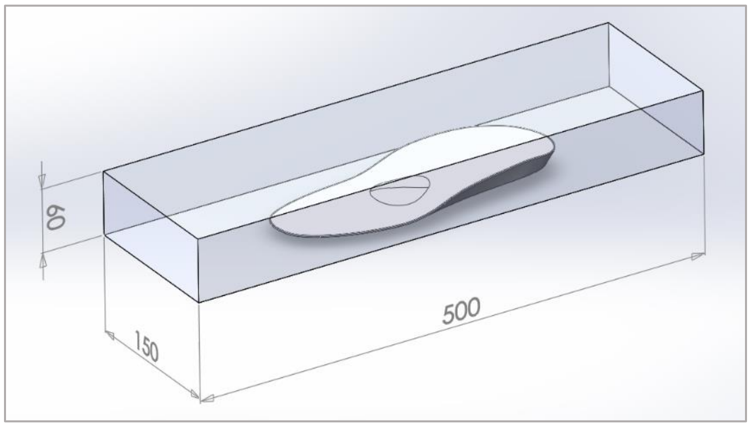


The minimum thickness is 1mm for any part of the insole.

In the event that any area of the insole is thinner than 1mm, there is a risk of defects such as holes appearing in the 3D printed insole. To address this, the user will receive a notification and will have the opportunity to review the insole in the preview window. They can then personally decide whether to accept or reject the 3D design based on their examination.

The Dimensions must be within a certain range.

For the digital 3D insole, it is crucial that its dimensions remain within 60mm x 150mm x 500mm. If the insole exceeds these measurements and does not fit within the specified 60mm x 150mm x 500mm box, the design will be rejected. To provide a visual representation of the maximum dimensions, the image below depicts a white insole contained within a transparent box measuring 60mm x 150mm x 500mm.

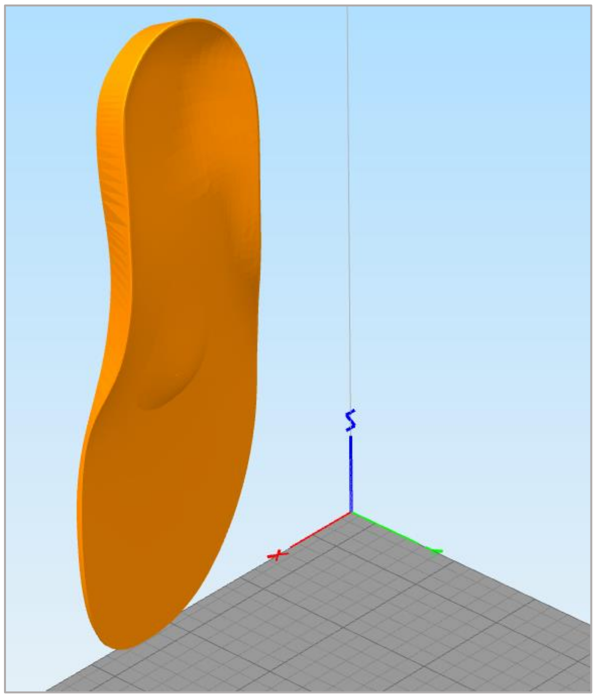


The 3D insole design needs to be watertight, so no errors occur within the slicing process.

When a 3D object exhibits defects in its outer walls, it can lead to issues during the exporting process in the slicing software. In such cases, the exported data may be corrupted or, in some instances, no data might be exported at all. Therefore, it is essential to address any open surfaces in the 3D insole design. If the design includes open surfaces, an automatic process may be implemented to close them to ensure the integrity and completeness of the model.

The digital 3D insole needs to be oriented in the correct position.

To ensure proper alignment for slicing, it is necessary for the underside of the insole to be parallel to the XZ-plane. Additionally, the insole itself should be aligned with the Z direction, pointing downward toward the X axis. The image provided below illustrates an insole that is correctly aligned with the coordinate system for slicing, with the X-axis depicted in red, the Y-axis in green, and the Z-axis in blue.



Reading 3d files

The technical solution for accurate printability assessment of 3D models requires a robust and efficient programming language to ensure seamless validation of files. But before validating, the file has to be sent by another service and received by the validation service through an API. In this regard, Python emerges as the ideal choice for developing such a tool, particularly leveraging the capabilities of the FastAPI framework. Python offers a wide range of libraries and modules specifically designed for 3D modeling and file processing, making it highly suitable for this task. Its simplicity, readability, and extensive community support contribute to faster development and easier maintenance. Additionally, Python's versatility allows seamless integration with existing systems and APIs, ensuring smooth data transfer and effective validation of 3D files. With the utilization of FastAPI, a modern, high-performance web framework, the tool can efficiently handle API requests and provide real-time feedback on printability assessment. Thus, Python, coupled with FastAPI, provides an optimal solution for developing a robust and accurate printability assessment tool for 3D models.

Until recently, people primarily relied on Matplotlib to visualize 3D content in Python. Matplotlib is convenient as it comes bundled with Python installations, but it lacks GPU hardware acceleration. Consequently, performance issues arise when dealing with large datasets, causing sluggishness and unresponsiveness. To overcome this limitation, many turned to the Point Cloud Library (PCL) for 3D analysis. While PCL, built in C++, offers versatility and responsiveness, it falls short in terms of Python integration and Windows compatibility. However, several Python libraries like Trimesh, PyVista, etc. have emerged to address these shortcomings. Each library offers unique functionalities for analysis, generation, manipulation, and visualization of meshes and point clouds. Each of these libraries has diverse visualization capabilities, manual controls, lighting options, animation, and raytracing visuals. The accompanying code, all of the documentation needed, and an open issue-exchanging community can be found in GitHub repositories.

Reading 3dm file formats

In our research, we will adopt an approach to reading 3DM files for accurate printability assessment of 3D models. To achieve this, we have chosen to utilize the rhino3dm library in conjunction with compute-rhino3dm. These libraries offer comprehensive solutions that fulfill all the necessary requirements for our file reading and processing needs. Rhino3dm provides extensive support for reading, manipulating, and analyzing 3D models in the 3DM file format. It is a part of the Rhino3dm libraries. Additionally, compute-rhino3dm offers advanced computational capabilities, enabling us to perform complex operations and computations on the 3D models. According to the official Rhino Developer website this is a work in progress package which is meant to add classes which are not available through rhino3dm.py. One of the functions – which would have been definitely put into practice is the function to compute the thickness of a 3d model. Deeper into my research I found out that this package works by calling the API of a Rhino Compute Server – fully accessible by any user. However, right after this I also found out that this package is no longer a work in progress, no longer working with an open API. Now, compute-rhino3dm can be used with locally running instance of the Compute Rhino Server. In order to set up Rhino Compute Server locally you would need a to have downloaded Rhino 7 – which is pretty expensive for commercial use. **That does not mean that we are going to exclude this file format from this research document, however, the main focus will be on reading and validating 3d model with .STL extensions.**

Rhino3dm.py is a python package that can be used in all current versions of Python and is available on all platforms (Windows, macOS, Linux).

```
>> pip install --user rhino3dm
```

Reading STL file formats

In our research, we will focus on developing a comprehensive approach to reading STL files for accurate printability assessment of 3D models. In the wild, STL is perhaps the most common format. STL files are extremely simple: it is basically just a list of triangles. They are robust and are a good choice for basic geometry. To accomplish this, we have chosen to utilize the Trimesh library, as it offers robust solutions that meet all the necessary requirements. Trimesh provides extensive support for reading and manipulating 3D mesh data, making it an ideal choice for our file processing needs. Additionally, we have opted to employ PyVista for visualizing the 3D content. The content that is going to be visualized by PyVista is going to be done only for developing purposes. This is not the visualization that the user will be provided with on their end of the web-application. PyVista offers a powerful and user-friendly interface for interactive 3D visualization, allowing us to effectively analyze and assess the printability of the models. Thus, helping the developer in the implementation process and to showcase the intermediate results.

Trimesh is a Python-based library specifically designed for loading, analyzing, and visualizing meshes and point clouds. It is widely utilized in various applications, including pre-processing 3D assets before statistical analysis and machine learning, as well as in 3D printing software like Cura. One of the notable advantages of Trimesh is its ease of installation, as it only requires the numpy library as a prerequisite.

```
>> pip install numpy
>> pip install trimesh
```

Once installed, the library can be imported by calling `import trimesh`. Loading meshes is straightforward using the `trimesh.load()` method, with optional inputs for specifying the file type. Upon loading a mesh, a lot of information can be directly extracted. For instance, determining if the mesh is watertight can be achieved using `mesh.is_watertight`, calculating its convex hull can be done via `mesh.convex_hull`, and obtaining the volume can be accomplished using `mesh.volume`. In cases where the mesh consists of multiple objects, it is possible to split them into individual meshes using the `mesh.split()` function.

How to cite the library?

```
@software{trimesh,
    author = {{Dawson-Haggerty et al.}},
    title = {trimesh},
    url = {https://trimsh.org/},
    version = {3.2.0},
    date = {2019-12-8},
}
```

If you want to deploy something in a container that uses trimesh automated `debian:slim-bullseye` based builds with trimesh and most dependencies are available on Docker Hub with image tags for `latest`, git short hash for the commit in main (i.e., `trimesh/trimesh:0c1298d`), and version (i.e., `trimesh/trimesh:3.5.27`): `docker pull trimesh/trimesh`.

PyVista is a powerful plotting and mesh analysis library that leverages the Visualization Toolkit (VTK). It simplifies the VTK interface, providing a more Pythonic and user-friendly approach. PyVista supports both point clouds and meshes, offering a wide range of functionalities. With an abundance of examples and tutorials available, it serves to developers at all levels, including students who can easily integrate it into their learning process. PyVista is compatible with Linux, Mac, and Windows operating systems and requires Python 3.8 or later. As it is built on top of VTK, it necessitates the installation of VTK and numpy as minimum prerequisites. The installation process is straightforward and can be accomplished using pip.

```
>> pip install pyvista
```

To verify the successful installation, one can import the library by calling `import pyvista as pv` and then test it by invoking `pv.demos.plot_wave()`. If a plotting window displaying a sine wave appears, it confirms the successful installation of the library. How to cite the library?

```
@article{sullivan2019pyvista,
  doi = {10.21105/joss.01450},
  url = {https://doi.org/10.21105/joss.01450},
  year = {2019},
  month = {May},
  publisher = {The Open Journal},
  volume = {4},
  number = {37},
  pages = {1450},
  author = {Bane Sullivan and Alexander Kaszynski},
  title = {{PyVista}: {3D} plotting and mesh analysis through a streamlined interface
for the {Visualization Toolkit} ({VTK})},
  journal = {Journal of Open Source Software}
}
```

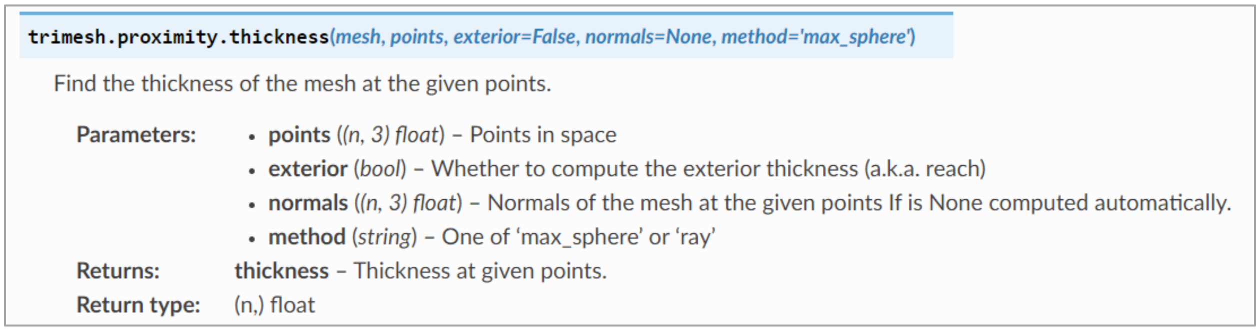
Computing the thickness of a 3d model

The importance of accurately assessing the thickness of 3D printed insoles is crucial to ensure their structural integrity and functionality. When any area of the insole is thinner than 1mm, it poses a significant risk of defects, such as holes or weak spots, emerging during the printing process. These defects can compromise the overall quality and performance of the insole, leading to discomfort or potential failure when used. To mitigate this risk, our system incorporates an evaluation mechanism. Using computing functions provided by the library in use, we precisely analyze the thickness of the insole across its entire surface. By comparing the thickness values against the predetermined threshold of 1mm, we can identify areas that may be susceptible to defects and visualize them.

STL

```
def measure_stl_thickness(mesh: trimesh):
    mesh = trimesh.load('file path', process=False)
    start_points = mesh.vertices - (0.09 * mesh.vertex_normals)
    thickness = trimesh.proximity.thickness(mesh, start_points,
normals=mesh.vertex_normals)
    pv.wrap(mesh).plot(scalars=thickness)
```

(trimesh.proximity, 2023)


trimesh.proximity.thickness(*mesh, points, exterior=False, normals=None, method='max_sphere'*)

Find the thickness of the mesh at the given points.

Parameters:	• **points** (*(n, 3) float*) – Points in space • **exterior** (*bool*) – Whether to compute the exterior thickness (a.k.a. reach) • **normals** (*(n, 3) float*) – Normals of the mesh at the given points If is None computed automatically. • **method** (*string*) – One of 'max_sphere' or 'ray'
Returns:	**thickness** – Thickness at given points.
Return type:	(n,) float

The provided code defines a function named **measure_stl_thickness** that takes a **trimesh** object as input. The function starts by loading an STL file from the specified file path using **trimesh.load**. The **process=False** argument ensures that the loaded mesh is not processed or modified during loading, preserving its original state.

Next, the code calculates the **start_points** by subtracting a fraction of the mesh's vertex normals from its vertices. trimesh function as it is computes the thickness of the whole model including the top surface layer of the mesh. Due to all of the angles of the mesh this would lead to wrong results To mitigate this the operation offsets the starting points for measuring thickness inside the mesh.

The **trimesh.proximity.thickness()** function from the **trimesh** library implements the Shrinking Sphere algorithm. It takes the following parameters:

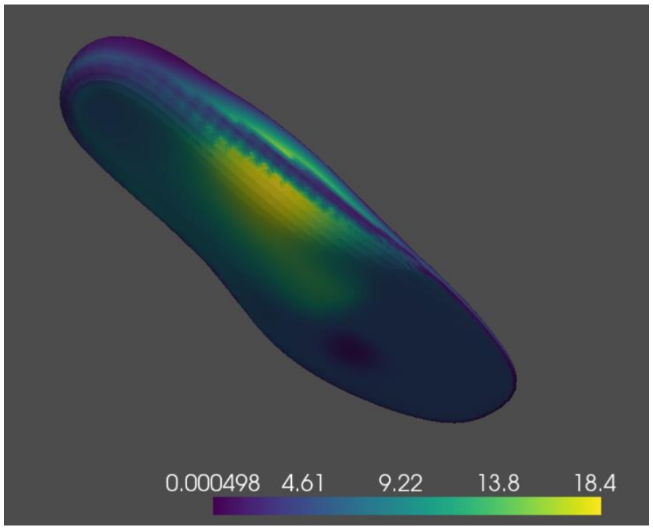
- **mesh**: The input 3D object represented as a mesh.
- **start_points**: The set of start points on the object's surface, which determines the initial sphere positions.
- **normals**: Optional parameter providing the vertex normals of the mesh. These normals assist in offsetting the start points along the surface.

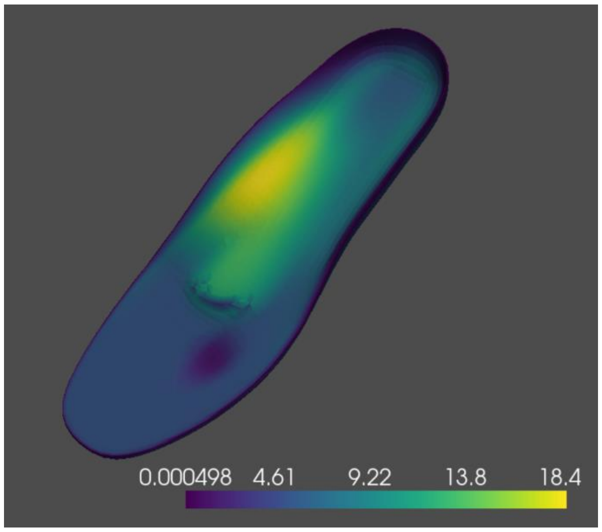
The function utilizes the Shrinking Sphere algorithm internally to compute the thickness values. It performs the necessary iterations and calculations to determine the distances between the sphere's touch points. The result is a thickness value assigned to each vertex of the mesh, representing the estimated thickness at that location. The resulting thickness values are stored in the **thickness** variable.

The Shrinking Sphere algorithm is a parallel algorithm used for computing the thickness of 3D objects, such as meshes or point clouds. It operates by iteratively shrinking a sphere placed around the object until it touches the surface at two points. The distance between these two points represents the estimated thickness. To explain how the algorithm works, let's break down the steps involved:

- Initialization: The algorithm requires a 3D object, such as a mesh, as input. Additionally, it needs to define a set of start points on the object's surface. These start points are typically obtained by offsetting the surface along the vertex normals to ensure they lie slightly inside the object.
- Shrinking Sphere Iteration: The algorithm starts with an initial sphere encompassing the object. In each iteration, the sphere's radius is reduced, and its center moves towards the object's surface. The sphere is adjusted until it touches the surface at two distinct points.
- Distance Calculation: Once the sphere touches the surface at two points, the distance between these points is calculated. This distance represents an estimate of the object's thickness at that location.
- Repeat: The shrinking sphere iteration continues until the desired accuracy or convergence criteria are met. The process repeats for multiple start points on the object's surface to obtain thickness values at various locations.

Finally, the result is visualized using PyVista by wrapping the **mesh** object with **pv.wrap()** and calling the **plot()** method. The **scalars** parameter is set to the **thickness** array, which assigns the calculated thickness values as the scalar data for coloring the mesh during visualization.



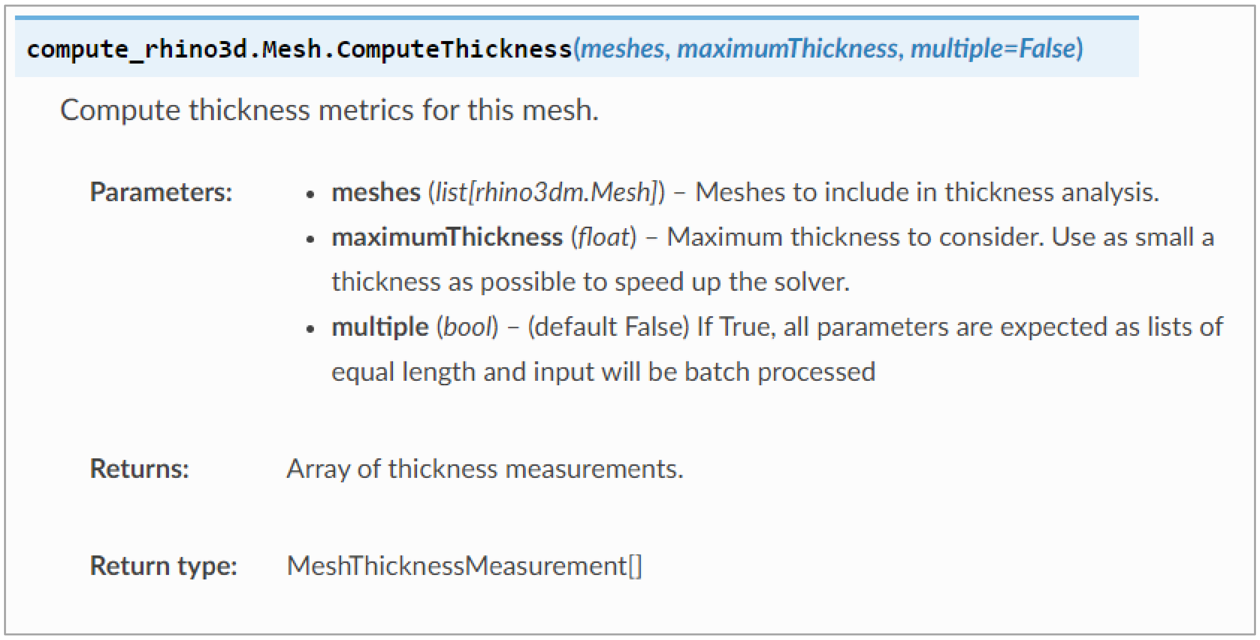


This is an example visualization of a mesh model. The thickness of the insole is showed using colours. The thinner the region – the darker the colour, and vice versa – the thicker the region – the brighter the colour. In this example the insole’s maximum thickness is 18.4cm. And the minimum is 0.000498cm. Of course, even after making sure the starting points of the insole are offset slightly inside the model – we can still get the surface edges in the computation. For this reason we decided to visualize the model and the thickness regions to let the user to determine for themselves if the insole is valid or not.

3dm

```python
def measure_3dm_thickness():
    model = rhino3dm.File3dm.Read('file path')
    mesh = rhino3dm.Mesh()
    for obj in model.Objects:
      if isinstance(obj.Geometry, rhino3dm.Mesh):
            mesh.Append(obj.Geometry)
    maximumThickness = None
    thickness = compute_rhino3d.Mesh.ComputeThickness(mesh, maximumThickness,
multiple=False)
```

(Mesh - compute_rhino3d 0.14.0 documentation, 2023)


`compute_rhino3d.Mesh.ComputeThickness`(*meshes*, *maximumThickness*, *multiple=False*)

Compute thickness metrics for this mesh.

Parameters:
- **meshes** (*list[rhino3dm.Mesh]*) – Meshes to include in thickness analysis.
- **maximumThickness** (*float*) – Maximum thickness to consider. Use as small a thickness as possible to speed up the solver.
- **multiple** (*bool*) – (default False) If True, all parameters are expected as lists of equal length and input will be batch processed

Returns: Array of thickness measurements.

Return type: MeshThicknessMeasurement[]

The code snippet above demonstrates how to measure the thickness of a 3DM (Rhino 3D) model using the rhino3dm and compute-rhino3dm libraries. The function **measure_3dm_thickness()** reads a 3DM file by calling the **Read()** method from the rhino3dm library, passing the file path as an argument. The loaded 3DM model is stored in the **model** variable. A new empty **rhino3dm.Mesh()** object called **mesh** is created to hold the mesh data extracted from the 3DM model. The code iterates through each object in the **model.Objects** collection using a **for** loop. Inside the loop, it checks if the object's geometry is an instance of **rhino3dm.Mesh** by using the **isinstance()** function. If the geometry is a mesh, it appends the mesh to the **mesh** object using the **Append()** method. The **compute_rhino3d.Mesh.ComputeThickness()** function is called, passing the **mesh**, **maximumThickness**, and **multiple=False** arguments. This function calculates the thickness of the mesh. The calculated thickness value is returned as the result of the function.

Position of the 3d model in space

Proper alignment of the insole for slicing is crucial in the 3D printing process to ensure accurate and successful fabrication. The text emphasizes two essential alignment requirements: the underside of the insole being parallel to the XZ-plane and the insole itself aligned with the Z direction, pointing downward toward the X axis.

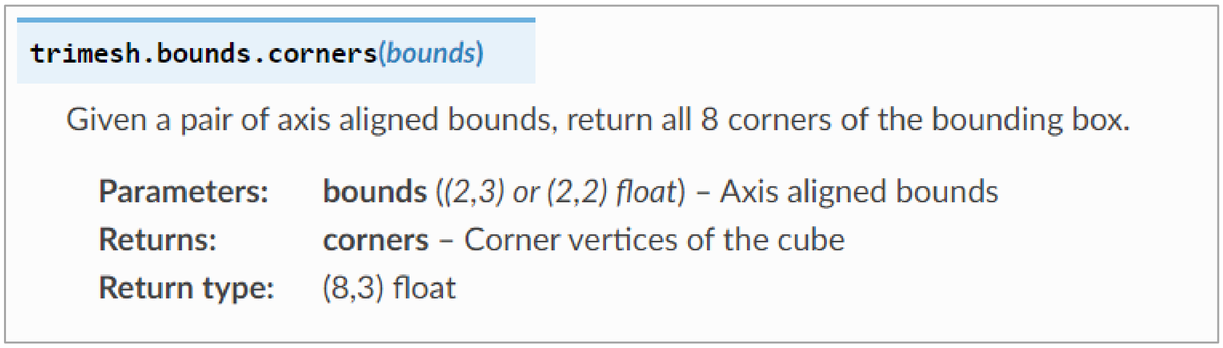
The XZ-plane is a horizontal plane that represents the build plate or the printing surface of the 3D printer. Aligning the underside of the insole parallel to this plane ensures that it rests perfectly on the build plate during printing. This alignment is essential to maintain stability and prevent any tilting or warping of the insole during the printing process. If the underside is not parallel to the XZ-plane, it can result in uneven layering or poor adhesion to the build plate, leading to defects or a failed print.

Aligning the insole with the Z direction, pointing downward toward the X axis, is necessary to establish the correct orientation for printing. The Z direction represents the vertical axis, indicating the layer-by-layer build direction of the print. By aligning the insole with the Z direction, the printer can accurately follow the design and build each layer in the intended direction. This alignment ensures that the structural integrity and shape of the insole are preserved throughout the printing process.

STL

```
def measure_stl_position(mesh: trimesh) -> bool:
    listOfBoundCorners =
trimesh.bounds.corners(mesh.bounding_box_oriented.bounds)
    if listOfBoundCorners.__len__()>8:
        return False
    x_corners = []
    y_corners = []
    z_corners = []

    for bound in listOfBoundCorners:
        if not x_corners.__contains__(bound[0]):
            x_corners.append(bound[0])
        if not y_corners.__contains__(bound[1]):
            y_corners.append(bound[1])
        if not z_corners.__contains__(bound[2]):
            z_corners.append(bound[2])

    x_bound = round(np.max(x_corners) - np.min(x_corners))
    y_bound = round(np.max(y_corners) - np.min(y_corners))
    z_bound = round(np.max(z_corners) - np.min(z_corners))
    if not y_bound < x_bound < z_bound: return False
    return True
```

(trimesh.bounds, 2023)



`trimesh.bounds.corners(bounds)`

Given a pair of axis aligned bounds, return all 8 corners of the bounding box.

Parameters:	**bounds** (*(2,3) or (2,2) float*) – Axis aligned bounds
Returns:	**corners** – Corner vertices of the cube
Return type:	(8,3) float

The code snippet above demonstrates a function **measure_stl_position()** that measures the position of a 3D mesh object using the trimesh library.

The function **measure_stl_position()** takes a **mesh** object of type **trimesh** as input and returns a Boolean value. The code first computes the corners of the bounding box of the mesh using the **trimesh.bounds.corners()** function. These corners represent the extreme points of the mesh's bounding box in 3D space. If the number of corners is greater than 8, which indicates that the mesh is not well-formed or has invalid geometry, the function returns **False**. The code initializes three empty lists, **x_corners**, **y_corners**, and **z_corners**, to store the unique coordinates of the bounding box corners along the X, Y, and Z axes, respectively. The **for** loop iterates over each corner in the **listOfBoundCorners** and checks if the corner's X, Y, or Z coordinate is not already present in the respective lists. If it is not present, the coordinate is added to the corresponding list. The code then calculates the bound (range) of each axis by subtracting the minimum value from the maximum value in the corresponding list of corners. The **round()** function is used to round the bound values to the nearest integer. The next condition checks if the Y bound is less than the X bound, and the X bound is less than the Z bound. If this condition is not met, it indicates that the mesh is not properly positioned, and the function returns **False**. If all the conditions are met, indicating that the mesh is well-positioned, the function returns **True**.

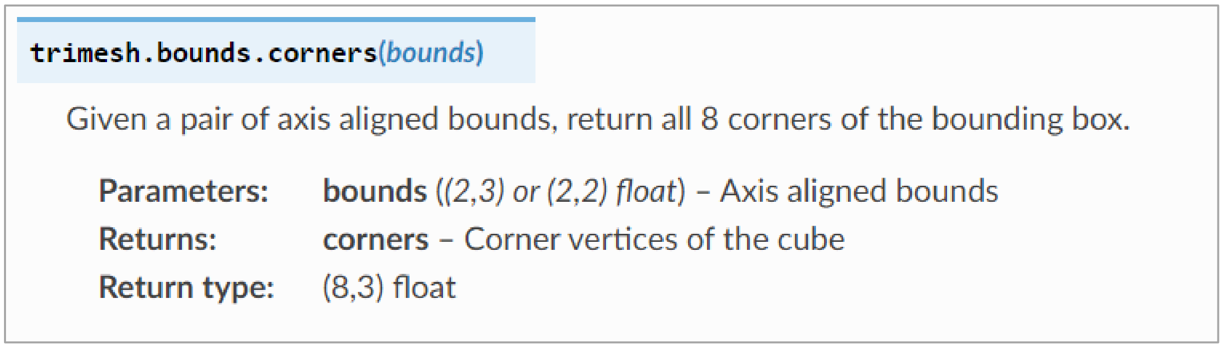
Dimensions of the 3d model

Ensuring that the dimensions of the digital 3D insole remain within the specified limits of 60mm x 150mm x 500mm is of utmost importance. The specified dimensions of 60mm x 150mm x 500mm represent the maximum allowable size for the insole. These dimensions are typically based on the capabilities of the 3D printing or manufacturing equipment. Exceeding these limits may result in practical difficulties during the fabrication process. Limiting the dimensions of the insole aids in material optimization. By defining a maximum size, the design can be optimized to use the appropriate amount of material efficiently. This helps minimize material waste, reduce production costs, and improve the overall sustainability of the manufacturing process.

STL

```
def measure_stl_dimension(mesh: trimesh) -> bool:

    const_bounds = [60, 150, 500]
    listOfBoundCorners =
trimesh.bounds.corners(mesh.bounding_box_oriented.bounds)
    if listOfBoundCorners.__len__()>8:
        return False
    x_corners = []
    y_corners = []
    z_corners = []

    for bound in listOfBoundCorners:
        if not x_corners.__contains__(bound[0]):
            x_corners.append(bound[0])
        if not y_corners.__contains__(bound[1]):
            y_corners.append(bound[1])
        if not z_corners.__contains__(bound[2]):
            z_corners.append(bound[2])

    arranged_bounds.sort()
    arranged_bounds = [x_bound, y_bound, z_bound]

    for i in range(2):
        if(arranged_bounds[i]>const_bounds[i]):
            return False
    return True
```

(trimesh.bounds, 2023)



`trimesh.bounds.corners(bounds)`

Given a pair of axis aligned bounds, return all 8 corners of the bounding box.

Parameters:	**bounds** (*(2,3) or (2,2) float*) – Axis aligned bounds
Returns:	**corners** – Corner vertices of the cube
Return type:	(8,3) float

The provided code aims to measure the dimensions of an STL mesh using the trimesh library and determine whether it satisfies certain predefined size constraints. The code starts by defining a list of constant bounds [60, 150, 500]. These values represent the maximum allowable dimensions in the X, Y, and Z directions, respectively. The code then obtains the corner points of the bounding box of the mesh using **trimesh.bounds.corners(mesh.bounding_box_oriented.bounds)**. This provides a list of eight corners that define the extent of the mesh in 3D space. The code initializes three empty lists: **x_corners**, **y_corners**, and **z_corners**. It iterates over each corner in the list of bound corners and checks if the X, Y, and Z coordinates of the corner have already been added to their respective lists. If not, the coordinate is appended to the appropriate list. This process ensures that each coordinate axis has a list of unique values representing the extreme points of the mesh in that direction. The code sorts the **x_corners**, **y_corners**, and **z_corners** lists in ascending order using **arranged_bounds.sort()**. Then, the sorted values are assigned to the **x_bound**, **y_bound**, and **z_bound** variables respectively. The code enters a loop that iterates twice, corresponding to the X and Y dimensions. For each iteration, it compares the value of the arranged bounds against the corresponding constant bound. If any of the arranged bounds exceed the constant bounds, the function returns **False**, indicating that the mesh exceeds the allowable dimensions in the X or Y direction. If the dimensions of the mesh satisfy the size constraints in both the X and Y directions, the function returns **True**, indicating that the mesh is within the allowable dimensions.

To summarize, the code calculates the dimensions of an STL mesh by obtaining its bounding box corners, extracting the unique coordinate values for each axis, and comparing them against predefined maximum bounds. This allows for the validation of whether the mesh fits within the specified size limitations.

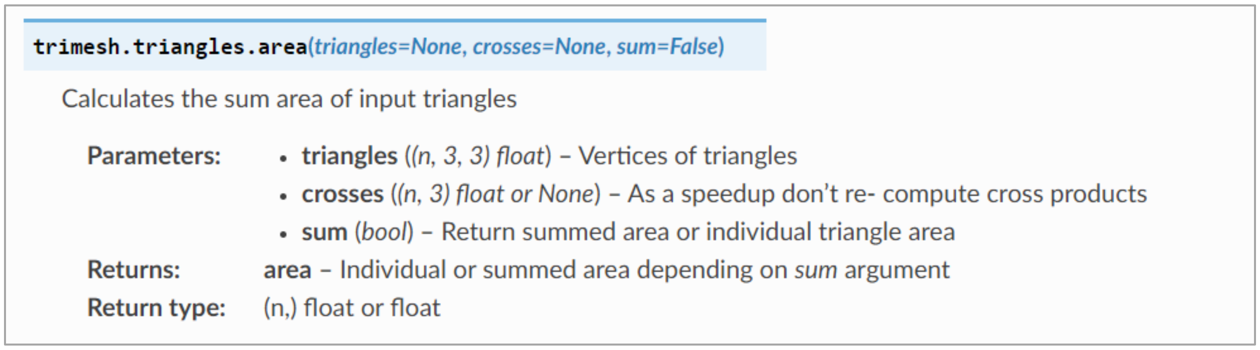
Surface size proportions

Ensuring that the bottom surface area of the digital 3D insole is at least 70% of its footprint is crucial for maintaining structural integrity and stability. This requirement helps prevent issues such as instability, weak support, and discomfort that may arise if the bottom surface area is insufficient. A sufficient bottom surface area provides increased stability to the printed insole.

STL

```
def check_if_bottom_surface_is_large_enough(mesh: trimesh) -> bool:
    up_axis = 2 # z vertical axis
    bottom_threshold = -0.99
    top_threshold = 0.25

    bottom_mask = mesh.vertex_normals[:, up_axis] < bottom_threshold
    top_mask = mesh.vertex_normals[:, up_axis] > top_threshold

    triangle_bottom_mask = numpy.any(bottom_mask[mesh.faces],axis=-1)
    triangle_top_mask =  numpy.any(top_mask[mesh.faces],axis=-1)

    bottom_area = trimesh.triangles.area(mesh.triangles[triangle_bottom_mask])
    top_area = trimesh.triangles.area(mesh.triangles[triangle_top_mask])

    print(mesh.vertices.shape)

    return f'The bottom of the insole is at {int(100 *
numpy.sum(bottom_area)/numpy.sum(top_area))}% of the top surface area of the
insole'
```

(trimesh.triangles, 2023)



trimesh.triangles.area(*triangles=None, crosses=None, sum=False*)

Calculates the sum area of input triangles

Parameters:
- **triangles** (*(n, 3, 3) float*) – Vertices of triangles
- **crosses** (*(n, 3) float or None*) – As a speedup don't re- compute cross products
- **sum** (*bool*) – Return summed area or individual triangle area

Returns: **area** – Individual or summed area depending on *sum* argument

Return type: (n,) float or float

The provided code checks whether the bottom surface area of a 3D mesh representing a digital insole is large enough compared to the top surface area. **up_axis = 2** defines the vertical axis as the Z-axis, indicating that the insole is expected to be aligned with the Z direction. **bottom_threshold = -0.99** and **top_threshold = 0.25** set the threshold values for the vertex normals along the Z-axis. These values determine the range within which the bottom and top surfaces are identified. **bottom_mask** creates a boolean mask that identifies vertices with normals below the **bottom_threshold**, indicating the bottom surface of the insole. **top_mask** creates a boolean mask that identifies vertices with normals above the **top_threshold**, representing the top surface of the insole. **triangle_bottom_mask** applies the **bottom_mask** to the faces of the mesh to identify triangles that have at least one vertex on the bottom surface. **triangle_top_mask** applies the **top_mask** to the faces of the mesh to identify triangles that have at least one vertex on the top surface. **bottom_area** calculates the total area of the triangles identified by **triangle_bottom_mask** using the **trimesh.triangles.area()** function. This represents the area of the bottom surface. **top_area** calculates the total area of the triangles identified by **triangle_top_mask** using the **trimesh.triangles.area()** function. This represents the area of the top surface. **print(mesh.vertices.shape)** displays the shape of the mesh's vertices, providing information about the mesh's geometry. The function returns a string indicating the percentage of the bottom surface area compared to the top surface area of the insole. Example output: "The bottom of the insole is at 70% of the top surface area of the insole"

The code evaluates the relative size of the bottom surface area compared to the top surface area, allowing for the assessment of whether the insole design meets the criteria of having a sufficiently large bottom surface. This information can be used to determine if the design should be accepted or rejected based on the specified requirements.

Bottom surface curvature

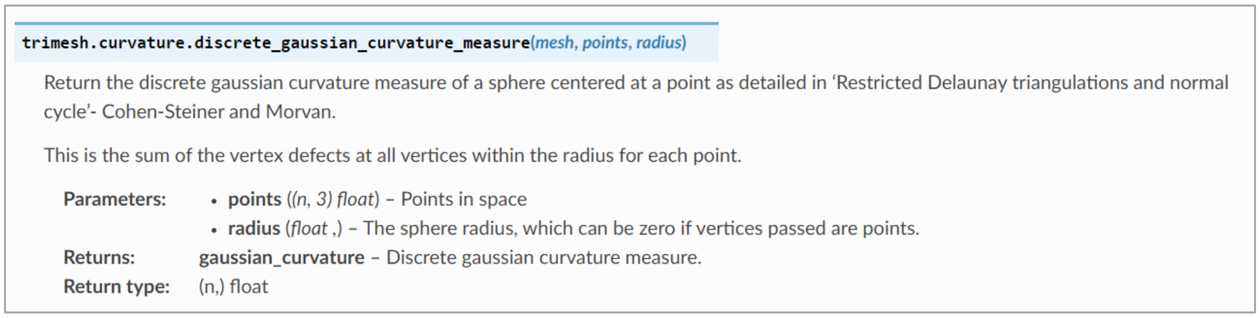
The bottom surface of the insole plays a critical role in the 3D printing process as it directly interacts with the conveyor belt. It is crucial for this surface to be perfectly flat to ensure a reliable and accurate printing outcome. A flat bottom surface ensures that the insole remains stable and securely adheres to the conveyor belt throughout the printing process. This stability prevents any shifting or displacement of the insole, which could lead to print defects or misalignment. A flat surface allows for consistent layer deposition during the printing process. If the bottom surface is uneven or has irregularities, it can affect the layer height and result in an inconsistent print. This can impact the overall quality and structural integrity of the insole.

By specifying a maximum tolerance of ±0.1mm for flatness irregularities, the design ensures that only insoles with an acceptably flat underside are printed. Deviations beyond this threshold indicate significant irregularities that could affect print quality and performance.

STL

```
def check_if_bottom_surface_is_flat(mesh: trimesh):
    up_axis = 2 # z vertical axis
    bottom_threshold = -0.99
    radius = 0.9

    bottom_mask = mesh.vertex_normals[:, up_axis] < bottom_threshold
    mean_curvature = 
trimesh.curvature.discrete_gaussian_curvature_measure(mesh, mesh.vertices, 
radius)
    mean_curvature[~bottom_mask ] = 0 # we are not interested in the curvature 
of the upper part
    pv.wrap(mesh).plot(scalars=mean_curvature)
```

(trimesh.curvature, 2023)



trimesh.curvature.discrete_gaussian_curvature_measure(*mesh, points, radius*)

Return the discrete gaussian curvature measure of a sphere centered at a point as detailed in 'Restricted Delaunay triangulations and normal cycle'- Cohen-Steiner and Morvan.

This is the sum of the vertex defects at all vertices within the radius for each point.

Parameters:
- **points** ((*n, 3*) *float*) – Points in space
- **radius** (*float* ,) – The sphere radius, which can be zero if vertices passed are points.

Returns: **gaussian_curvature** – Discrete gaussian curvature measure.

Return type: (n,) float

The provided code performs the following steps to check if the bottom surface of a given mesh is flat. Define Variables: The **up_axis** specifies the vertical axis, where 2 represents the Z-axis in the coordinate system. The **bottom_threshold** sets a threshold value below which vertices are considered part of the bottom surface. The **radius** defines the radius used for calculating mean curvature. Next is the creation of bottom surface mask. **bottom_mask** evaluates the vertex normals along the vertical axis (**up_axis**) and creates a boolean mask where values below **bottom_threshold** indicate vertices belonging to the bottom surface. Then the computing of mean curvature follows. **mean_curvature** calculates the discrete Gaussian curvature measure for the mesh using the **trimesh.curvature.discrete_gaussian_curvature_measure** function. The curvature measure is computed based on the mesh vertices and the specified **radius**.

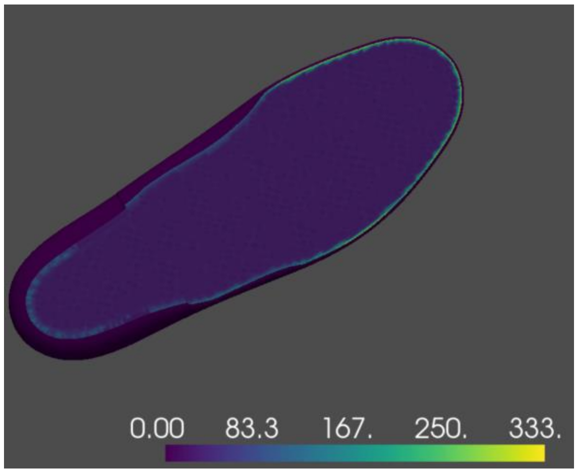
The **discrete_gaussian_curvature_measure** function calculates the discrete Gaussian curvature measure for each vertex in a mesh. Gaussian curvature is a fundamental concept in differential geometry that characterizes the curvature of a surface at a specific point. The discrete Gaussian curvature measure approximates the Gaussian curvature of a surface by computing the angular deficits around each vertex in the mesh. The angular deficit is defined as the difference between the sum of the angles of the neighboring triangles around a vertex and the corresponding angles in the Euclidean plane.

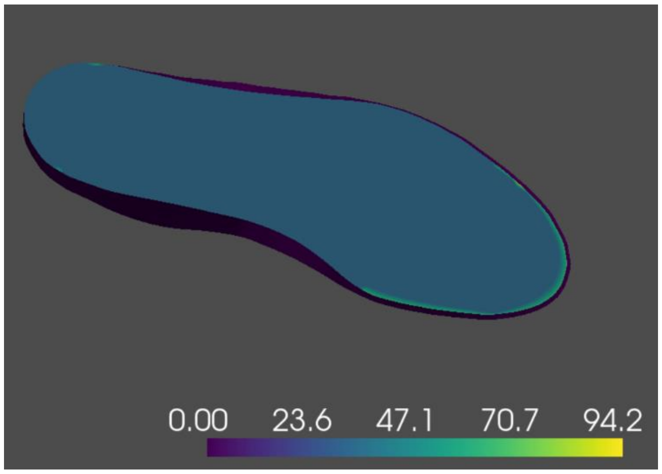
The **discrete_gaussian_curvature_measure** function takes the following parameters:

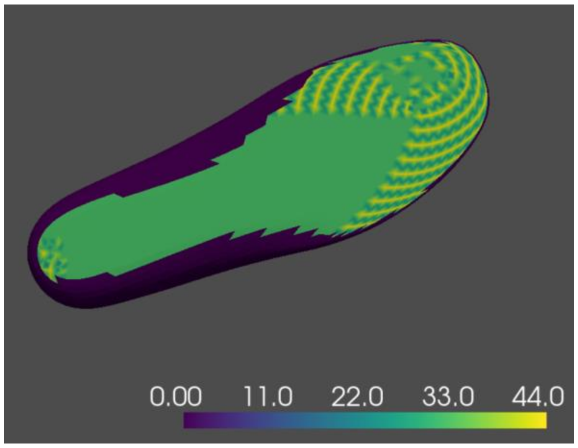
- **mesh**: The input mesh for which the Gaussian curvature is computed.
- **vertices**: The vertices of the mesh, used to determine the neighborhood of each vertex.
- **radius**: The radius that defines the size of the neighborhood for curvature estimation. This parameter determines the number of neighboring vertices used in the curvature calculation.

The function iterates over each vertex in the mesh and computes the discrete Gaussian curvature by summing the angular deficits of the neighboring triangles. The angular deficit is proportional to the Gaussian curvature at that vertex. The resulting Gaussian curvature values are typically normalized or scaled to a specific range for visualization purposes. Higher positive values indicate regions of positive curvature (such as convex areas), while lower or negative values represent regions of negative curvature (such as concave areas). By calculating the discrete Gaussian curvature measure, you can analyze the curvature properties of a mesh's surface, identify regions of high or low curvature, and assess the flatness or curvature of specific areas of interest.

mean_curvature[~bottom_mask] = 0 - sets the curvature values to 0 for vertices that are not part of the bottom surface. This eliminates the curvature values for the upper part of the mesh, focusing only on the bottom surface. Finally, the result is visualized: **pv.wrap(mesh).plot(scalars=mean_curvature)** uses the PyVista library to plot the mesh and assign the calculated mean curvature as scalar values. This visualization helps visualize the curvature distribution across the bottom surface of the mesh.







By executing this code, you can visualize the mean curvature values of the bottom surface of the mesh. Flat surfaces will exhibit lower curvature values, while non-flat surfaces will show higher curvature values. The visualization can provide insights into the flatness of the bottom surface and assist in assessing if it meets the desired flatness criteria.

The images above and on the left show the curvature of the bottom surface area of three different STL 3d files. Higher positive values indicate regions of positive curvature (such as convex areas), while lower or negative values represent regions of negative curvature (such as concave areas). The regions with a constant colour palette indicates a flat surface area.

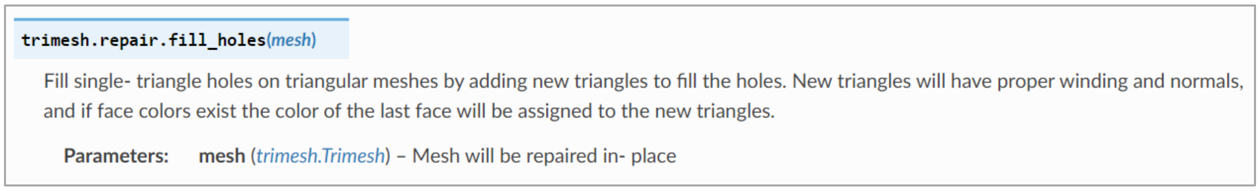
Watertightness of a 3d model

When a 3D model is not watertight, meaning it contains defects or open surfaces in its outer walls, several issues can arise during the exporting process in the slicing software. These issues can have a significant impact on the final output of the 3D model. If the 3D model has defects in its outer walls, the slicing software may struggle to interpret the model correctly. This can lead to errors or inconsistencies in the exported data. The resulting file may contain incomplete or incorrect information, rendering the model unusable or producing unexpected results. In some cases, when a 3D model has open surfaces or defects, the slicing software may fail to export any data at all. If a 3D model with open surfaces or defects manages to be exported successfully, it can still lead to problems during the printing process.

STL

```
def measure_stl_watertightness(mesh: trimesh) -> bool:
    if mesh.is_empty: return False
    if not mesh.is_watertight:
        return mesh.fill_holes() #Check if a mesh has all the properties
required to represent a valid volume, rather than just a surface.
    return mesh.is_watertight
```

(trimesh.repair, 2023)


trimesh.repair.fill_holes(*mesh*)

Fill single- triangle holes on triangular meshes by adding new triangles to fill the holes. New triangles will have proper winding and normals, and if face colors exist the color of the last face will be assigned to the new triangles.

Parameters: **mesh** (*trimesh.Trimesh*) – Mesh will be repaired in- place

The code defines a function called **measure_stl_watertightness** that takes a **mesh** as input, which is expected to be of type **trimesh**. The function returns a boolean value indicating whether the mesh is watertight or not.

Here's a breakdown of how the function works: **if mesh.is_empty: return False**: This line checks if the **mesh** is empty. If it is, it means there are no vertices or faces in the mesh, so the function returns **False** to indicate that it is not watertight. **if not mesh.is_watertight: return mesh.fill_holes()**: This line checks if the **mesh** is not watertight. If it is not watertight, it means there are gaps or holes in the mesh's geometry. In this case, the function calls **mesh.fill_holes()** to attempt to fill these holes and make the mesh watertight. The **fill_holes()** method is a function provided by the **trimesh** library that attempts to close any open surfaces or gaps in the mesh. The function then returns the result of the **fill_holes()** method, which could be a modified version of the original mesh with the holes filled, or it could be a completely different mesh that represents the watertight version of the input mesh. **return mesh.is_watertight**: If the mesh is neither empty nor required to be filled, this line checks if the mesh is watertight. If it is, the function simply returns **True**, indicating that the mesh is watertight.

In summary, this function takes a mesh as input and checks if it is watertight. If the mesh is empty, it returns **False**. If the mesh is not watertight, it attempts to fill any holes in the mesh using the **fill_holes()** method. Finally, it returns **True** if the mesh is watertight or the result of attempting to fill the holes in the mesh.

Conclusion

To expand their workflow and accommodate other CAD programs, PodoPrinter aims to incorporate an additional input that fulfills security and feedback requirements. This research document focuses on the development of a technical solution, called SOLE-Filter, to validate 3D files and assess their printability across multiple formats.

The primary objective of this research is to create a technical solution that accurately evaluates the printability of 3D models. The solution will address the technical requirements for assessing printability, define key criteria for evaluation, and establish best practices for analyzing geometry, topology, and other model features. The proposed solution will be implemented and utilized for the group project.

To achieve accurate printability assessment, the research explores the reading of 3D files, particularly STL and 3DM formats. Python, with the FastAPI framework, is identified as the ideal programming language for developing the validation tool, ensuring efficiency and seamless file validation. For reading 3DM files, the research utilizes the rhino3dm library in conjunction with compute-rhino3dm. On the other hand, the Trimesh library is chosen for reading STL files due to its robust capabilities and suitability for basic geometry.

It is important to note that this research and the development of the technical solution are still ongoing. The project aims to provide a reliable and efficient tool for assessing the printability of 3D models, meeting the specific design requirements provided by the client. The results of this research will contribute to the advancement of SOLE by PodoPrinter's workflow and enable them to provide even better insoles to improve mobility for their customers.

References

Docker Hub. (n.d.). Citated from Docker Hub: https://hub.docker.com/

FastAPI. (n.d.). Citated from FastAPI: https://fastapi.tiangolo.com/

Kaszynski, S. (19 5 2019 r.). *PyVista: 3D plotting and mesh analysis through a streamlined interface for the Visualization Toolkit (VTK)*. Citated from Journal of Open Source Software: https://doi.org/10.21105/joss.01450

Masatomo Inui, N. U. (7 6 2023 r.). *Shrinking Sphere: A Parallel Algorithm for Computing the Thickness of 3D Objects* . Citated from CAD Journal | Volume 13 Number 2: https://www.cad-journal.net/files/vol_13/Vol13No2.html

mcneel/compute.rhino3d: REST geometry server based on RhinoCommon and headless Rhino. (7 6 2023 r.). Citated from GitHub: https://github.com/mcneel/compute.rhino3d

Mesh - compute_rhino3d 0.14.0 documentation. (7 6 2023 r.). Citated from Mesh - compute_rhino3d 0.14.0 documentation: https://compute-rhino3d.readthedocs.io/en/latest/Mesh.html

Nikolov, I. (5 5 2022 r.). *Python Libraries for Mesh, Point Cloud, and Data Visualization (Part 1)*. Citated from Medium: https://towardsdatascience.com/python-libraries-for-mesh-and-point-cloud-visualization-part-1-daa2af36de30

NumPy. (n.d.). Citated from NumPy: https://numpy.org/

PyVista — PyVista 0.39.1 documentation - The PyVista Project. (7 6 2023 r.). Citated from PyVista: https://docs.pyvista.org/version/stable/

The Visualization Toolkit (VTK). (7 6 2023 r.). Citated from VTK: https://vtk.org/

trimesh.bounds. (7 6 2023 r.). Citated from trimesh.bounds - trimesh 3.22.0 documentation: https://trimsh.org/trimesh.bounds.html

trimesh.curvature. (7 6 2023 r.). Citated from trimesh.curvature - trimesh 3.22.0 documentation: https://trimsh.org/trimesh.curvature.html

trimesh.proximity. (7 6 2023 r.). Citated from trimesh.proximity - trimesh 3.22.0 documentation: https://trimsh.org/trimesh.proximity.html

trimesh.repair. (7 6 2023 r.). Citated from trimesh.repair - trimesh 3.22.0 documentation: https://trimsh.org/trimesh.repair.html

trimesh.triangles. (7 6 2023 r.). Citated from trimesh.triangles - trimesh 3.22.0 documentation: https://trimsh.org/trimesh.triangles.html

UltiMaker Cura. (30 5 2023 r.). Citated from UltiMaker: https://ultimaker.com/software/ultimaker-cura/

Welcome to Python.org. (7 6 2023 r.). Citated from Python.org: https://www.python.org/