Optimal Part Rotation for Additive Manufacturing with Fused Filament Fabrication

Sameer Puri

August 17, 2019

1 Introduction

Additive manufacturing has made fabricating arbitrary three-dimensional parts more attainable. Fused filament fabrication (FFF) is the most popular method for hobbyist use cases, as other higher quality methods like selective laser sintering (SLS) are orders of magnitude more expensive[1]. The idea behind FFF is simple: the desired material, in filament form, is pushed through a hot nozzle which brings it to its melting point and extruded onto a build plate. Motors control axial movement and extrusion speed, building parts layer by layer from the bottom up. There are many factors that can affect print quality. While most are hardware related, user choices are also at play.

Slicer software

The stereo-lithography (STL) file format is widely-used to approximate parts as polyhedrons with triangular faces[2]. To print a part, its STL file is converted by slicer software into GCode: numerical control instructions for the printer[3]. There are many FFF printer manufacturers, each with their own recommended slicing software. Rather than trying to improve print quality within any particular slicer software, I consider a factor controllable during the post-design phase: part rotation.

Part rotation

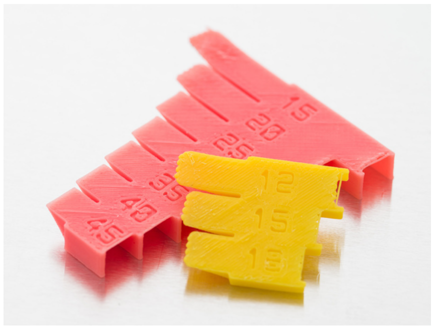
Overhangs are bottom faces of a part unsupported by previous layers or the build plate. They are undesirable in FFF because they print poorly with visible lines as shown in Figure 1(b). The smaller the angle between the build plate and an overhang is, the worse the printed part appears. Although slicing software creates support structures for large overhangs, print quality is still poor. Rotating a part before importing it into slicer software can reduce overhang. Consider a cube printed on a corner – there are three overhang faces. If it was rotated to be flat on a face and then printed, there would be no overhang.

2 Optimization Problem

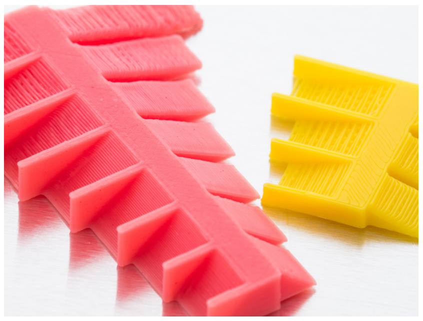
Selecting the best rotation of a print should improve print quality regardless of the FFF printer and slicing software used. Finding it is suitable for formulation as an optimization problem to minimize overhang surface area by rotating the part and has been investigated in previous work[4]. It could also be treated as a computational geometry problem, but it would not be as easy to extend into a generic framework considering multiple factors for orientation optimization.

Variables and Constraints

An STL file can be rotated in 3D space using classic Euler angles. Rotation in the z-axis is unnecessary; overhang will remain the same. Thus, only rotation in the x-axis and y-axis θ_x, θ_y is considered. Rotation is constrained to $(-\pi, \pi)$ in both axes, since any rotation beyond this range just maps back into the range.



(a) Prints face-up with overhang angle from the build plate denoted.



(b) Prints face-down showing how quality worsens at lower angles.

Figure 1: Overhang test prints[7]

Objective Function

Computing surface area

Overhang can be quantified as downward facing surface area after rotation. Conveniently, all faces are triangles so surface area can be computed as $\sum_{i=1}^{n} \sqrt{s_i(s_i - \|a_i\|)(s_i - \|b_i\|)(s_i - \|c_i\|)}$ where (a_i, b_i, c_i) are the sides of triangle t_i as vectors after rotation and the semiperimeter $s_i = \frac{\|a_i\|+\|b_i\|+\|c_i\|}{2}$. To select downward faces in particular, surface area for t_i is multiplied by an overhang bias $H\left[-(a_{i,x}b_{i,y} - a_{i,y}b_{i,x})\right]$. This computes the z-component of the vector normal $a \times b$ where $a_{i,x}$ is the x-component of the a side vector and uses the Heaviside unit step function $H(x)$ to output 1 for downward faces and 0 for upward faces. $H(x)$ is difficult to optimize with as its derivative is the Dirac delta function: 0 at all points except $x = 0$, where it is infinite. An approximation $H(x) \approx \frac{\tanh(x)+1}{2}$ which gives good gradients for optimization is used instead in implementation[8]. As previously discussed, overhang print quality worsens as the angle made with the build plate decreases. This is handled with an efficient trick: compute surface area by taking the vector norm in only the x and y dimensions. Now, the surface area is that of the triangles when projected onto the build plate, effectively giving a linear weighting to the surface area based on its angle from the build plate.

Dealing with bottom faces

The above formulation ignores bottom faces: those part faces that rest flat on the build plate. For instance, when orienting a cube using the above objective function formulation, it will be placed on a corner instead in the above formulation, because placing it on a side would actually be a global maximizer. To correct this, bottom faces must have 0 surface area in the objective function. Formally, a bottom face is any t_i for which $a_{i,z} = b_{i,z} = c_{i,z} = min_z, i \in 1, 2, \ldots, n$ where $min_z = min\left(a_{i,z}, b_{i,z}, c_{i,z}\right) \forall i \in 1, 2, \ldots, n$. This definition can be relaxed to $a_{i,z} + b_{i,z} + c_{i,z} = 3min_z$ and used as a bottom face bias $H\left(a_{i,z} + b_{i,z} + c_{i,z} - 3min_z\right)$ which outputs 1 for non-bottom faces and 0 for bottom faces. $H(x) \approx \tanh(x)$ is used in implementation, as $\tan(0) = 0$ so a part with no overhang should have an objective function value near 0. Thus, the objective function is:

$$\begin{array}{rl} min\, f(\theta_x, \theta_y) = & \sum_{i=1}^{n} H[-(a_{i,x}b_{i,y} - a_{i,y}b_{i,x})] * H(a_{i,z} + b_{i,z} + c_{i,z} - 3min_z) * \sqrt{s_i(s_i - \|a_i\|)(s_i - \|b_i\|)(s_i - \|c_i\|)} \\ subject\, to & \theta_x \in (-\pi, \pi) \\ & \theta_y \in (-\pi, \pi) \end{array}$$

where for triangle t_i with vertices $t_{i,1}, t_{i,2}, t_{i,3}$:
$t'_{i,j}$ is the rotation about classic Euler angles θ_x, θ_y in the x and y axis respectively:

$$t'_{i,j} = rotate\left(t_{i,j}, \theta_x, \theta_y\right) = \left[\begin{array}{c} \cos(\theta_x)(t_{i,j,x}\cos(\theta_y) + \sin(\theta_y)(t_{i,j,y}\sin(\theta_x) - t_{i,j,z}\cos(\theta_x))) + \sin(\theta_z)(t_{i,j,y}\cos(\theta_x) + t_{i,j,z}\sin(\theta_x)) \\ \cos(\theta_z)(t_{i,j,y}\cos(\theta_x) + t_{i,j,z}\sin(\theta_x)) - \sin(\theta_z)(t_{i,j,x}\cos(\theta_y) + \sin(\theta_y)(t_{i,j,y}\sin(\theta_x) - t_{i,j,z}\cos(\theta_x))) \\ t_{i,j,x}\sin(\theta_y) + \cos(\theta_y)(t_{i,j,z}\cos(\theta_x) - t_{i,j,y}\sin(\theta_x)) \end{array}\right]$$

and $a_i = t'_{i,2} - t'_{i,1}, b_i = t'_{i,3} - t'_{i,1}, c_i = t'_{i,3} - t'_{i,2}$.

Implementation

The objective function is implemented in *objective_function.py* using NumPy. Although expensive with the initial naive version, an efficient version which precomputes useful products was derived. The Mathematica notebook *calculations.nb* shows derivation of the efficient version. It could be easily converted to work on GPUs with PyTorch, but this was out of the scope of the project. 1,000 function evaluations complete in under 5 seconds for all evaluated parts.

Numerical Inaccuracy

Early in the project, a numerically stable Heron's formula was used to compute surface area, as I was concerned that the objective function may mislead optimization methods if a part is highly faceted, like a fractal shape. However, few if any FFF printers available to consumers can achieve accuracy at the micrometer level, so the final implementation is not numerically stable.

3 Optimization Method

Unfortunately, methods discussed in class turned out to be inapplicable. The function is not convex; there are many, many local extrema even for a cube as shown in Figure 2(a). Finding even the first derivative of the objective function was not feasible; there are dozens of terms in the derivative due to the rotations and it becomes more complex when the biases are considered. I investigated methods for derivative-free constrained global optimization and found several candidate algorithms[5]: differential evolution, dual annealing, multilevel coordinate search (MCS), and dividing rectangles (DIRECT). Because the objective function is efficient and I desired a true global minimizer, I selected DIRECT. Dividing Rectangles is a deterministic global search method that, as its name indicates, divides the search space into hyperrectangles. The constraints are scaled so the algorithm begins with a single unit hypercube which is iteratively split into smaller and smaller hyperrectangles. Those that may lead to minimizers are selected and split. A global search is effected by ensuring that the largest level of rectangles is divided in each iteration.

DIRECT Implementation

The algorithm was implemented from scratch using the user guide[6]. It is included in the electronic submission as *direct.py*. Figure 2(c) shows how the DIRECT method converges for the objective function using a unit cube. An arbitrary limit of 1,000 function evaluations is set to ensure that a minimizer is returned within a reasonable amount of time.

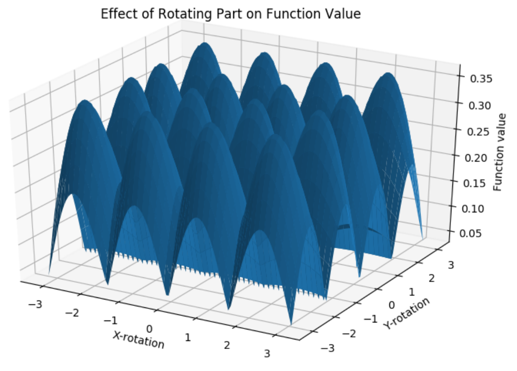
4 Program Usage

The program can be started from *main.py*. It requires the first argument to be a path to an STL file followed by a subcommand.

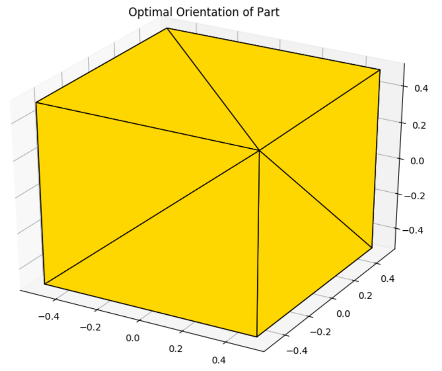
Orient

The *orient* command runs the DIRECT algorithm on the objective function and prints out the best angle and objective function value found for the input STL file:

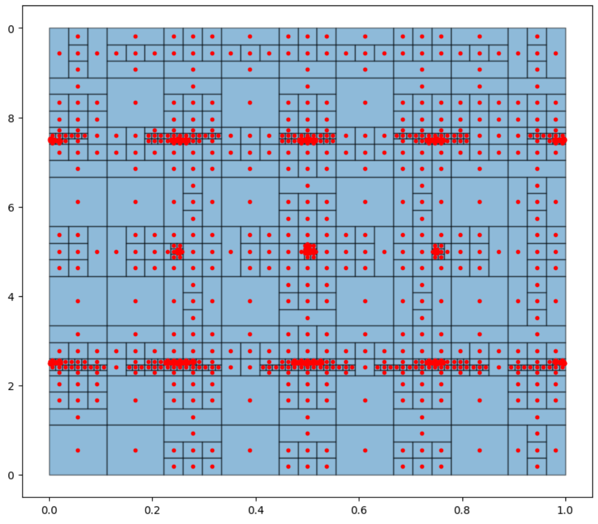
```
> python main.py examples/cube.stl orient
File "examples/cube.stl" oriented with angles [0.  0.]  and value 0.0347 in 1637 ms
```



(a) Graph of objective function values. There are recurring peaks and valleys due to the cube's symmetry.



(b) The unit cube after the suggested rotation



(c) Plot of the DIRECT algorithm hyperrectangles and their centers after 1,025 function evaluations. The algorithm discovers hyperrectangles of the function where the value decreases and chooses to divide them, effectively creating a heatmap of minimizer location.

Figure 2: Plots related to the rotation of a unit cube.

Orientplot

The *orientplot* command does the same thing as the orient command but also displays the part after the recommended rotation, as shown in Figure 2(b).

```
> python main.py examples/cube.stl orientplot
File "examples/cube.stl" oriented with angles [0.  0.]  and value 0.0347 in 1094 ms
```

Plot

The *plot* command plots the objective function value as shown in Figure 2(a) with a grid of 100x100 points, or 10,000 function evaluations.

```
> python main.py examples/cube.stl plot
Plotting function value for examples/cube.stl Minimum value of 0.0347 found
```

5 Evaluation

For all tested parts, the implementation found the global minimizer within the prescribed 1,000 function evaluation limit. The tested parts are available in the *examples* folder. Further investigation is needed to confirm that this limit is appropriate.

References

[1] MANUFACTUR3D. *A Comprehensive List of All 3D Printing Technologies.* `https://manufactur3dmag.com/comprehensive-list-all-3d-printing-technologies`, November 2018.

[2] All3Dp. *STL File Format (3D Printing) – Simply Explained.* `https://all3dp.com/what-is-stl-file-format-extension-3d-printing/`, February 2019.

[3] Jakk. *What is Slicing Software, and what does it do?.* `https://www.goprint3d.co.uk/blog/what-is-slicing-software-and-what-does-it-do/`, July 2016.

[4] Christoph Schranz. *Tweaker - Auto Rotation Module for FDM 3D Printing.* `https://www.researchgate.net/publication/311765131_Tweaker_-_Auto_Rotation_Module_for_FDM_3D_Printing`, December 2016.

[5] Rios, Luis & Sahinidis, Nikolaos. Derivative-free optimization: A review of algorithms and comparison of software implementations. In *Journal of Global Optimization* (November 2009), Springer, pp. 1247-1293. DOI 10.1007/s10898-012-9951-y.

[6] Finkel, Daniel. *DIRECT Optimization Algorithm User Guide.* `http://www2.peq.coppe.ufrj.br/Pessoal/Professores/Arge/COQ897/Naturais/DirectUserGuide.pdf`, March 2003.

[7] Hsiao, Walter. *Overhang Test Print (Customizable).* `https://www.thingiverse.com/thing:58218`, March 2013.

[8] Hill, Scott. *Heaviside Step Function, Analytic Approximations.* `https://en.wikipedia.org/wiki/Heaviside_step_function#Analytic_approximations`, March 2006.

Appendix

main.py

```
from typing import List, Callable, Tuple
import math
import sys
import time

from scipy import optimize
import numpy as np
import stl

from objective_function import build_f
from plot import plot_f, plot_stl
from direct import direct


def orient_stl(f: Callable[[List[float]],float], debug: bool) -> Tuple[List[
    float],float]:
    res: Tuple[List[float],float] = optimize.minimize(f, [0, 0], method=direct
        , bounds=np.array([[-math.pi, math.pi], [-math.pi, math.pi]]), options
        =dict(maxfev=1000))
    return (res.x, res.fun)


if __name__ == '__main__':
    debug = False
    if len(sys.argv) != 3:
        print('Usage: python main.py STL_FILENAME (orient|plot|orientplot)')
    else:
        filename = sys.argv[1]
        command = sys.argv[2]
        mesh = stl.mesh.Mesh.from_file(filename)
        f = build_f(mesh, debug)
        start = time.time()
        if command == 'plot':
            print(f'Plotting function value for {filename}')
            plot_f((100,100), f)
        elif command == 'orient':
            theta, value = orient_stl(f, debug)
            finish = time.time()
            print(f'File "{filename}" oriented with angles {np.round(theta, 
                decimals=2)} and value {value} in {round((finish-start)*1000)}
                 ms')
        elif command == 'orientplot':
            theta, value = orient_stl(f, debug)
            finish = time.time()
            print(f'File "{filename}" oriented with angles {np.round(theta, 
                decimals=2)} and value {value} in {round((finish-start)*1000)}
                 ms')
            plot_stl(theta, mesh)
```

objective_function.py

```
from typing import Dict, List, Callable
```

```
import math
import time

import numpy as np
import stl

# All used products of triangle components (v1x*v1y, v1x*v1z, ..., v1x*v3z,
    etc.)
# see my Mathematica notebook for where these come from
def compute_sums_and_products(mesh: stl.mesh.Mesh) -> Dict[str, List[float]]:
    sp: Dict[str, List[float]] = {}
    i: int
    j: int
    k: int
    l: int
    for i in range(3):
        for j in range(3):
            for k in range(3):
                for l in range(3):
                    product_string: str = f'{i+1}{chr(ord("x")+j)}{k+1}{chr(
                        ord("x")+l)}'
                    product_string_rev: str = f'{k+1}{chr(ord("x")+l)}{i+1}{
                        chr(ord("x")+j)}'
                    if i == k or j == l: # Squares (1x1x) are never used,
                        neither are 1x2x or 2x2y forms
                        continue
                    if product_string_rev in sp: # When 2x1y exists, set 1y2x
                        equal to it, instead of redoing
                        sp[product_string] = sp[product_string_rev]
                        continue
                    sp[f'{i+1}{chr(ord("x")+j)}{k+1}{chr(ord("x")+l)}'] = \
                        mesh.vectors[:, i, j] * \
                        mesh.vectors[:, k, l]
    sp['sa'] = sp['1y2x'] - sp['1x2y'] - sp['1y3x'] + sp['2y3x'] + sp['1x3y']
        - sp['2x3y']
    sp['sb'] = sp['1z2x'] - sp['1x2z'] - sp['1z3x'] + sp['2z3x'] + sp['1x3z']
        - sp['2x3z']
    sp['sc'] = sp['1z2y'] - sp['1y2z'] - sp['1z3y'] + sp['2z3y'] + sp['1y3z']
        - sp['2y3z']

    return sp



def build_f(mesh: stl.mesh.Mesh, debug: bool) -> Callable[[List[float]], float
    ]:
    sp: Dict[str, List[float]] = compute_sums_and_products(mesh)
    def f(theta: List[float]) -> float:
        if debug:
            start: float = time.time()

        sinx: float = math.sin(theta[0])
        cosx: float = math.cos(theta[0])
        siny: float = math.sin(theta[1])
        cosy: float = math.cos(theta[1])
        cosxcosy: float = cosx*cosy
        cosysinx: float = cosy*sinx

        S: List[float] = sp['sa']*cosxcosy + sp['sb']*cosysinx + sp['sc']*siny

        z_height: List[float] = mesh.vectors[:,:,2]*cosxcosy - mesh.vectors
            [:,:,1]*cosysinx + mesh.vectors[:,:,0]*siny
        z_min: float = np.min(z_height)

        overhang_bias: List[float] = (1 + np.tanh(S))

        bottom_face_bias: List[float] = np.tanh((np.sum(z_height ,axis=1) - 3*
            z_min)/10)


        value: float = .25 * np.sum(np.abs(S) * overhang_bias *
            bottom_face_bias)

        if debug:
            finish: float = time.time()
            print(f'Computed f({theta})={value} in {round((finish-start)*1000)
                } ms')

        return value
    return f
```

plot.py

```
import math
from typing import List, Tuple, Callable

import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits import mplot3d
import stl



def plot_f(resolution: Tuple[float,float], f):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_title(f'Effect of Rotating Part on Function Value')
    ax.set_xlabel('X-rotation')
    ax.set_ylabel('Y-rotation')
    ax.set_zlabel('Function value')

    x_rotation = np.linspace(-math.pi, math.pi, resolution[0])
    y_rotation = np.linspace(-math.pi, math.pi, resolution[1])
    x_rotation_mesh, y_rotation_mesh = np.meshgrid(x_rotation, y_rotation,
        indexing='ij')

    f_of_t = np.array([[f([x, y, 0]) for x, y in zip(x_row, y_row)] for x_row,
         y_row in zip(x_rotation_mesh, y_rotation_mesh)])
    print(f'Minimum value of {np.amin(f_of_t)} found')
    s = ax.plot_surface(x_rotation_mesh, y_rotation_mesh, f_of_t)
    plt.show()

def plot_stl(theta: List[float], mesh: stl.mesh.Mesh):
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    ax.set_title(f'Optimal Orientation of Part')

    mesh = stl.mesh.Mesh(mesh.data.copy())
    mesh.rotate([1, 0, 0], theta[0])
    mesh.rotate([0, 1, 0], theta[1])
    if len(theta) > 2:
        mesh.rotate([0, 0, 1], theta[2])
    stl_polygons = mplot3d.art3d.Poly3DCollection(mesh.vectors)
    stl_polygons.set_facecolor('gold')
    stl_polygons.set_edgecolor('black')
    ax.add_collection3d(stl_polygons)
    scale = mesh.points.ravel()
    ax.auto_scale_xyz(scale, scale, scale)
    plt.show()
```

direct.py

Not included due to length. Available at `https://github.com/sameer/orient-stl`.