Method used in creating a parametric OpenSCAD model of a progressive twist for use in an ECM rifling mandrel

Chad Davis, nostr: chad@ungovernable.art, 𝕏: @mrchaddavis

December 25, 2024

Abstract

As the quality of consumer Fused Deposition Modeling Additive (FDMA) 3D printers has advanced, significant efforts have been made to lower the barriers to firearm manufacturing at home. While 3D-printed plastic is often inappropriate for components like barrels, 3D-printable tools and jigs have emerged to enable fabrication using ubiquitous hardware items of suitable materials and common hand tools. Electrochemical Machining (ECM) has become a viable method for boring and rifling barrels—processes traditionally requiring expensive, high-precision equipment beyond the reach of hobbyists.

Jeff Rodriguez's Liberator12k project pioneered the use of 3D-printed ECM fixtures for barrel production, a concept later refined by IvanTheTroll in the FGC-9 and its successor, the FGC-9 MKII. The MKII introduced the use of a progressive twist rifling—a feature prized for its ballistic benefits but rarely seen due to the complexity and cost of conventional manufacturing methods. With ECM, however, machining a progressive twist barrel is as straightforward as a constant twist, making this advanced feature accessible to hobbyists.

The remaining challenge lies in the complexity of designing the CAD model. This work aims to address this gap by developing a parametric OpenSCAD model that allows users to easily customize progressive twist features by modifying a few key parameters. This approach empowers hobbyists to integrate advanced rifling geometries into their projects with minimal technical barriers.

1 Defining the twist functions

1.1 Measuring the twist of the FGC-9 MKII mandrel

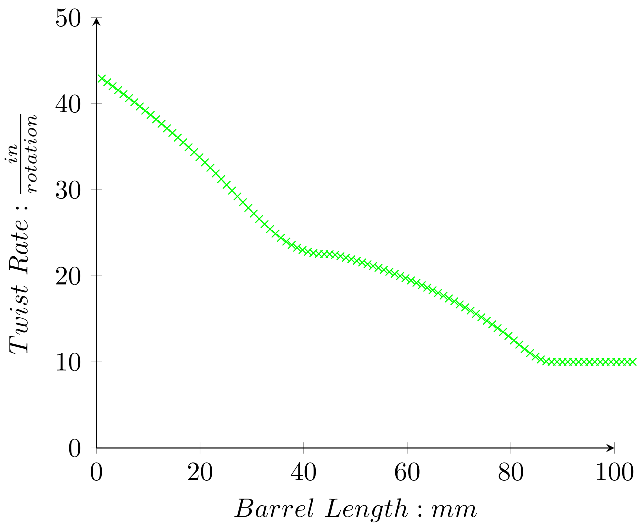
The FGC-9 MKII release included a STEP file for the ECM mandrel, which was imported into FreeCAD, an open-source CAD modeling tool. The following steps were used to measure the twist rate:

1. **Aligning and Centering:**
   - The mandrel was aligned and centered along the Z-axis.
   - A cylinder, also centered on the Z-axis, was added with a height greater than the mandrel and a radius intersecting the wire channel
2. **Boolean Cut Operation:**
   - The cylinder and mandrel were selected, and a Boolean cut operation was performed.
   - This operation created an edge on the wire channel that followed the twist.
3. **Extracting Edge Data:**
   - Using the Curves Workbench in FreeCAD, the edge was discretized (`Curves > Discretize`) to create a series of points along the edge.
   - These points were exported to a text file (`Points > Export Points`) in ASC format.

4. **Calculating Twist Rate:**
   - The coordinates of the points were used to calculate the change in rotation (Δrotation) around the Z-axis between consecutive points.
   - Dividing Δrotation by the change in z (Δz) gave the twist rate at each point.

$$TwistRate = \frac{\text{atan2}(y_{n+1}, x_{n+1}) - \text{atan2}(y_n, x_n)}{z_{n+1} - z_n} \frac{in}{rotation}$$

x	y	z	$\frac{in}{rotation}$
-0.597759	-4.3693	21.5974	42.92126245
-0.571098	-4.37287	22.6449	42.48180185
-0.544131	-4.3763	23.6923	42.03192352
-0.516847	-4.37961	24.7397	41.57343646
-0.489235	-4.38278	25.7871	41.11076139
...	...	...	...
4.01473	-1.82483	118.842	10.00378569
4.06041	-1.7208	119.884	10.00450163
4.1034	-1.61562	120.925	9.994106839
4.14366	-1.50936	121.967	10.00328837
4.18117	-1.40211	123.008	9.994649931



Observations

- The calculated twist rates showed some deviation from a smooth progression.
- These deviations may result from imperfections in the original CAD file, inaccuracies introduced during file conversion, or irregularities in the cross-section profile.
- Despite this, the final twist rate of approximately 1:10 (suitable for 9mm parabellum projectiles) aligns with expectations, indicating sufficient accuracy for practical use.

1.2 Creating a function for a parametric twist

A simple linear function was chosen to model the twist progression :

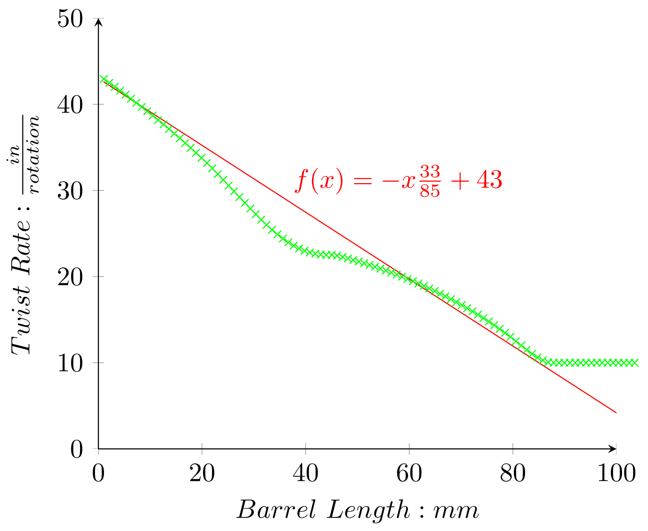
$$f(x) = -x\frac{g}{l} + b$$

Where:

- g is the difference (the "gain") between the twist rate at the breech (b) and muzzle (m): $g = b - m$
- l is the length over which the twist progresses

Using the measured data:

- $m = 10$
- $b = 43$
- $g = b - m = 43 - 10 = 33$
- $l = 85$



The selected formula and values present a decent fit for the beginning and end of the twist and provides a smooth transition through the observed deviation. Additionally, the common unit, inches/rotation, may exaggerate the deviation due to not being a linear unit—the difference between a twist rate of 1:43 to 1:42 is significantly less than the difference between 1:9 to 1:10. Inverting this unit ($\frac{inch}{rotation}$ to $\frac{rotation}{inch}$) will give a more intuitive plot for understanding the gain rate. We will call this hyperbolic curve $h(x)$.

$$h(x) = f(x)^{-1}$$

$$h(x) = \left(\frac{-x\frac{33in}{85mm} + 43in}{rotation} \right)^{-1} = \frac{rotation}{-x\frac{33in}{85in} + 43in}$$

OpenSCAD uses millimeters, rather than inches, as a native unit, so we convert to that:

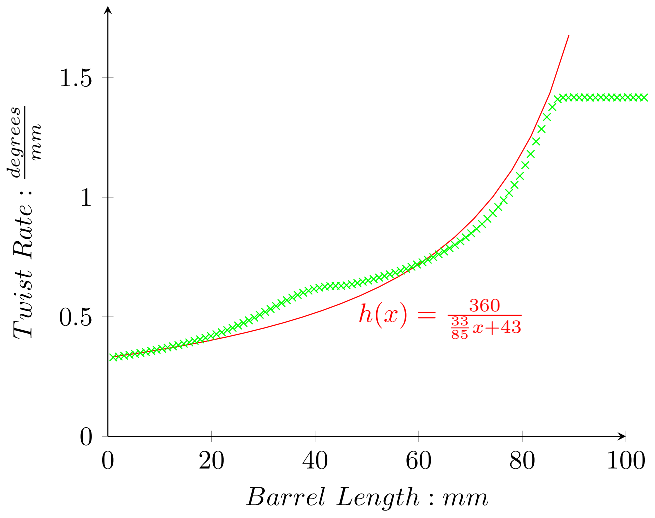
$$= \frac{rotation}{-x\frac{33*\frac{25.4mm}{in}}{85mm} + 43in * \frac{25.4mm}{in}} = \frac{rotation}{-x\frac{838.2mm}{85mm} + 1092.2mm}$$

Also, OpenSCAD uses degrees in the transformations and trigonometric functions, so we convert rotation to degrees:

$$= \frac{rotation}{-x\frac{838.2mm}{85mm} + 1092.2mm} * \frac{360°}{rotation} = \frac{360°}{-x\frac{838.2mm}{85mm} + 1092.2mm}$$

or generally as:

$$h(x) = -x\frac{360}{g+b}\frac{degrees}{mm}$$

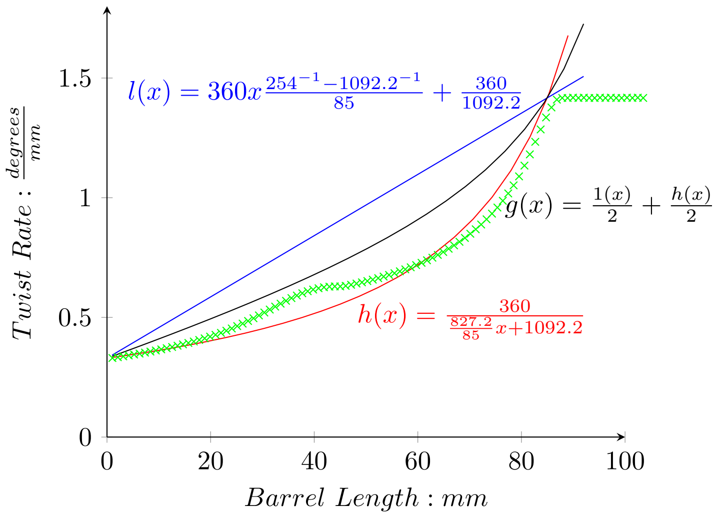


Anticipating a desire to experiment with rate at which the gain increases, an alternate function was chosen—a linear rate gain would, perhaps, distribute the torque exerted on the projectile more evenly. We call this function $l(x)$. We adjust the rate gain to any curve between $h(x)$ and $l(x)$ with the user supplying a value to take a weighted average.

$$l(x) = 360x\frac{g}{l} + \frac{360}{b}$$

And our function to combine the two functions into the final gain profile (p) where $0 < p < 1$:

$$g(x) = l(x)(1-p) + h(x)p$$



An interactive graph may be found at

`https://www.desmos.com/calculator/hhnmuiw27l`

Our initial attempt was to use these functions to iterate steps over the length of the gain twist and then use the progressive sum to position the twist at each step. However, the twist progresses over the length so each step was rotated slightly less than it should have been. It was incorrectly assumed that this error would be negligible.

A common way to work with objects in OpenSCAD is to use lower resolutions while drafting to reduce the render time when making frequent changes and increasing that resolution for the final render. For instance, the $circle()$ object is actually an $\$fn$-sided polygon where $\$fn$ is set to a low value, reducing the number of point, but generally maintaining the same geometry when $\$fn$ is set higher for a final render.

In our attempt, when the length of the iterated step was changed there there was a very noticeable shift in the total rotation. This could cause significant problems if the object was aligned with additional static objects in a design.

The solution was to change the functions to be integrals of our previous functions.

For our linear gain rate:

$$L(x) = \int 360x\frac{\frac{1}{m} - \frac{1}{b}}{l} + \frac{360}{b}dx$$

$$= 180x^2\frac{\frac{1}{m} - \frac{1}{b}}{l} + x\frac{360}{b} + C$$

For our hyperbolic gain rate:

$$H(x) = \int \frac{360}{-x\frac{g}{l} + b}dx$$

$$= -360\frac{l}{g}ln(|-x\frac{g}{l} + b|) + C$$

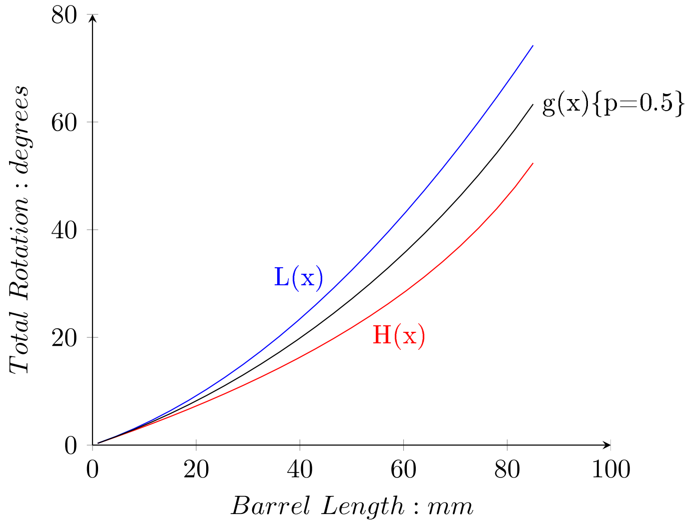
We evaluate $H(0)$ to find C

$$C = H(0) = -360\frac{l}{g}ln(|b|)$$

$$H(x) = -360\tfrac{l}{g}(ln(|-x\tfrac{g}{l} + b|) - ln(|b|))$$

For the weighted average between the two we redefine g(x):

$$g(x) = L(x)(1 - p) + H(x)p$$



An interactive graph may be found at

`https://www.desmos.com/calculator/wseu05ihcp`

And finally we need a function for the constant rotation at the muzzle end. This will be a simple line with a slope of m. But we need to start where the gain twist stops.

$$M(x) = m(x - l) + g(l)$$

1.3 Using these functions in an OpenSCAD script

The twist functions derived earlier were implemented in OpenSCAD to generate a parametric model of the rifling twist. The following constants and functions allow users to define and control the twist parameters effectively.

```
twistLength = 104;          // Total length of the rifling (mm)
muzzleTwistLength = 19;     // Length of constant twist at the muzzle (mm)
gainTwistLength = twistLength - muzzleTwistLength;
breechTwistRate = 1092.2;  // Initial twist rate at the breech (degrees/mm)
muzzleTwistRate = 254;      // Final twist rate at the muzzle (degrees/mm)
gain = breechTwistRate - muzzleTwistRate;
gainProfile = .5;           // Weighting factor for linear vs. hyperbolic profile
```

Implementing Twist Functions

1. **Linear Gain Rotation**

```
function linearGainRotation(z) = 180*z^2*(muzzleTwistRate^-1-breechTwistRate^-1)/
    gainTwistLength+360*z/breechTwistRate;
```

2. **Hyperbolic Gain Rotation:**

   The know working gain rate.

```
function hyperbolicGainRotation(z) = -360*gainTwistLength/gain*(ln(z*-gain/
    gainTwistLength+breechTwistRate)-ln(breechTwistRate));
```

3. **Combined Gain Rotation:**

   A weighted average of the linear and hyperbolic profiles allows users to balance smoothness and precision:

```
function gainRotation(z) = (linearGainRotation(z)*(1-gainProfile)) +
    hyperbolicGainRotation(z)*gainProfile;
```

4. **Muzzle Rotation:**

   The rotation for the constant twist near the muzzle builds on the cumulative rotation from the gain twist:

```
function muzzleRotation(z) = (z-gainTwistLength)*muzzleTwistRate^-1*360+gainRotation(
    gainTwistLength);
```

5. **Total Rotation:**

   This function calculates the rotation angle at any point along the Z-axis, switching between gain twist and muzzle twist as needed:

```
function rotation(z) = z <= gainTwistLength ? gainRotation(z):muzzleRotation(z);
```

Usage

The *rotation*(z) function outputs the cumulative rotation in degrees for a given z-coordinate of the progressive twist described by our chosen constants.

2 Building a polyhedron

Existing 3D-printed ECM fixture designs contain two intersecting channels that follow the twist—one for the cathode wire and one for the electrolytic fluid to flow. To implement each we create a module *channel*() that can be called for each with the distinct attributes as inputs.

```
module channel(radius,sideCount,centerOffset){
}
```

- **Inputs**:
  - *radius*: The distance from the center to the vertex of the regular polygon that is the cross section of the channel.
  - *sideCount*: The number of sides of the regular polygon that is the cross section of the channel.
  - *centerOffset*: The distance between $(0,0)$ and the center the regular polygon that is the cross section of the channel.

The geometry of the channel is generated with an OpenSCAD *polyhedron*(). We will define two parameters

- *points*: A list of coordinates for all vertices of the channel.
- *faces*: A list of each face defined by its vertices as the indices of the *points* list.

2.1 Points

Defining the Points of the Cross Section

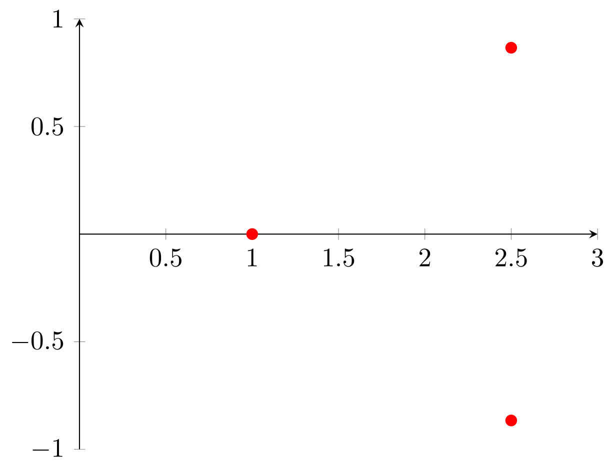
We create the function *polygonPoints*() to generate a list of 2D points for a regular polygon, given its radius, number of sides, and offset from the rotation axis.

```
function polygonPoints(radius,sideCount,centerOffset) =
                              [for(i=[0:sideCount-1])
                                  [
                                      radius * sin(360/sideCount*i-90) + centerOffset ,
                                      radius * cos(360/sideCount*i-90)
                                  ]
                              ];
```

- Inputs:
  - radius: The distance from the polygon's center to its vertices.
  - sideCount: The number of sides in the polygon.
  - centerOffset: The offset of the polygon's center from the rotation axis.
- Output: A list of 2D points defining the polygon.

Example: For a triangular polygon with radius 1 and center offset 2:

```
echo(polygonPoints(1,3,2)); //ECHO: [[1, 0], [2.5, 0.866025], [2.5, -0.866025]]
```



Translating and Rotating Points Along the Z-Axis

We locate each vertex with the function *translatePoint*(). The rotation is determined by passing the z input to the *rotation*() function. Then trigonometric functions are applied to located the x and y coordinates.

```
function translatePoint(point,z) = [
                                cos(rotation(z)) * point.x - sin(rotation(z)) * point.y ,
                                cos(rotation(z)) * point.y + sin(rotation(z)) * point.x ,
                                z
                            ];
```

- Inputs:
  - *point*: A 2D point [x,y].
  - z: The Z-axis position of the point.
- Output: A transformed 3D point [x_1,y_1,z].

Creating an List of Points

The list *channelPoints* stores all 3D points for the polyhedron by iterating through the Z-axis in increments of the user-defined constant *twistStepLength*:

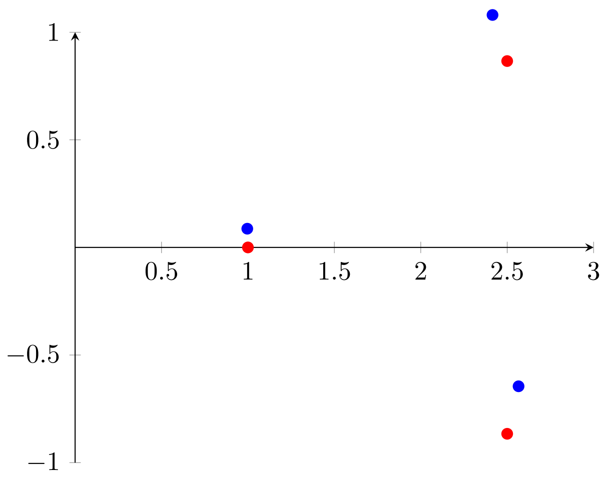
```
twistStepLength = 1;
```

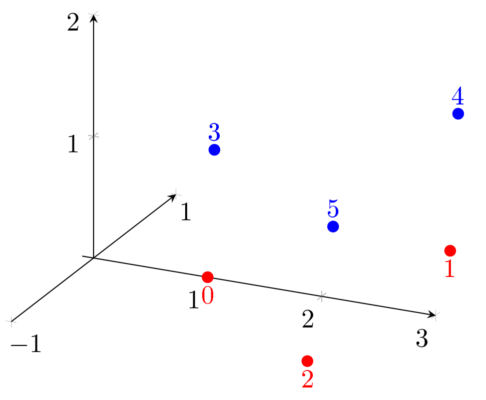
channelPoints is a list that iterates through every z value in steps of *twistStepLength* and through each point from the output of *polygonPoints*() and pass it to *translatePoint*() to obtain the location of all vertices.

When iterating up to *twistLength* by an increment for which it is not evenly divisible by *twistStepLength*, the for loop will stop short by a value of *twistLength* mod *twistStepLength*. So we iterate through *twistLength* − *twistStepLength* and then add one final z value at *twitsLength*. This results in in a slightly longer final step but ensures the total length and final position of the twist is the same at differing values of *twistStepLength*.

```
    channelProfile = polygonPoints(radius,sideCount,centerOffset);
    channelPoints =  [
                            for(z=[0:twistStepLength:twistLength-twistStepLength],i=[0:len(channelProfile)-1])
                                translatePoint(channelProfile[i],z),
                            for(z=twistLength,i=[0:len(channelProfile)-1])
                                translatePoint(channelProfile[i],z)
                          ];
```

radius=2; sideCount=3; centerOffset=2;
echo(channelPoints);
//ECHO: [[1, 0, 0], [2.5, 0.866025, 0], [2.5, -0.866025, 0], [0.996209, 0.0869933, 1], [2.41518, 1.08023, 1], [2.56586, -0.645259, 1]]





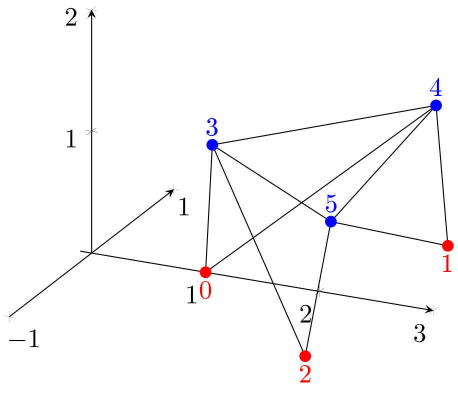
2.2 Faces

The faces parameter for the *polyhedron*() object is a list, in no particular order, of lists of the vertices in each face. *channelPoints* contains all of the vertex coordinates, so the face is defined by a referencing the index of the coordinates in *channelPoints*. The first vertex is arbitrary but each vertex list must be ordered so the vertices are listed clockwise when viewing the face from the outside of the object inward.

The first two faces we define are the top and bottom faces. These faces are each defined by the first and last *sideCount*-points in *channelPoints*. We structure the for loop to list the points is the proper order.

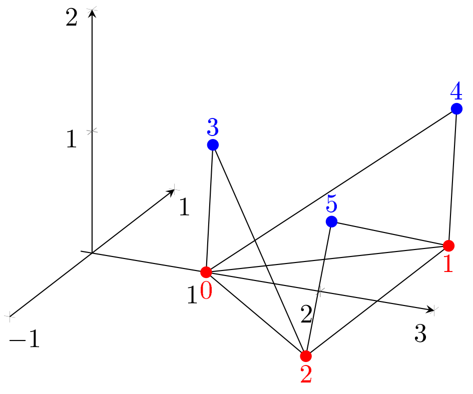
```
    channelTop = [for(i=[len(channelPoints)-sideCount:len(channelPoints)-1])i];
    channelBottom = [for(i=[sideCount-1:-1:0])i];
```

We will divide the remaining faces into two groups. Triangle that "point down" and triangles that "point up". To aid in defining them we fill a table with the points.



channelFaces1

i	point 1	point 2	point 3
0	0	4	3
1	1	5	4
2	2	3	5



channelFaces2

i	point 1	point 2	point 3
0	0	1	4
1	1	2	5
2	2	0	3

We define the values in terms of i and *sideCount*. Point 1 in each table is simply i. Point 3 in *channelFaces*1 is $i + sideCount$. The remaining patterns are completed by checking if $i + 1$ is evenly divided by *sideCount* and using a conditional and the modulo operator to account for the points "wrapping around" the indices of the cross section.

```
    channelFaces1 =  [for(i=[0:twistLength/twistStepLength*sideCount-1])
                        [
                            i,
                            (i+1)%sideCount   ==  0 ? i+1 : i+sideCount+1,
                            i+sideCount
                        ]
                     ];
    channelFaces2 =  [for(i=[0:twistLength/twistStepLength*sideCount-1])
                        [
                            i,
                            (i+1)%sideCount   ==  0 ? i-sideCount+1 :  i+1,
                            (i+1)%sideCount   ==  0 ? i+1 : i+sideCount+1
                         ]
                     ];
```

We combine our the four lists of faces.

```
    channelFaces = concat( [channelTop],[channelBottom] , channelFaces1, channelFaces2 );
```

3 Final details

We create:

- A constant to define how many channels there are concurrently twisting up the axis.

```
grooveCount = 6;
```

- A loop to evenly distribute *grooveCount* channels around the axis.

```
    for( i = [360/grooveCount:360/grooveCount:360])
        rotate([0,0,i])
        polyhedron( points=channelPoints , faces=channelFaces
    );
```

- Constants to define the parameters of the two channels. We compute the radius from the diameter to match the units such as the wire size that the uer will be using to determine these parameters.

```
waterChannelDiameter = 4;
waterChannelRadius = waterChannelDiameter/2;
waterChannelSideCount = 4;
waterChannelOffset = 4.95;
wireChannelDiameter = 1.1;
wireChannelRadius = wireChannelDiameter/2;
wireChannelSideCount = 16;
wireChannelOffset = 2.65;
```

- Calls to the *channel*() module for each channel and appropriate parameters to define the top-level geometry

```
channel(wireChannelRadius,wireChannelSideCount,wireChannelOffset);
channel(waterChannelRadius,waterChannelSideCount,waterChannelOffset);
```

Conclusion

The integration of user-defined parameters allows for flexibility in designing the rifling profile. This approach eliminates many of the technical barriers associated with designing the complex geometry of a progressive twist.

Further testing is required to evaluate the impact of various twist profiles on projectile performance and to refine the model based on experimental results.

The equations and methods presented here aim to balance functionality with simplicity, avoiding unnecessary complexity that could hinder usability. However, as experimentation with these designs continues, new insights may lead to improved models and better outcomes.

Finally, in the spirit of open innovation, this code and methodology have been released into the public domain to encourage experimentation and collaboration within the maker and hobbyist community. By removing barriers to access and sharing this knowledge freely, we hope to inspire continued advancements in home firearm manufacturing and the democratization of advanced machining techniques.

Addendum

Code License

```
twistLength = 104;
muzzleTwistLength = 19;
gainTwistLength = twistLength - muzzleTwistLength;
breechTwistRate = 1092.2;
muzzleTwistRate = 254;
gain = breechTwistRate - muzzleTwistRate;
gainProfile = .94;
twistStepLength = .5;
grooveCount = 6;
waterChannelDiameter = 4;
waterChannelRadius = waterChannelDiameter/2;
waterChannelSideCount = 4;
waterChannelOffset = 4.95;
wireChannelDiameter = 1.1;
wireChannelRadius = wireChannelDiameter/2;
wireChannelSideCount = 16;
wireChannelOffset = 2.65;
function linearGainRotation(z) = 180*z^2*(muzzleTwistRate^-1-breechTwistRate^-1)/gainTwistLength+360*z/breechTwistRate;
function hyperbolicGainRotation(z) = -360*gainTwistLength/gain*(ln(z*-gain/gainTwistLength+breechTwistRate)-ln(breechTwistRate));
function gainRotation(z) = (linearGainRotation(z)*(1-gainProfile)) + hyperbolicGainRotation(z)*gainProfile;
function muzzleRotation(z) = (z-gainTwistLength)*muzzleTwistRate^-1*360+gainRotation(gainTwistLength);
function rotation(z) = z <= gainTwistLength ? gainRotation(z):muzzleRotation(z);
function polygonPoints(radius,sideCount,centerOffset) =
                            [for(i=[0:sideCount-1])
                                [
                                    radius * sin(360/sideCount*i-90) + centerOffset ,
                                    radius * cos(360/sideCount*i-90)
                                ]
                            ];
function translatePoint(point,z) = [
                            cos(rotation(z)) * point.x - sin(rotation(z)) * point.y ,
                            cos(rotation(z)) * point.y + sin(rotation(z)) * point.x ,
                            z
                        ];
module channel(radius,sideCount,centerOffset){
    channelProfile = polygonPoints(radius,sideCount,centerOffset);
    channelPoints =  [
                         for(z=[0:twistStepLength:twistLength-twistStepLength],i=[0:len(channelProfile)-1])
                            translatePoint(channelProfile[i],z),
                        for(z=twistLength,i=[0:len(channelProfile)-1])
                            translatePoint(channelProfile[i],z)
                      ];
    channelTop = [for(i=[len(channelPoints)-1:-1:len(channelPoints)-sideCount])i];
    channelBottom = [for(i=[0:sideCount-1])i];
    channelFaces1 =  [for(i=[0:twistLength/twistStepLength*sideCount-1])
                        [
                            i,
                            (i+1)%sideCount  ==  0 ? i+1 : i+sideCount+1,
                            (i+1)%sideCount  ==  0 ? i-sideCount+1 :  i+1
                        ]
                     ];
    channelFaces2 =  [for(i=[0:twistLength/twistStepLength*sideCount-1])
                        [
                            i,
                            i+sideCount,
                            (i+1)%sideCount  ==  0 ? i+1 : i+sideCount+1
                         ]
                     ];
    channelFaces = concat( [channelTop],[channelBottom] , channelFaces1, channelFaces2 );
    for( i = [360/grooveCount:360/grooveCount:360])
        rotate([0,0,i])
        polyhedron( points=channelPoints , faces=channelFaces
    );
}
channel(wireChannelRadius,wireChannelSideCount,wireChannelOffset);
channel(waterChannelRadius,waterChannelSideCount,waterChannelOffset);
```