The gcodepreview PythonSCAD library*

Author: William F. Adams
willadams at aol dot com

2025/07/4

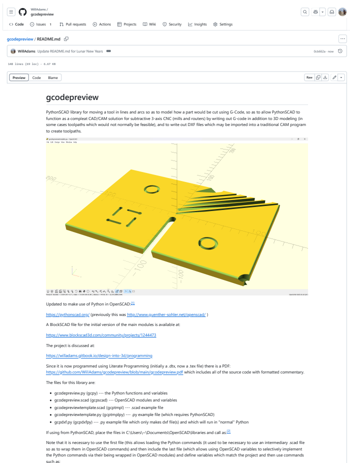
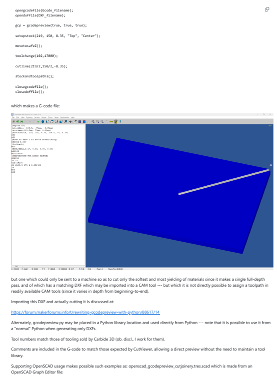
Abstract

The gcodepreview library allows using PythonSCAD (OpenPythonSCAD) to move a tool in lines and arcs and output DXF and G-code files so as to work as a CAD/CAM program for CNC.

Contents

*This file (`gcodepreview`) has version number v0.9, last revised 2025/07/4.

1 readme.md


```
# gcodepreview

PythonSCAD library for moving a tool in lines and arcs so as to model how a part would be cut using G-Code, so as to allow PythonSCAD to function as a compleat CAD/CAM solution for subtractive 3-axis CNC (mills or routers at this time, 4th-axis support may come in a future version) by writing out G-code in addition to 3D modeling (in certain cases toolpaths which would not normally be feasible), and to write out DXF files which may be imported into a traditional CAM program to create toolpaths.

![OpenSCAD gcodepreview Unit Tests](https://raw.githubusercontent.com/WillAdams/gcodepreview/main/gcodepreviewtemplate.png?raw=true)

Updated to make use of Python in OpenSCAD:[^rapcad]

[^rapcad]: Previous versions had used RapCAD, so as to take advantage of the writeln command, which has since been re-written in Python.

https://pythonscad.org/ (previously this was http://www.guenther-sohler.net/openscad/ )

A BlockSCAD file for the initial version of the
main modules is available at:

https://www.blockscad3d.com/community/projects/1244473

The project is discussed at:

https://willadams.gitbook.io/design-into-3d/programming

Since it is now programmed using Literate Programming (initially a .dtx, now a .tex file) there is a PDF: https://github.com/WillAdams/gcodepreview/blob/main/gcodepreview.pdf which includes all of the source code with formatted comments.

The files for this library are:

 - gcodepreview.py (gcpy) --- the Python class/functions and variables
 - gcodepreview.scad (gcpscad) --- OpenSCAD modules and parameters

And there several sample/template files which may be used as the starting point for a given project:

 - gcodepreviewtemplate.scad (gcptmpl) --- .scad example file
 - gcodepreviewtemplate.py (gcptmplpy) --- .py example file
 - gcpdxf.py (gcpdxfpy) --- .py example file which only makes dxf file(s) and which will run in "normal" Python in addition to PythonSCAD
 - gcpgc.py (gcpgc) --- .py example which loads a G-code file and generates a 3D preview showing how the G-code will cut

If using from PythonSCAD, place the files in C:\Users\\\~\Documents\OpenSCAD\libraries [^libraries] or, load them from Github using the command:
```

```

    nimport("https://raw.githubusercontent.com/WillAdams/gcodepreview/refs/heads/main/gcodepreview.py")

[^libraries]: C:\Users\\\~\Documents\RapCAD\libraries is deprecated since RapCAD is no longer needed since Python is now used for writing out files.

If using gcodepreview.scad call as:

    use <gcodepreview.py>
    include <gcodepreview.scad>

Note that it is necessary to use the first file (this allows loading the Python commands and then include the last file (which allows using OpenSCAD variables to selectively implement the Python commands via their being wrapped in OpenSCAD modules) and define variables which match the project and then use commands such as:

    opengcodefile(Gcode_filename);
    opendxffile(DXF_filename);

    gcp = gcodepreview(true, true);

    setupstock(219, 150, 8.35, "Top", "Center");

    movetosafeZ();

    toolchange(102, 17000);

    cutline(219/2, 150/2, -8.35);

    stockandtoolpaths();

    closegcodefile();
    closedxffile();

which makes a G-code file:

![OpenSCAD template G-code file](https://raw.githubusercontent.com/WillAdams/gcodepreview/main/gcodepreview_template.png?raw=true)

but one which could only be sent to a machine so as to cut only the softest and most yielding of materials since it makes a single full-depth pass, and which has a matching DXF which may be imported into a CAM tool --- but which it is not directly possible to assign a toolpath in readily available CAM tools (since it varies in depth from beginning-to-end which is not included in the DXF since few tools make use of that information).

Importing this DXF and actually cutting it is discussed at:

https://forum.makerforums.info/t/rewriting-gcodepreview-with-python/88617/14

Alternately, gcodepreview.py may be placed in a Python library location and used directly from Python --- note that it is possible to use it from a "normal" Python when generating only DXFs as shown in gcpdxf.py.

In the current version, tool numbers may match those of tooling sold by Carbide 3D (ob. discl., I work for them) and other vendors, or, a vendor-neutral system may be used.

Comments are included in the G-code to match those expected by CutViewer, allowing a direct preview without the need to maintain a tool library (for such tooling as that program supports).

Supporting OpenSCAD usage makes possible such examples as: openscad_gcodepreview_cutjoinery.tres.scad which is made from an OpenSCAD Graph Editor file:

![OpenSCAD Graph Editor Cut Joinery File](https://raw.githubusercontent.com/WillAdams/gcodepreview/main/OSGE_cutjoinery.png?raw=true)
```

```

| Version       | Notes        |
| ------------- | ------------ |
| 0.1           | Version  supports setting up stock, origin, rapid positioning, making cuts, and writing out matching G-code, and creating a DXF with polylines.                     |
|               |  - separate dxf files are written out for each tool where tool is ball/square/V and small/large (10/31/23)                                   |
|               |  - re-writing as a Literate Program using the LaTeX package docmfp (begun 4/12/24)                                   |
|               |  - support for additional tooling shapes such as dovetail and keyhole tools                                   |
| 0.2           | Adds support for arcs, specialty toolpaths such as Keyhole which may be used for dovetail as well as keyhole cutters                                   |
| 0.3           | Support for curves along the 3rd dimension, roundover tooling                                   |
| 0.4           | Rewrite using literati documentclass, suppression of SVG code, dxfrectangle                                   |
| 0.5           | More shapes, consolidate rectangles, arcs, and circles in gcodepreview.scad                                   |
| 0.6           | Notes on modules, change file for setupstock                                   |
| 0.61          | Validate all code so that it runs without errors from sample (NEW: Note that this version is archived as gcodepreview-openscad_0_6.tex and the matching PDF is available as well|
| 0.7           | Re-write completely in Python                                   |
| 0.8           | Re-re-write completely in Python and OpenSCAD, iteratively testing                                   |
| 0.801         | Add support for bowl bits with flat bottom                                   |
| 0.802         | Add support for tapered ball-nose and  V tools with flat bottom                                   |
| 0.803         | Implement initial color support and joinery modules (dovetail and full blind box joint modules)                                   |
| 0.9           | Re-write to use Python lists for 3D shapes for toolpaths and rapids.                                   |

Possible future improvements:

 - support for post-processors
 - support for 4th-axis
 - support for two-sided machining (import an STL or other file to use for stock, or possibly preserve the state after one cut and then rotate the cut stock/part)
 - support for additional tooling shapes (lollipop cutters)
 - create a single line font for use where text is wanted
 - Support Bézier curves (required for fonts if not to be limited to lines and arcs) and surfaces

Note for G-code generation that it is up to the user to implement Depth per Pass so as to not take a single full-depth pass as noted above. Working from a DXF of course allows one to off-load such considerations to a specialized CAM tool.
```

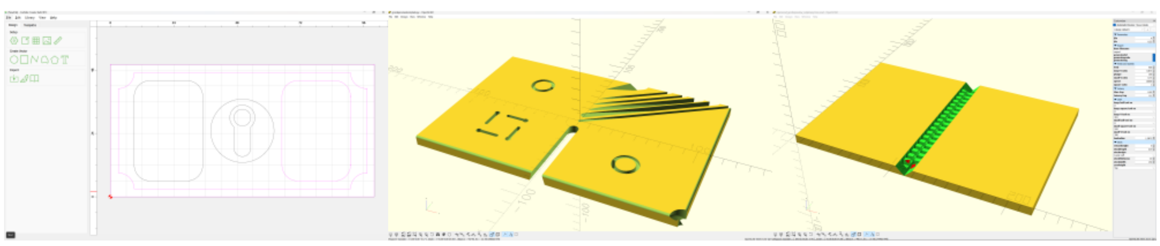
```

To-do:

 - determine why one quadrant of arc command doesn't work in
     OpenSCAD
 - clock-wise arcs
 - add toolpath for cutting countersinks using ball-nose tool from
     inside working out
 - verify OpenSCAD wrapper and add any missing commands for Python
 - verify support for shaft on tooling
 - create a folder of template and sample files
 - clean up/comment out all mentions of previous versions/features/
     implementations/deprecated features
 - fully implement/verify describing/saving/loading tools using
     CutViewer comments

Deprecated feature:

 - exporting SVGs --- coordinate system differences between
     OpenSCAD/DXFs and SVGs would require managing the inversion of
     the coordinate system (using METAPOST, which shares the same
     orientation and which can write out SVGs may be used for future
      versions)
```

2 Usage and Templates

The gcodepreview library allows the modeling of 2D geometry and 3D shapes using Python or by calling Python from within Open(Python)SCAD, enabling the creation of 2D DXFs, G-code (which cuts a 2D or 3D part), or 3D models as a preview of how the file will cut. These abilities may be accessed in "plain" Python (to make DXFs), or Python or OpenSCAD in PythonSCAD (to make DXFs, and/or G-code with 3D modeling) for a preview. Providing them in a programmatic context allows making parts or design elements of parts (*e.g.*, joinery) which would be tedious or difficult (or verging on impossible) to draw by hand in a traditional CAD or vector drawing application. A further consideration is that this is "Design for Manufacture" taken to its ultimate extreme, and that a part so designed is inherently manufacturable (so long as the dimensions and radii allows for reasonable tool (and toolpath) geometries).

The various commands are shown all together in templates so as to provide examples of usage, and to ensure that the various files are used/included as necessary, all variables are set up with the correct names (note that the sparse template in `readme.md` eschews variables), and that if enabled, files are opened before being written to, and that each is closed at the end in the correct order. Note that while the template files seem overly verbose, they specifically incorporate variables for each tool shape, possibly in two different sizes, and a feed rate parameter or ratio for each, which may be used (by setting a tool #) or ignored (by leaving the variable for a given tool at zero (0)).

It should be that the readme at the project page which serves as an overview, and this section (which serves as a collection of templates and a tutorial) are all the documentation which most users will need (and arguably is still too much). The balance of the document after this section shows all the code and implementation details, and will where appropriate show examples of usage excerpted from the template files (serving as a how-to guide as well as documenting the code) as well as Indices (which serve as a front-end for reference).



Some comments on the templates:

- minimal — each is intended as a framework for a minimal working example (MWE) — it should be possible to comment out unused/unneeded portions and so arrive at code which tests any aspect of this project and which may be used as a starting point for a new part/project
- compleat — a quite wide variety of tools are listed (and probably more will be added in the future), but pre-defining them and having these "hooks" seems the easiest mechanism to handle the requirements of subtractive machining.
- shortcuts — as the various examples show, while in real life it is necessary to make many passes with a tool, an expedient shortcut is to forgo the `loop` operation and just use a `hull()` operation and avoid the requirement of implementing Depth per Pass (but note that this will lose the previewing of scalloped tool marks in places where they might appear otherwise)

One fundamental aspect of this tool is the question of *Layers of Abstraction* (as put forward by Dr. Donald Knuth as the crux of computer science) and *Problem Decomposition* (Prof. John Ousterhout's answer to that question). To a great degree, the basic implementation of this tool will use G-code as a reference implementation, simultaneously using the abstraction from the mechanical task of machining which it affords as a decomposed version of that task, and creating what is in essence, both a front-end, and a tool, and an API for working with G-code programmatically. This then requires an architecture which allows 3D modeling (OpenSCAD), and writing out files (Python).

Further features will be added to the templates as they are created, and the main image updated to reflect the capabilities of the system.

2.1 gcpdxf.py

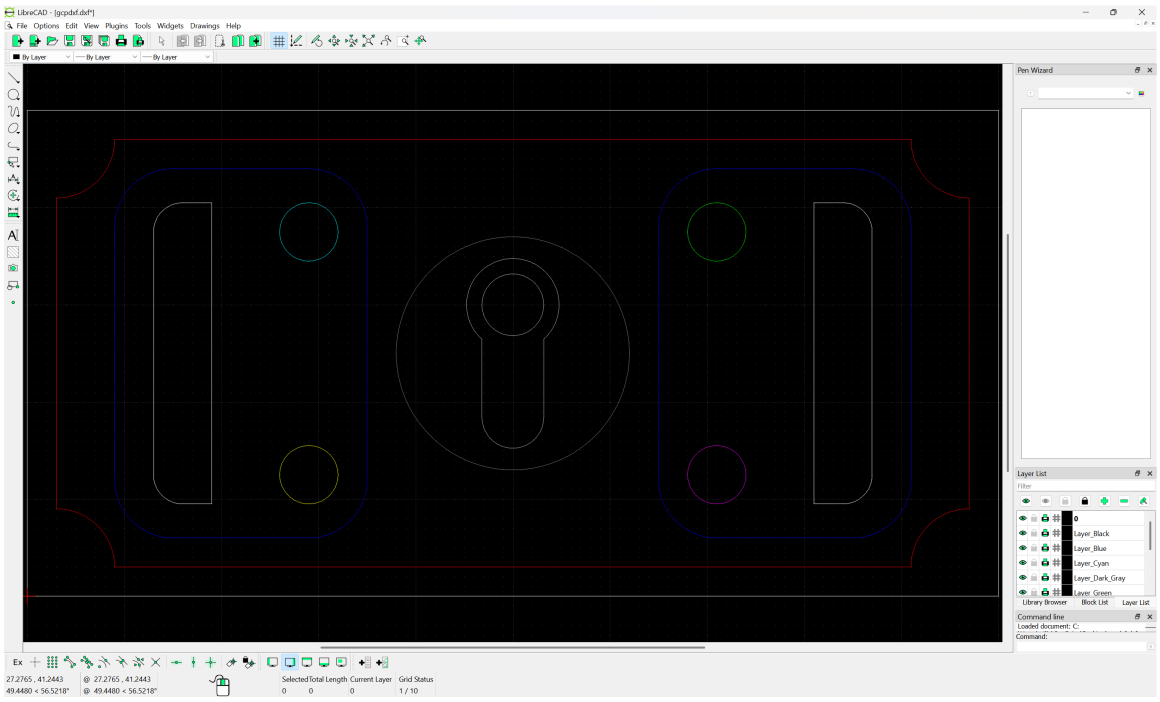
The most basic usage, with the fewest dependencies is to use "plain" Python to create `dxf` files. Note that this example includes an optional command `nimport(<URL>)` which if enabled/uncommented (and the following line commented out), will allow one to use OpenPythonSCAD to import the library from Github, sidestepping the need to download and install the library locally into an installation of OpenPythonSCAD. Usage in "normal" Python will require manually installing the `gcodepreview.py` file where Python can find it. A further consideration is where the file will be placed if the full path is not enumerated, the Desktop is the default destination for Microsoft Windows.

```python
from openscad import *
# nimport("https://raw.githubusercontent.com/WillAdams/gcodepreview/refs/heads/main/gcodepreview.py")
from gcodepreview import *

gcp = gcodepreview(False, # generategcode
                   True   # generatedxf
                   )

# [Stock] */
stockXwidth = 100
# [Stock] */
stockYheight = 50

# [Export] */
Base_filename = "gcpdxf"


# [CAM] */
large_square_tool_num = 102
# [CAM] */
small_square_tool_num = 0
# [CAM] */
large_ball_tool_num = 0
# [CAM] */
small_ball_tool_num = 0
# [CAM] */
large_V_tool_num = 0
# [CAM] */
small_V_tool_num = 0
# [CAM] */
DT_tool_num = 374
# [CAM] */
KH_tool_num = 0
# [CAM] */
Roundover_tool_num = 0
# [CAM] */
MISC_tool_num = 0

# [Design] */
inset = 3
# [Design] */
radius = 6
# [Design] */
cornerstyle = "Fillet"  # "Chamfer", "Flipped Fillet"

gcp.opendxffile(Base_filename)

gcp.dxfrectangle(large_square_tool_num, 0, 0, stockXwidth, stockYheight)

gcp.setdxfcolor("Red")

gcp.dxfarc(large_square_tool_num, inset, inset, radius,  0, 90)
gcp.dxfarc(large_square_tool_num, stockXwidth - inset, inset, radius, 90, 180)
gcp.dxfarc(large_square_tool_num, stockXwidth - inset, stockYheight - inset, radius, 180, 270)
gcp.dxfarc(large_square_tool_num, inset, stockYheight - inset, radius, 270, 360)

gcp.dxfline(large_square_tool_num, inset, inset + radius, inset, stockYheight - (inset + radius))
gcp.dxfline(large_square_tool_num, inset + radius, inset, stockXwidth - (inset + radius), inset)
gcp.dxfline(large_square_tool_num, stockXwidth - inset, inset + radius, stockXwidth - inset, stockYheight - (inset + radius))
gcp.dxfline(large_square_tool_num, inset + radius, stockYheight-inset, stockXwidth - (inset + radius), stockYheight - inset)

gcp.setdxfcolor("Blue")

gcp.dxfrectangle(large_square_tool_num, radius +inset, radius, stockXwidth/2 - (radius * 4), stockYheight - (radius * 2), cornerstyle, radius)
gcp.dxfrectangle(large_square_tool_num, stockXwidth/2 + (radius * 2) + inset, radius, stockXwidth/2 - (radius * 4), stockYheight - (radius * 2), cornerstyle, radius)
```

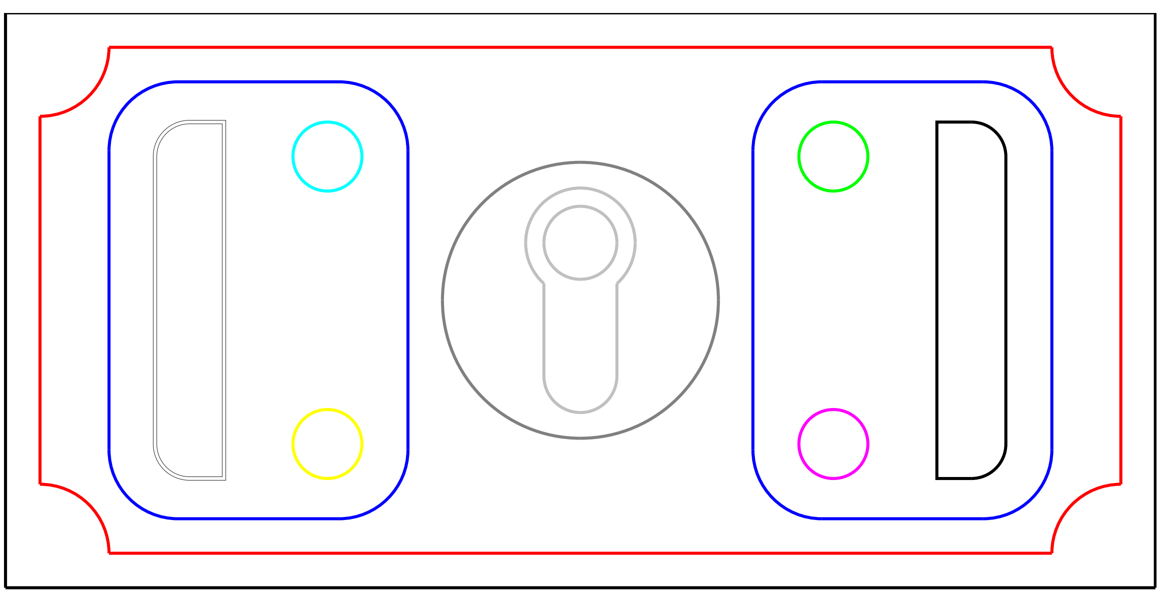
```
gcpdxfpy
gcpdxfpy gcp.setdxfcolor("Black")
gcpdxfpy
gcpdxfpy gcp.beginpolyline(large_square_tool_num)
gcpdxfpy gcp.addvertex(large_square_tool_num, stockXwidth*0.75+radius*1.5,
             stockYheight/4-radius/2)
gcpdxfpy gcp.addvertex(large_square_tool_num, stockXwidth*0.75+radius,
             stockYheight/4-radius/2)
gcpdxfpy gcp.addvertex(large_square_tool_num, stockXwidth*0.75+radius,
             stockYheight*0.75+radius/2)
gcpdxfpy gcp.addvertex(large_square_tool_num, stockXwidth*0.75+radius*1.5,
             stockYheight*0.75+radius/2)
gcpdxfpy gcp.closepolyline(large_square_tool_num)
gcpdxfpy
gcpdxfpy gcp.dxfarc(large_square_tool_num, stockXwidth*0.75+radius*1.5,
             stockYheight*0.75, radius/2,  0, 90)
gcpdxfpy
gcpdxfpy gcp.beginpolyline(large_square_tool_num)
gcpdxfpy gcp.addvertex(large_square_tool_num, stockXwidth*0.75+radius*2,
             stockYheight*0.75)
gcpdxfpy gcp.addvertex(large_square_tool_num, stockXwidth*0.75+radius*2,
             stockYheight/4)
gcpdxfpy gcp.closepolyline(large_square_tool_num)
gcpdxfpy
gcpdxfpy gcp.dxfarc(large_square_tool_num, stockXwidth*0.75+radius*1.5,
             stockYheight/4, radius/2,  270, 360)
gcpdxfpy
gcpdxfpy gcp.setdxfcolor("White")
gcpdxfpy
gcpdxfpy gcp.beginpolyline(large_square_tool_num)
gcpdxfpy gcp.addvertex(large_square_tool_num, stockXwidth*0.25-radius*1.5,
             stockYheight/4-radius/2)
gcpdxfpy gcp.addvertex(large_square_tool_num, stockXwidth*0.25-radius,
             stockYheight/4-radius/2)
gcpdxfpy gcp.addvertex(large_square_tool_num, stockXwidth*0.25-radius,
             stockYheight*0.75+radius/2)
gcpdxfpy gcp.addvertex(large_square_tool_num, stockXwidth*0.25-radius*1.5,
             stockYheight*0.75+radius/2)
gcpdxfpy gcp.closepolyline(large_square_tool_num)
gcpdxfpy
gcpdxfpy gcp.dxfarc(large_square_tool_num, stockXwidth*0.25-radius*1.5,
             stockYheight*0.75, radius/2,  90, 180)
gcpdxfpy
gcpdxfpy gcp.beginpolyline(large_square_tool_num)
gcpdxfpy gcp.addvertex(large_square_tool_num, stockXwidth*0.25-radius*2,
             stockYheight*0.75)
gcpdxfpy gcp.addvertex(large_square_tool_num, stockXwidth*0.25-radius*2,
             stockYheight/4)
gcpdxfpy gcp.closepolyline(large_square_tool_num)
gcpdxfpy
gcpdxfpy gcp.dxfarc(large_square_tool_num, stockXwidth*0.25-radius*1.5,
             stockYheight/4, radius/2,  180, 270)
gcpdxfpy
gcpdxfpy gcp.setdxfcolor("Yellow")
gcpdxfpy gcp.dxfcircle(large_square_tool_num, stockXwidth/4+1+radius/2,
             stockYheight/4, radius/2)
gcpdxfpy
gcpdxfpy gcp.setdxfcolor("Green")
gcpdxfpy gcp.dxfcircle(large_square_tool_num, stockXwidth*0.75-(1+radius/2),
              stockYheight*0.75, radius/2)
gcpdxfpy
gcpdxfpy gcp.setdxfcolor("Cyan")
gcpdxfpy gcp.dxfcircle(large_square_tool_num, stockXwidth/4+1+radius/2,
             stockYheight*0.75, radius/2)
gcpdxfpy
gcpdxfpy gcp.setdxfcolor("Magenta")
gcpdxfpy gcp.dxfcircle(large_square_tool_num, stockXwidth*0.75-(1+radius/2),
              stockYheight/4, radius/2)
gcpdxfpy
gcpdxfpy gcp.setdxfcolor("Dark␣Gray")
gcpdxfpy
gcpdxfpy gcp.dxfcircle(large_square_tool_num, stockXwidth/2, stockYheight/2,
              radius * 2)
gcpdxfpy
gcpdxfpy gcp.setdxfcolor("Light␣Gray")
gcpdxfpy
gcpdxfpy gcp.dxfKH(374, stockXwidth/2, stockYheight/5*3, 0, -7, 270,
             11.5875)
```

```
gcp.closedxffile()
```

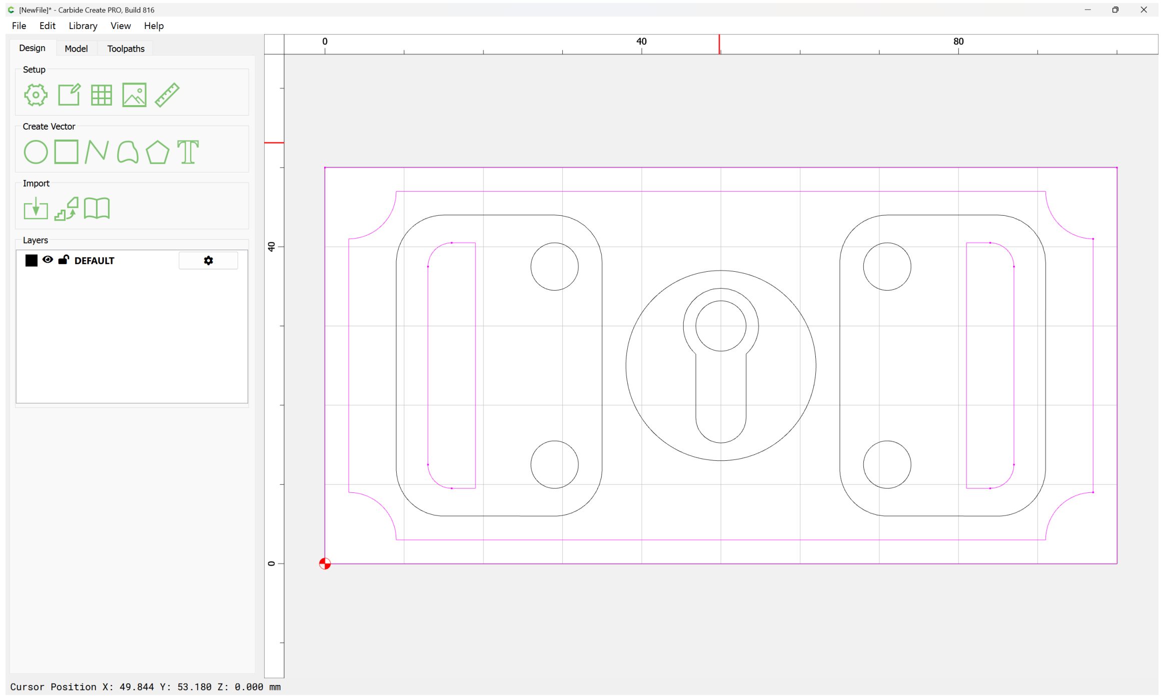
which creates a `.dxf` file which may be imported into any CAD program:



with the appearance (once converted into a `.svg` and then re-saved as a `.pdf` and edited so as to show the white elements):



and which may be imported into pretty much any CAD or CAM application, e.g., Carbide Create:



As shown/implied by the above code, the following commands/shapes are implemented:

- `dxfrectangle` (specify lower-left and upper-right corners)
  - `dxfrectangleround` (specified as "Fillet" and radius for the round option)
  - `dxfrectanglechamfer` (specified as "Chamfer" and radius for the round option)
  - `dxfrectangleflippedfillet` (specified as "Flipped Fillet" and radius for the option)
- `dxfcircle` (specifying their center and radius)
- `dxfline` (specifying begin/end points)
- `dxfarc` (specifying arc center, radius, and beginning/ending angles)
- `dxfKH` (specifying origin, depth, angle, distance)

2.2 gcpcutdxf.py

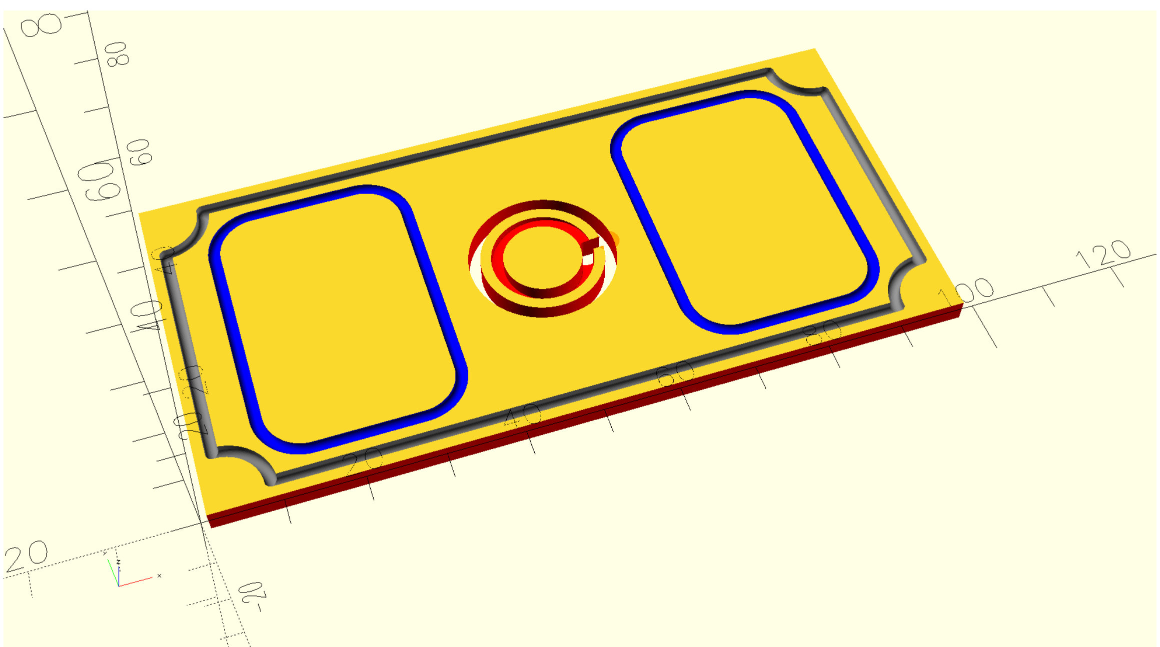
A notable limitation of the above is that there is no interactivity — the `.dxf` file is generated, then must be opened and the result of the run checked (if there is a DXF viewer/editor which will live-reload the file based on it being updated that would be obviated). Reworking the commands for a simplified version of the above design so as to show a 3D model is a straight-forward task:

```
gcpcutdxfpy from openscad import *
gcpcutdxfpy # nimport("https://raw.githubusercontent.com/WillAdams/gcodepreview
                /refs/heads/main/gcodepreview.py")
gcpcutdxfpy from gcodepreview import *
gcpcutdxfpy
gcpcutdxfpy fa = 2
gcpcutdxfpy fs = 0.125
gcpcutdxfpy
gcpcutdxfpy gcp = gcodepreview(False, # generategcode
gcpcutdxfpy                    True   # generatedxf
gcpcutdxfpy                    )
gcpcutdxfpy
gcpcutdxfpy # [Stock] */
gcpcutdxfpy stockXwidth = 100
gcpcutdxfpy # [Stock] */
gcpcutdxfpy stockYheight = 50
gcpcutdxfpy # [Stock] */
gcpcutdxfpy stockZthickness = 3.175
gcpcutdxfpy # [Stock] */
gcpcutdxfpy zeroheight = "Top"  # [Top, Bottom]
gcpcutdxfpy # [Stock] */
gcpcutdxfpy stockzero = "Lower-Left"  # [Lower-Left, Center-Left, Top-Left,
                Center]
gcpcutdxfpy # [Stock] */
gcpcutdxfpy retractheight = 3.175
gcpcutdxfpy
gcpcutdxfpy # [Export] */
gcpcutdxfpy Base_filename = "gcpdxf"
gcpcutdxfpy
```

```
gcpcutdxfpy
gcpcutdxfpy # [CAM] */
gcpcutdxfpy large_square_tool_num = 112
gcpcutdxfpy # [CAM] */
gcpcutdxfpy small_square_tool_num = 0
gcpcutdxfpy # [CAM] */
gcpcutdxfpy large_ball_tool_num = 111
gcpcutdxfpy # [CAM] */
gcpcutdxfpy small_ball_tool_num = 0
gcpcutdxfpy # [CAM] */
gcpcutdxfpy large_V_tool_num = 0
gcpcutdxfpy # [CAM] */
gcpcutdxfpy small_V_tool_num = 0
gcpcutdxfpy # [CAM] */
gcpcutdxfpy DT_tool_num = 374
gcpcutdxfpy # [CAM] */
gcpcutdxfpy KH_tool_num = 0
gcpcutdxfpy # [CAM] */
gcpcutdxfpy Roundover_tool_num = 0
gcpcutdxfpy # [CAM] */
gcpcutdxfpy MISC_tool_num = 0
gcpcutdxfpy
gcpcutdxfpy # [Design] */
gcpcutdxfpy inset = 3
gcpcutdxfpy # [Design] */
gcpcutdxfpy radius = 6
gcpcutdxfpy # [Design] */
gcpcutdxfpy cornerstyle = "Fillet"  # "Chamfer", "Flipped Fillet"
gcpcutdxfpy
gcpcutdxfpy gcp.opendxffile(Base_filename)
gcpcutdxfpy
gcpcutdxfpy gcp.setupstock(stockXwidth, stockYheight, stockZthickness, zeroheight, stockzero, retractheight)
gcpcutdxfpy
gcpcutdxfpy gcp.toolchange(large_square_tool_num)
gcpcutdxfpy
gcpcutdxfpy gcp.setdxfcolor("Red")
gcpcutdxfpy
gcpcutdxfpy gcp.cutrectangledxf(large_square_tool_num, 0, 0, 0, stockXwidth, stockYheight, stockZthickness)
gcpcutdxfpy
gcpcutdxfpy gcp.toolchange(large_ball_tool_num)
gcpcutdxfpy
gcpcutdxfpy gcp.setdxfcolor("Gray")
gcpcutdxfpy
gcpcutdxfpy gcp.rapid(inset + radius, inset, 0, "laser")
gcpcutdxfpy
gcpcutdxfpy gcp.cutlinedxf(inset + radius, inset, -stockZthickness/2)
gcpcutdxfpy gcp.cutquarterCCNEdxf(inset, inset + radius, -stockZthickness/2, radius)
gcpcutdxfpy
gcpcutdxfpy gcp.cutlinedxf(inset, stockYheight - (inset + radius), -stockZthickness/2)
gcpcutdxfpy
gcpcutdxfpy gcp.cutquarterCCSEdxf(inset + radius, stockYheight - inset, -stockZthickness/2, radius)
gcpcutdxfpy
gcpcutdxfpy gcp.cutlinedxf(stockXwidth - (inset + radius), stockYheight - inset, -stockZthickness/2)
gcpcutdxfpy
gcpcutdxfpy gcp.cutquarterCCSWdxf(stockXwidth - inset, stockYheight - (inset + radius), -stockZthickness/2, radius)
gcpcutdxfpy
gcpcutdxfpy gcp.cutlinedxf(stockXwidth - (inset), (inset + radius), -stockZthickness/2)
gcpcutdxfpy
gcpcutdxfpy gcp.cutquarterCCNWdxf(stockXwidth - (inset + radius), inset, -stockZthickness/2, radius)
gcpcutdxfpy
gcpcutdxfpy gcp.cutlinedxf((inset + radius), inset, -stockZthickness/2)
gcpcutdxfpy
gcpcutdxfpy gcp.setdxfcolor("Blue")
gcpcutdxfpy
gcpcutdxfpy gcp.rapid(radius + inset + radius, radius, 0, "laser")
gcpcutdxfpy
gcpcutdxfpy gcp.cutrectangleroundxf(large_square_tool_num, radius +inset, radius, 0, stockXwidth/2 - (radius * 4), stockYheight - (radius * 2), -stockZthickness/4, radius)
```

```
gcpcutdxfpy
gcpcutdxfpy gcp.rapid(stockXwidth/2 + (radius * 2) + inset + radius, radius, 0,
             "laser")
gcpcutdxfpy
gcpcutdxfpy gcp.cutrectanglerounddxf(large_square_tool_num, stockXwidth/2 + (
            radius * 2) + inset, radius, 0, stockXwidth/2 - (radius * 4),
            stockYheight - (radius * 2), -stockZthickness/4, radius)
gcpcutdxfpy
gcpcutdxfpy gcp.setdxfcolor("Red")
gcpcutdxfpy
gcpcutdxfpy gcp.rapid(stockXwidth/2 + radius, stockYheight/2, 0, "laser")
gcpcutdxfpy
gcpcutdxfpy gcp.toolchange(large_square_tool_num)
gcpcutdxfpy
gcpcutdxfpy gcp.cutcircleCC(stockXwidth/2, stockYheight/2, 0, -stockZthickness,
             radius)
gcpcutdxfpy
gcpcutdxfpy gcp.cutcircleCC(stockXwidth/2, stockYheight/2, -stockZthickness, -
            stockZthickness, radius*1.5)
gcpcutdxfpy
gcpcutdxfpy gcp.closedxffile()
gcpcutdxfpy
gcpcutdxfpy gcp.stockandtoolpaths()
```

which creates the design:



and which allows an interactive usage in working up a design such as for lasercutting, and which incorporates and option to the `rapid(x,y,z)` command which simulates turning a laser off, repositioning, then powering up the laser after.

2.3 gcodepreviewtemplate.py

Note that since the v0.7 re-write, it is possible to directly use the underlying Python code. Using Python to generate 3D previews of how DXFs or G-code will cut requires the use of PythonSCAD.

```
gcptmplpy #!/usr/bin/env python
gcptmplpy
gcptmplpy import sys
gcptmplpy
gcptmplpy try:
gcptmplpy     if 'gcodepreview' in sys.modules:
gcptmplpy         del sys.modules['gcodepreview']
gcptmplpy except AttributeError:
gcptmplpy     pass
gcptmplpy
gcptmplpy from gcodepreview import *
gcptmplpy
gcptmplpy fa = 2
gcptmplpy fs = 0.125
gcptmplpy
gcptmplpy # [Export] */
gcptmplpy Base_filename = "aexport"
gcptmplpy # [Export] */
gcptmplpy generatedxf = True
```

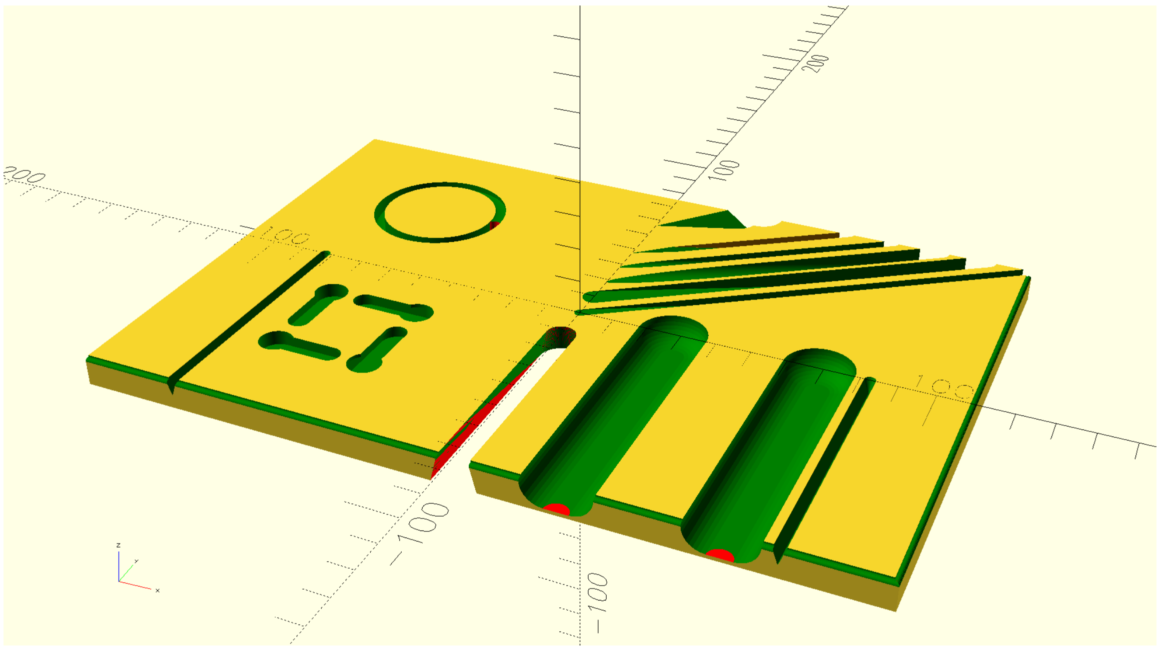
```
gcptmplpy # [Export] */
gcptmplpy generategcode = True
gcptmplpy
gcptmplpy # [Stock] */
gcptmplpy stockXwidth = 220
gcptmplpy # [Stock] */
gcptmplpy stockYheight = 150
gcptmplpy # [Stock] */
gcptmplpy stockZthickness = 8.35
gcptmplpy # [Stock] */
gcptmplpy zeroheight = "Top"  # [Top, Bottom]
gcptmplpy # [Stock] */
gcptmplpy stockzero = "Center"  # [Lower-Left, Center-Left, Top-Left, Center]
gcptmplpy # [Stock] */
gcptmplpy retractheight = 9
gcptmplpy
gcptmplpy # [CAM] */
gcptmplpy toolradius = 1.5875
gcptmplpy # [CAM] */
gcptmplpy large_square_tool_num = 201  # [0:0, 112:112, 102:102, 201:201]
gcptmplpy # [CAM] */
gcptmplpy small_square_tool_num = 102  # [0:0, 122:122, 112:112, 102:102]
gcptmplpy # [CAM] */
gcptmplpy large_ball_tool_num = 202  # [0:0, 111:111, 101:101, 202:202]
gcptmplpy # [CAM] */
gcptmplpy small_ball_tool_num = 101  # [0:0, 121:121, 111:111, 101:101]
gcptmplpy # [CAM] */
gcptmplpy large_V_tool_num = 301  # [0:0, 301:301, 690:690]
gcptmplpy # [CAM] */
gcptmplpy small_V_tool_num = 390  # [0:0, 390:390, 301:301]
gcptmplpy # [CAM] */
gcptmplpy DT_tool_num = 814  # [0:0, 814:814, 808079:808079]
gcptmplpy # [CAM] */
gcptmplpy KH_tool_num = 374  # [0:0, 374:374, 375:375, 376:376, 378:378]
gcptmplpy # [CAM] */
gcptmplpy Roundover_tool_num = 56142  # [56142:56142, 56125:56125, 1570:1570]
gcptmplpy # [CAM] */
gcptmplpy MISC_tool_num = 0  # [501:501, 502:502, 45982:45982]
gcptmplpy #501 https://shop.carbide3d.com/collections/cutters/products/501-
              engraving-bit
gcptmplpy #502 https://shop.carbide3d.com/collections/cutters/products/502-
              engraving-bit
gcptmplpy #204 tapered ball nose 0.0625", 0.2500", 1.50", 3.6ř
gcptmplpy #304 tapered ball nose 0.1250", 0.2500", 1.50", 2.4ř
gcptmplpy #648 threadmill_shaft(2.4, 0.75, 18)
gcptmplpy #45982 Carbide Tipped Bowl & Tray 1/4 Radius x 3/4 Dia x 5/8 x 1/4
              Inch Shank
gcptmplpy #13921 https://www.amazon.com/Yonico-Groove-Bottom-Router-Degree/dp
              /B0CPJPTMPP
gcptmplpy
gcptmplpy # [Feeds and Speeds] */
gcptmplpy plunge = 100
gcptmplpy # [Feeds and Speeds] */
gcptmplpy feed = 400
gcptmplpy # [Feeds and Speeds] */
gcptmplpy speed = 16000
gcptmplpy # [Feeds and Speeds] */
gcptmplpy small_square_ratio = 0.75  # [0.25:2]
gcptmplpy # [Feeds and Speeds] */
gcptmplpy large_ball_ratio = 1.0  # [0.25:2]
gcptmplpy # [Feeds and Speeds] */
gcptmplpy small_ball_ratio = 0.75  # [0.25:2]
gcptmplpy # [Feeds and Speeds] */
gcptmplpy large_V_ratio = 0.875  # [0.25:2]
gcptmplpy # [Feeds and Speeds] */
gcptmplpy small_V_ratio = 0.625  # [0.25:2]
gcptmplpy # [Feeds and Speeds] */
gcptmplpy DT_ratio = 0.75  # [0.25:2]
gcptmplpy # [Feeds and Speeds] */
gcptmplpy KH_ratio = 0.75  # [0.25:2]
gcptmplpy # [Feeds and Speeds] */
gcptmplpy RO_ratio = 0.5  # [0.25:2]
gcptmplpy # [Feeds and Speeds] */
gcptmplpy MISC_ratio = 0.5  # [0.25:2]
gcptmplpy
gcptmplpy gcp = gcodepreview(generategcode,
gcptmplpy                    generatedxf,
gcptmplpy                    )
```

```

gcp.opengcodefile(Base_filename)
gcp.opendxffile(Base_filename)
gcp.opendxffiles(Base_filename,
                 large_square_tool_num,
                 small_square_tool_num,
                 large_ball_tool_num,
                 small_ball_tool_num,
                 large_V_tool_num,
                 small_V_tool_num,
                 DT_tool_num,
                 KH_tool_num,
                 Roundover_tool_num,
                 MISC_tool_num)
gcp.setupstock(stockXwidth, stockYheight, stockZthickness, zeroheight, stockzero, retractheight)

gcp.movetosafeZ()

gcp.toolchange(102, 10000)

gcp.rapidZ(0)

gcp.cutlinedxfgc(stockXwidth/2, stockYheight/2, -stockZthickness)

gcp.rapidZ(retractheight)
gcp.toolchange(201, 10000)
gcp.rapidXY(0, stockYheight/16)
gcp.rapidZ(0)
gcp.cutlinedxfgc(stockXwidth/16*7, stockYheight/2, -stockZthickness)

gcp.rapidZ(retractheight)
gcp.toolchange(202, 10000)
gcp.rapidXY(0, stockYheight/8)
gcp.rapidZ(0)
gcp.cutlinedxfgc(stockXwidth/16*6, stockYheight/2, -stockZthickness)

gcp.rapidZ(retractheight)
gcp.toolchange(101, 10000)
gcp.rapidXY(0, stockYheight/16*3)
gcp.rapidZ(0)
gcp.cutlinedxfgc(stockXwidth/16*5, stockYheight/2, -stockZthickness)

gcp.setzpos(retractheight)
gcp.toolchange(390, 10000)
gcp.rapidXY(0, stockYheight/16*4)
gcp.rapidZ(0)
gcp.cutlinedxfgc(stockXwidth/16*4, stockYheight/2, -stockZthickness)
gcp.rapidZ(retractheight)

gcp.toolchange(301, 10000)
gcp.rapidXY(0, stockYheight/16*6)
gcp.rapidZ(0)
gcp.cutlinedxfgc(stockXwidth/16*2, stockYheight/2, -stockZthickness)

rapids = gcp.rapid(gcp.xpos(), gcp.ypos(), retractheight)
gcp.toolchange(102, 10000)

gcp.rapid(-stockXwidth/4+stockYheight/16, +stockYheight/4, 0)

#gcp.cutarcCC(0, 90, gcp.xpos()-stockYheight/16, gcp.ypos(), stockYheight/16, -stockZthickness/4)
#gcp.cutarcCC(90, 180, gcp.xpos(), gcp.ypos()-stockYheight/16, stockYheight/16, -stockZthickness/4)
#gcp.cutarcCC(180, 270, gcp.xpos()+stockYheight/16, gcp.ypos(), stockYheight/16, -stockZthickness/4)
#gcp.cutarcCC(270, 360, gcp.xpos(), gcp.ypos()+stockYheight/16, stockYheight/16, -stockZthickness/4)
gcp.cutquarterCCNEdxf(gcp.xpos() - stockYheight/8, gcp.ypos() + stockYheight/8, -stockZthickness/4, stockYheight/8)
gcp.cutquarterCCNWdxf(gcp.xpos() - stockYheight/8, gcp.ypos() - stockYheight/8, -stockZthickness/2, stockYheight/8)
gcp.cutquarterCCSWdxf(gcp.xpos() + stockYheight/8, gcp.ypos() -
```

```
stockYheight/8, -stockZthickness * 0.75, stockYheight/8)
gcp.cutquarterCCSEdxf(gcp.xpos() + stockYheight/8, gcp.ypos() + stockYheight/8, -stockZthickness, stockYheight/8)

gcp.movetosafeZ()
gcp.rapidXY(stockXwidth/4-stockYheight/16, -stockYheight/4)
gcp.rapidZ(0)


#gcp.cutarcCW(180, 90, gcp.xpos()+stockYheight/16, gcp.ypos(), stockYheight/16, -stockZthickness/4)
#gcp.cutarcCW(90, 0, gcp.xpos(), gcp.ypos()-stockYheight/16, stockYheight/16, -stockZthickness/4)
#gcp.cutarcCW(360, 270, gcp.xpos()-stockYheight/16, gcp.ypos(), stockYheight/16, -stockZthickness/4)
#gcp.cutarcCW(270, 180, gcp.xpos(), gcp.ypos()+stockYheight/16, stockYheight/16, -stockZthickness/4)

#gcp.movetosafeZ()
#gcp.toolchange(201, 10000)
#gcp.rapidXY(stockXwidth/2, -stockYheight/2)
#gcp.rapidZ(0)

#gcp.cutlinedxfgc(gcp.xpos(), gcp.ypos(), -stockZthickness)
#test = gcp.cutlinedxfgc(gcp.xpos(), gcp.ypos(), -stockZthickness)

#gcp.movetosafeZ()
#gcp.rapidXY(stockXwidth/2-6.34, -stockYheight/2)
#gcp.rapidZ(0)

#gcp.cutarcCW(180, 90, stockXwidth/2, -stockYheight/2, 6.34, -stockZthickness)


gcp.movetosafeZ()
gcp.toolchange(814, 10000)
gcp.rapidXY(0, -(stockYheight/2+12.7))
gcp.rapidZ(0)

gcp.cutlinedxfgc(gcp.xpos(), gcp.ypos(), -stockZthickness)
gcp.cutlinedxfgc(gcp.xpos(), -12.7, -stockZthickness)

gcp.rapidXY(0, -(stockYheight/2+12.7))
gcp.movetosafeZ()
gcp.toolchange(374, 10000)
gcp.rapidXY(stockXwidth/4-stockXwidth/16, -(stockYheight/4+stockYheight/16))
gcp.rapidZ(0)

gcp.rapidZ(retractheight)
gcp.toolchange(374, 10000)
gcp.rapidXY(-stockXwidth/4-stockXwidth/16, -(stockYheight/4+stockYheight/16))
gcp.rapidZ(0)

gcp.cutline(gcp.xpos(), gcp.ypos(), -stockZthickness/2)
gcp.cutlinedxfgc(gcp.xpos()+stockYheight/9, gcp.ypos(), gcp.zpos())

gcp.cutline(gcp.xpos()-stockYheight/9, gcp.ypos(), gcp.zpos())
gcp.cutline(gcp.xpos(), gcp.ypos(), 0)

#key = gcp.cutkeyholegcdxf(KH_tool_num, 0, stockZthickness*0.75, "E", stockYheight/9)
#key = gcp.cutKHgcdxf(374, 0, stockZthickness*0.75, 90, stockYheight/9)
#toolpaths = toolpaths.union(key)

gcp.rapidZ(retractheight)
gcp.rapidXY(-stockXwidth/4+stockXwidth/16, -(stockYheight/4+stockYheight/16))
gcp.rapidZ(0)
gcp.cutline(gcp.xpos(), gcp.ypos(), -stockZthickness/2)
gcp.cutlinedxfgc(gcp.xpos(), gcp.ypos()+stockYheight/9, gcp.zpos())

gcp.cutline(gcp.xpos(), gcp.ypos()-stockYheight/9, gcp.zpos())
gcp.cutline(gcp.xpos(), gcp.ypos(), 0)

gcp.rapidZ(retractheight)
```

```
gcp.rapidXY(-stockXwidth/4+stockXwidth/16, -(stockYheight/4-stockYheight/8))
gcp.rapidZ(0)

gcp.cutline(gcp.xpos(), gcp.ypos(), -stockZthickness/2)
gcp.cutlinedxfgc(gcp.xpos()-stockYheight/9, gcp.ypos(), gcp.zpos())

gcp.cutline(gcp.xpos()+stockYheight/9, gcp.ypos(), gcp.zpos())
gcp.cutline(gcp.xpos(), gcp.ypos(), 0)

gcp.rapidZ(retractheight)
gcp.rapidXY(-stockXwidth/4-stockXwidth/16, -(stockYheight/4-stockYheight/8))
gcp.rapidZ(0)
gcp.cutline(gcp.xpos(), gcp.ypos(), -stockZthickness/2)
gcp.cutlinedxfgc(gcp.xpos(), gcp.ypos()-stockYheight/9, gcp.zpos())
gcp.cutline(gcp.xpos(), gcp.ypos()+stockYheight/9, gcp.zpos())
gcp.cutline(gcp.xpos(), gcp.ypos(), 0)

gcp.rapidZ(retractheight)
gcp.toolchange(56142, 10000)
gcp.rapidXY(-stockXwidth/2, -(stockYheight/2+0.508/2))
gcp.cutline(gcp.xpos(), gcp.ypos(), -1.531)
gcp.cutlinedxfgc(stockXwidth/2+0.508/2, -(stockYheight/2+0.508/2), -1.531)

gcp.rapidZ(retractheight)

gcp.cutline(gcp.xpos(), gcp.ypos(), -1.531)
gcp.cutlinedxfgc(stockXwidth/2+0.508/2, (stockYheight/2+0.508/2), -1.531)

gcp.rapidZ(retractheight)
gcp.toolchange(45982, 10000)
gcp.rapidXY(stockXwidth/8, 0)
gcp.cutline(gcp.xpos(), gcp.ypos(), -(stockZthickness*7/8))
gcp.cutlinedxfgc(gcp.xpos(), -stockYheight/2, -(stockZthickness*7/8))

gcp.rapidZ(retractheight)
gcp.toolchange(204, 10000)
gcp.rapidXY(stockXwidth*0.3125, 0)
gcp.cutline(gcp.xpos(), gcp.ypos(), -(stockZthickness*7/8))
gcp.cutlinedxfgc(gcp.xpos(), -stockYheight/2, -(stockZthickness*7/8))

gcp.rapidZ(retractheight)
gcp.toolchange(502, 10000)
gcp.rapidXY(stockXwidth*0.375, 0)
gcp.cutline(gcp.xpos(), gcp.ypos(), -4.24)
gcp.cutlinedxfgc(gcp.xpos(), -stockYheight/2, -4.24)

gcp.rapidZ(retractheight)
gcp.toolchange(13921, 10000)
gcp.rapidXY(-stockXwidth*0.375, 0)
gcp.cutline(gcp.xpos(), gcp.ypos(), -stockZthickness/2)
gcp.cutlinedxfgc(gcp.xpos(), -stockYheight/2, -stockZthickness/2)

gcp.rapidZ(retractheight)

gcp.stockandtoolpaths()

gcp.closegcodefile()
gcp.closedxffiles()
gcp.closedxffile()
```

Which generates a 3D model which previews in PythonSCAD as:



2.4 gcodepreviewtemplate.scad

Since the project began in OpenSCAD, having an implementation in that language has always been a goal. This is quite straight-forward since the Python code when imported into OpenSCAD may be accessed by quite simple modules which are for the most part, a series of decorators/descriptors which wrap up the Python definitions as OpenSCAD modules. Moreover, such an implementation will facilitate usage by tools intended for this application such as OpenSCAD Graph Editor: https://github.com/derkork/openscad-graph-editor.

```
gcptmpl //!OpenSCAD
gcptmpl
gcptmpl use <gcodepreview.py>
gcptmpl include <gcodepreview.scad>
gcptmpl
gcptmpl $fa = 2;
gcptmpl $fs = 0.125;
gcptmpl fa = 2;
gcptmpl fs = 0.125;
gcptmpl
gcptmpl /* [Stock] */
gcptmpl stockXwidth = 220;
gcptmpl /* [Stock] */
gcptmpl stockYheight = 150;
gcptmpl /* [Stock] */
gcptmpl stockZthickness = 8.35;
gcptmpl /* [Stock] */
gcptmpl zeroheight = "Top"; // [Top, Bottom]
gcptmpl /* [Stock] */
gcptmpl stockzero = "Center"; // [Lower-Left, Center-Left, Top-Left, Center
            ]
gcptmpl /* [Stock] */
gcptmpl retractheight = 9;
gcptmpl
gcptmpl /* [Export] */
gcptmpl Base_filename = "export";
gcptmpl /* [Export] */
gcptmpl generatedxf = true;
gcptmpl /* [Export] */
gcptmpl generategcode = true;
gcptmpl
gcptmpl /* [CAM] */
gcptmpl toolradius = 1.5875;
gcptmpl /* [CAM] */
gcptmpl large_square_tool_num = 0; // [0:0, 112:112, 102:102, 201:201]
gcptmpl /* [CAM] */
gcptmpl small_square_tool_num = 102; // [0:0, 122:122, 112:112, 102:102]
gcptmpl /* [CAM] */
gcptmpl large_ball_tool_num = 0; // [0:0, 111:111, 101:101, 202:202]
gcptmpl /* [CAM] */
gcptmpl small_ball_tool_num = 0; // [0:0, 121:121, 111:111, 101:101]
gcptmpl /* [CAM] */
gcptmpl large_V_tool_num = 0; // [0:0, 301:301, 690:690]
gcptmpl /* [CAM] */
```

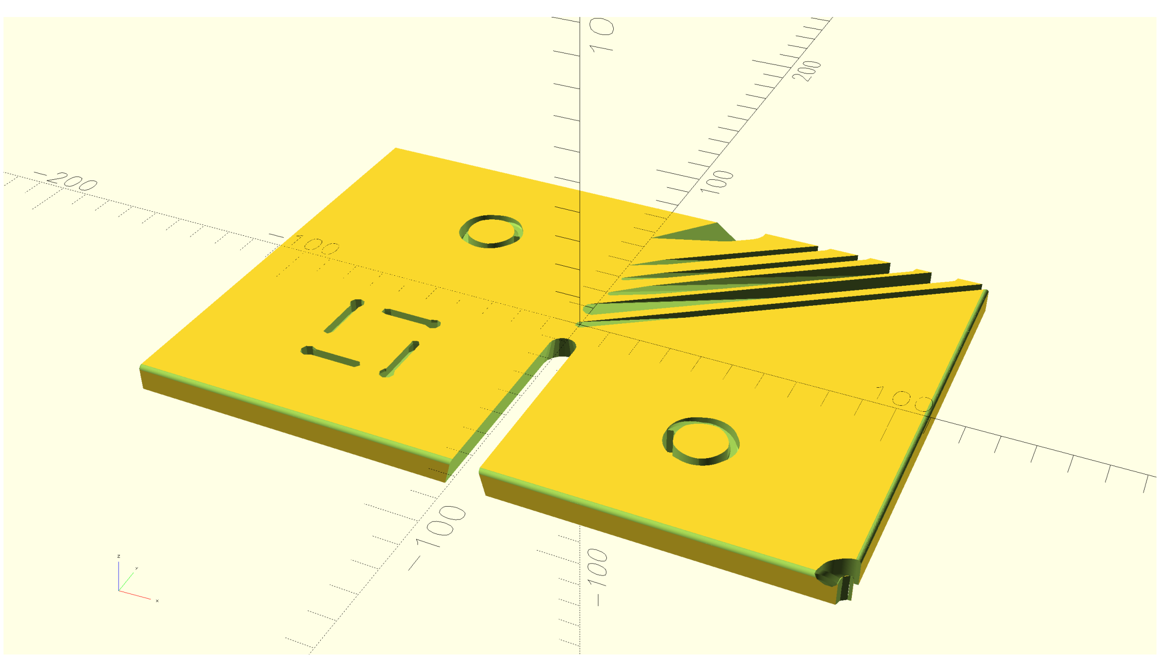
```
small_V_tool_num = 0; // [0:0, 390:390, 301:301]
/* [CAM] */
DT_tool_num = 0; // [0:0, 814:814, 808079:808079]
/* [CAM] */
KH_tool_num = 0; // [0:0, 374:374, 375:375, 376:376, 378:378]
/* [CAM] */
Roundover_tool_num = 0; // [56142:56142, 56125:56125, 1570:1570]
/* [CAM] */
MISC_tool_num = 0; // [648:648, 45982:45982]
//648 threadmill_shaft(2.4, 0.75, 18)
//45982 Carbide Tipped Bowl & Tray 1/4 Radius x 3/4 Dia x 5/8 x 1/4 Inch Shank

/* [Feeds and Speeds] */
plunge = 100;
/* [Feeds and Speeds] */
feed = 400;
/* [Feeds and Speeds] */
speed = 16000;
/* [Feeds and Speeds] */
small_square_ratio = 0.75; // [0.25:2]
/* [Feeds and Speeds] */
large_ball_ratio = 1.0; // [0.25:2]
/* [Feeds and Speeds] */
small_ball_ratio = 0.75; // [0.25:2]
/* [Feeds and Speeds] */
large_V_ratio = 0.875; // [0.25:2]
/* [Feeds and Speeds] */
small_V_ratio = 0.625; // [0.25:2]
/* [Feeds and Speeds] */
DT_ratio = 0.75; // [0.25:2]
/* [Feeds and Speeds] */
KH_ratio = 0.75; // [0.25:2]
/* [Feeds and Speeds] */
RO_ratio = 0.5; // [0.25:2]
/* [Feeds and Speeds] */
MISC_ratio = 0.5; // [0.25:2]

thegeneratedxf = generatedxf == true ? 1 : 0;
thegenerategcode = generategcode == true ? 1 : 0;

gcp = gcodepreview(thegenerategcode,
                    thegeneratedxf,
                    );

opengcodefile(Base_filename);
opendxffile(Base_filename);
opendxffiles(Base_filename,
                  large_square_tool_num,
                  small_square_tool_num,
                  large_ball_tool_num,
                  small_ball_tool_num,
                  large_V_tool_num,
                  small_V_tool_num,
                  DT_tool_num,
                  KH_tool_num,
                  Roundover_tool_num,
                  MISC_tool_num);

setupstock(stockXwidth, stockYheight, stockZthickness, zeroheight, stockzero);

//echo(gcp);
//gcpversion();

//c = myfunc(4);
//echo(c);

//echo(getvv());

cutline(stockXwidth/2, stockYheight/2, -stockZthickness);

rapidZ(retractheight);
toolchange(201, 10000);
rapidXY(0, stockYheight/16);
rapidZ(0);
cutlinedxfgc(stockXwidth/16*7, stockYheight/2, -stockZthickness);

```

```

rapidZ(retractheight);
toolchange(202, 10000);
rapidXY(0, stockYheight/8);
rapidZ(0);
cutlinedxfgc(stockXwidth/16*6, stockYheight/2, -stockZthickness);

rapidZ(retractheight);
toolchange(101, 10000);
rapidXY(0, stockYheight/16*3);
rapidZ(0);
cutlinedxfgc(stockXwidth/16*5, stockYheight/2, -stockZthickness);

rapidZ(retractheight);
toolchange(390, 10000);
rapidXY(0, stockYheight/16*4);
rapidZ(0);

cutlinedxfgc(stockXwidth/16*4, stockYheight/2, -stockZthickness);
rapidZ(retractheight);

toolchange(301, 10000);
rapidXY(0, stockYheight/16*6);
rapidZ(0);

cutlinedxfgc(stockXwidth/16*2, stockYheight/2, -stockZthickness);


movetosafeZ();
rapid(gcp.xpos(), gcp.ypos(), retractheight);
toolchange(102, 10000);

//rapidXY(stockXwidth/4+stockYheight/8+stockYheight/16, +stockYheight/8);
rapidXY(-stockXwidth/4+stockXwidth/16, (stockYheight/4));//+stockYheight/16
rapidZ(0);

//cutarcCW(360, 270, gcp.xpos()-stockYheight/16, gcp.ypos(), stockYheight/16, -stockZthickness);
//gcp.cutarcCW(270, 180, gcp.xpos(), gcp.ypos()+stockYheight/16, stockYheight/16))
//cutarcCC(0, 90, gcp.xpos()-stockYheight/16, gcp.ypos(), stockYheight/16, -stockZthickness/4);
//cutarcCC(90, 180, gcp.xpos(), gcp.ypos()-stockYheight/16, stockYheight/16, -stockZthickness/4);
//cutarcCC(180, 270, gcp.xpos()+stockYheight/16, gcp.ypos(), stockYheight/16, -stockZthickness/4);
//cutarcCC(270, 360, gcp.xpos(), gcp.ypos()+stockYheight/16, stockYheight/16, -stockZthickness/4);

movetosafeZ();
//rapidXY(stockXwidth/4+stockYheight/8-stockYheight/16, -stockYheight/8);
rapidXY(stockXwidth/4-stockYheight/16, -(stockYheight/4));
rapidZ(0);

//cutarcCW(180, 90, gcp.xpos()+stockYheight/16, gcp.ypos(), stockYheight/16, -stockZthickness/4);
//cutarcCW(90, 0, gcp.xpos(), gcp.ypos()-stockYheight/16, stockYheight/16, -stockZthickness/4);
//cutarcCW(360, 270, gcp.xpos()-stockYheight/16, gcp.ypos(), stockYheight/16, -stockZthickness/4);
//cutarcCW(270, 180, gcp.xpos(), gcp.ypos()+stockYheight/16, stockYheight/16, -stockZthickness/4);

movetosafeZ();

rapidXY(-stockXwidth/4 + stockYheight/8, (stockYheight/4));
rapidZ(0);

cutquarterCCNEdxf(xpos() - stockYheight/8, ypos() + stockYheight/8, -stockZthickness/4, stockYheight/8);
cutquarterCCNWdxf(xpos() - stockYheight/8, ypos() - stockYheight/8, -stockZthickness/2, stockYheight/8);
cutquarterCCSWdxf(xpos() + stockYheight/8, ypos() - stockYheight/8, -stockZthickness * 0.75, stockYheight/8);
//cutquarterCCSEdxf(xpos() + stockYheight/8, ypos() + stockYheight
```

```
        /8, -stockZthickness, stockYheight/8);
gcptmpl
gcptmpl movetosafeZ();
gcptmpl toolchange(201, 10000);
gcptmpl rapidXY(stockXwidth /2 -6.34, - stockYheight /2);
gcptmpl rapidZ(0);
gcptmpl //cutarcCW(180, 90, stockXwidth /2, -stockYheight/2, 6.34, -
        stockZthickness);
gcptmpl
gcptmpl movetosafeZ();
gcptmpl rapidXY(stockXwidth/2, -stockYheight/2);
gcptmpl rapidZ(0);
gcptmpl
gcptmpl //gcp.cutlinedxfgc(gcp.xpos(), gcp.ypos(), -stockZthickness);
gcptmpl
gcptmpl movetosafeZ();
gcptmpl toolchange(814, 10000);
gcptmpl rapidXY(0, -(stockYheight/2+12.7));
gcptmpl rapidZ(0);
gcptmpl
gcptmpl cutlinedxfgc(xpos(), ypos(), -stockZthickness);
gcptmpl cutlinedxfgc(xpos(), -12.7, -stockZthickness);
gcptmpl rapidXY(0, -(stockYheight/2+12.7));
gcptmpl
gcptmpl //rapidXY(stockXwidth/2-6.34, -stockYheight/2);
gcptmpl //rapidZ(0);
gcptmpl
gcptmpl //movetosafeZ();
gcptmpl //toolchange(374, 10000);
gcptmpl //rapidXY(-(stockXwidth/4 - stockXwidth /16), -(stockYheight/4 +
        stockYheight/16))
gcptmpl
gcptmpl //cutline(xpos(), ypos(), (stockZthickness/2) * -1);
gcptmpl //cutlinedxfgc(xpos() + stockYheight /9, ypos(), zpos());
gcptmpl //cutline(xpos() - stockYheight /9, ypos(), zpos());
gcptmpl //cutline(xpos(), ypos(), 0);
gcptmpl
gcptmpl movetosafeZ();
gcptmpl
gcptmpl toolchange(374, 10000);
gcptmpl rapidXY(-stockXwidth/4-stockXwidth/16, -(stockYheight/4+
        stockYheight/16))
gcptmpl //rapidXY(-(stockXwidth/4 - stockXwidth /16), -(stockYheight/4 +
        stockYheight/16))
gcptmpl rapidZ(0);
gcptmpl
gcptmpl cutline(xpos(), ypos(), (stockZthickness/2) * -1);
gcptmpl cutlinedxfgc(xpos() + stockYheight /9, ypos(), zpos());
gcptmpl cutline(xpos() - stockYheight /9, ypos(), zpos());
gcptmpl cutline(xpos(), ypos(), 0);
gcptmpl
gcptmpl rapidZ(retractheight);
gcptmpl rapidXY(-stockXwidth/4+stockXwidth/16, -(stockYheight/4+
        stockYheight/16));
gcptmpl rapidZ(0);
gcptmpl cutline(gcp.xpos(), gcp.ypos(), -stockZthickness/2);
gcptmpl cutlinedxfgc(gcp.xpos(), gcp.ypos()+stockYheight/9, gcp.zpos());
gcptmpl cutline(gcp.xpos(), gcp.ypos()-stockYheight/9, gcp.zpos());
gcptmpl cutline(gcp.xpos(), gcp.ypos(), 0);
gcptmpl
gcptmpl rapidZ(retractheight);
gcptmpl rapidXY(-stockXwidth/4+stockXwidth/16, -(stockYheight/4-
        stockYheight/8));
gcptmpl rapidZ(0);
gcptmpl cutline(gcp.xpos(), gcp.ypos(), -stockZthickness/2);
gcptmpl cutlinedxfgc(gcp.xpos()-stockYheight/9, gcp.ypos(), gcp.zpos());
gcptmpl cutline(gcp.xpos()+stockYheight/9, gcp.ypos(), gcp.zpos());
gcptmpl cutline(gcp.xpos(), gcp.ypos(), 0);
gcptmpl
gcptmpl rapidZ(retractheight);
gcptmpl rapidXY(-stockXwidth/4-stockXwidth/16, -(stockYheight/4-
        stockYheight/8));
gcptmpl rapidZ(0);
gcptmpl cutline(gcp.xpos(), gcp.ypos(), -stockZthickness/2);
gcptmpl cutlinedxfgc(gcp.xpos(), gcp.ypos()-stockYheight/9, gcp.zpos());
gcptmpl cutline(gcp.xpos(), gcp.ypos()+stockYheight/9, gcp.zpos());
gcptmpl cutline(gcp.xpos(), gcp.ypos(), 0);
gcptmpl
```

```
rapidZ(retractheight);
toolchange(45982, 10000);
rapidXY(stockXwidth/8, 0);
cutline(gcp.xpos(), gcp.ypos(), -(stockZthickness*7/8));
cutlinedxfgc(gcp.xpos(), -stockYheight/2, -(stockZthickness*7/8));

rapidZ(retractheight);
toolchange(204, 10000);
rapidXY(stockXwidth*0.3125, 0);
cutline(gcp.xpos(), gcp.ypos(), -(stockZthickness*7/8));
cutlinedxfgc(gcp.xpos(), -stockYheight/2, -(stockZthickness*7/8));

rapidZ(retractheight);
toolchange(502, 10000);
rapidXY(stockXwidth*0.375, 0);
cutline(gcp.xpos(), gcp.ypos(), -4.24);
cutlinedxfgc(gcp.xpos(), -stockYheight/2, -4.24);

rapidZ(retractheight);
toolchange(13921, 10000);
rapidXY(-stockXwidth*0.375, 0);
cutline(gcp.xpos(), gcp.ypos(), -stockZthickness/2);
cutlinedxfgc(gcp.xpos(), -stockYheight/2, -stockZthickness/2);

rapidZ(retractheight);
gcp.toolchange(56142, 10000);
gcp.rapidXY(-stockXwidth/2, -(stockYheight/2+0.508/2));
cutlineZgcfeed(-1.531, plunge);
//cutline(gcp.xpos(), gcp.ypos(), -1.531);
cutlinedxfgc(stockXwidth/2+0.508/2, -(stockYheight/2+0.508/2),
    -1.531);

rapidZ(retractheight);
//#gcp.toolchange(56125, 10000)
cutlineZgcfeed(-1.531, plunge);
//toolpaths.append(gcp.cutline(gcp.xpos(), gcp.ypos(), -1.531))
cutlinedxfgc(stockXwidth/2+0.508/2, (stockYheight/2+0.508/2),
    -1.531);

stockandtoolpaths();
//stockwotoolpaths();
//outputtoolpaths();

//makecube(3, 2, 1);

//instantiatecube();

closegcodefile();
closedxffiles();
closedxffile();
```

Which generates a 3D model which previews in OpenSCAD as:



3 gcodepreview

This library for PythonSCAD works by using Python code as a back-end so as to persistently store and access variables, and to write out files while both modeling the motion of a 3-axis CNC machine (note that at least a 4th additional axis may be worked up as a future option and supporting the work-around of two-sided (flip) machining by using an imported file as the Stock or preserving state and affording a second operation seems promising) and if desired, writing out DXF and/or G-code files (as opposed to the normal technique of rendering to a 3D model and writing out an STL or STEP or other model format and using a traditional CAM application). There are multiple modes for this, doing so may require at least two files:

- A Python file: gcodepreview.py (`gcpy`) — this has variables in the traditional sense which are used for tracking machine position and so forth. Note that where it is placed/loaded from will depend on whether it is imported into a Python file:
  `import gcodepreview_standalone as gcp`
  or used in an OpenSCAD file:
  `use <gcodepreview.py>`
  with an additional OpenSCAD module which allows accessing it and that there is an option for loading directly from the Github repository implemented in PythonSCAD
- An OpenSCAD file: gcodepreview.scad (`gcpscad`) — which `uses` the Python file and which is `included` allowing it to access OpenSCAD variables for branching

Note that this architecture requires that many OpenSCAD modules are essentially "Dispatchers" (another term is "Descriptors") which pass information from one aspect of the environment to another, but in some instances it will be necessary to re-write Python definitions in OpenSCAD rather than calling the matching Python function directly.

In earlier versions there were several possible ways to work with the 3D models of the cuts, either directly displaying the returned 3D model when explicitly called for after storing it in a variable or calling it up as a calculation (Python command `ouput(<foo>)` or OpenSCAD returning a model, or calling an appropriate OpenSCAD command), however as-of v0.9 the tool movements are modeled as lists of `hull()` operations which must be processed as such and are `differenced` from the stock. The templates set up these options as noted, and ensure that `True == true`.

PYTHON CODING CONSIDERATIONS: Python style may be checked using a tool such as: https://www.codewof.co.nz/style/python3/. Not all conventions will necessarily be adhered to — limiting line length in particular conflicts with the flexibility of Literate Programming. Note that `numpydoc`-style docstrings will be added to help define the functionality of each defined module in Python. https://numpydoc.readthedocs.io/en/latest/.

3.1 Module Naming Convention

The original implementation required three files and used a convention for prefacing commands with `o` or `p`, but this requirement was obviated in the full Python re-write. The current implentation depends upon the class being instantiated as `gcp` as a sufficent differentation between the Python and the OpenSCAD versions of commands which will otherwise share the same name.

Number will be abbreviated as `num` rather than `no`, and the short form will be used internally for variable names, while the compleat word will be used in commands.

In some instances, `the` will be used as a prefix.

Tool #s where used will be the first argument where possible — this makes it obvious if they are not used — the negative consideration, that it then doesn't allow for a usage where a DEFAULT tool is used is not an issue since the command `currenttoolnumber()` may be used to access that number, and is arguably the preferred mechanism. An exception is when there are multiple tool #s as when opening a file — collecting them all at the end is a more straight-forward approach.

In natural languages such as English, there is an order to various parts of speech such as adjectives — since various prefixes and suffixes will be used for module names, having a consistent ordering/usage will help in consistency and make expression clearer. The ordering should be: sequence (if necessary), action, function, parameter, filetype, and where possible a hierarchy of large/general to small/specific should be maintained.

- Both prefix and suffix
  - `dxf` (action (write out to DXF file), filetype)
- Prefixes
  - `generate` (Boolean) — used to identify which types of actions will be done (note that in the interest of brevity the check for this will be deferred until the last possible moment, see below)
  - `write` (action) — used to write to files, will include a check for the matching `generate` command, which being true will cause the write to the file to actually transpire
  - `cut` (action — create tool movement removing volume from 3D object)
  - `rapid` (action — create tool movement of 3D object so as to show any collision or rubbing)

  - `open` (action (file))
  - `close` (action (file))
  - `set` (action/function) — note that the matching `get` is implicit in functions which return variables, e.g., `xpos()`
  - `current`
- Nouns (shapes)
  - `arc`
  - `line`
  - `rectangle`
  - `circle`
- Suffixes
  - `feed` (parameter)
  - `gcode/gc` (filetype)
  - `pos` — position
  - `tool`
  - `loop`
  - `CC/CW`
  - `number/num` — note that `num` is used internally for variable names, while `number` will be used for module/function names, making it straight-forward to ensure that functions and variables have different names for purposes of scope

Further note that commands which are implicitly for the generation of G-code, such as `toolchange()` will omit `gc` for the sake of conciseness.

In particular, this means that the basic `cut...` and associated commands exist (or potentially exist) in the following forms and have matching versions which may be used when programming in Python or OpenSCAD:

	line			arc		
	cut	dxf	gcode	cut	dxf	gcode
cut	cutline		cutlinegc	cutarc		cutarcgc
dxf	cutlinedxf	dxfline		cutarcdxf	dxfarc	
gcode	cutlinegc		linegc	cutarcgc		arcgc
	cutlinedxfgc			cutarcdxfgc		

Note that certain commands (`dxflinegc`, `dxfarcgc`, `linegc`, `arcgc`) are either redundant or unlikely to be needed, and will most likely not be implemented (it seems contradictory that one would write out a move command to a G-code file without making that cut in the 3D preview). Note that there may be additional versions as required for the convenience of notation or cutting, in particular, a set of `cutarc`<quadrant><direction>`gc` commands was warranted during the initial development of `arc`-related commands.

A further consideration is that when processing G-code it is typical for a given command to be minimal and only include the axis of motion for the end-position, so for each of the above which is likely to appear in a `.nc` file, it will be necessary to have a matching command for the combinatorial possibilities, hence:

cutlineXYZ	cutlineXYZwithfeed
cutlineXY	cutlineXYwithfeed
cutlineXZ	cutlineXZwithfeed
cutlineYZ	cutlineYZwithfeed
cutlineX	cutlineXwithfeed
cutlineY	cutlineYwithfeed
cutlineZ	cutlineZwithfeed

Principles for naming modules (and variables):

- minimize use of underscores (for convenience sake, underscores are not used for index entries)
- identify which aspect of the project structure is being worked with (`cut`(ting), `dxf`, `gcode`, `tool`, etc.) note the `gcodepreview` class which will normally be imported as `gcp` so that module `<foo>` will be called as `gcp.<foo>` from Python and by the same `<foo>` in OpenSCAD

The following commands for various shapes either have been implemented (`monospace`) or have not yet been implemented, but likely will need to be (regular type):

- rectangle

  `cutrectangle`

  `cutrectangleround`

Another consideration is that all commands which write files will check to see if a given filetype is enabled or no, since that check is deferred to the last as noted above for the sake of conciseness.

There are multiple modes for programming PythonSCAD:

- Python — in gcodepreview this allows writing out `dxf` files
- OpenSCAD — see: https://openscad.org/documentation.html
- Programming in OpenSCAD with variables and calling Python — this requires 3 files and was originally used in the project as written up at: https://github.com/WillAdams/gcodepreview/blob/main/gcodepreview-openscad_0_6.pdf (for further details see below, notably various commented out lines in the source `.tex` file)
- Programming in OpenSCAD and calling Python where all variables as variables are held in Python classes (this is the technique used as of v0.8)
- Programming in Python and calling OpenSCAD — https://old.reddit.com/r/OpenPythonSCAD/comments/1heczmi/finally_using_scad_modules/

For reference, structurally, when developing OpenSCAD commands which make use of Python variables this was rendered as:

```
The user-facing module is \DescribeRoutine{FOOBAR}

\lstset{firstnumber=\thegcpscad}
\begin{writecode}{a}{gcodepreview.scad}{scad}
module FOOBAR(...) {
    oFOOBAR(...);
}

\end{writecode}
\addtocounter{gcpscad}{4}

which calls the internal OpenSCAD Module \DescribeSubroutine{FOOBAR}{oFOOBAR}

\begin{writecode}{a}{pygcodepreview.scad}{scad}
module oFOOBAR(...) {
    pFOOBAR(...);
 }

\end{writecode}
\addtocounter{pyscad}{4}

which in turn calls the internal Python definitioon \DescribeSubroutine{FOOBAR}{pFOOBAR}

\lstset{firstnumber=\thegcpy}
\begin{writecode}{a}{gcodepreview.py}{python}
def pFOOBAR (...)
    ...

\end{writecode}
\addtocounter{gcpy}{3}
```

Further note that this style of definition might not have been necessary for some later modules since they are in turn calling internal modules which already use this structure.

Lastly note that this style of programming was abandoned in favour of object-oriented dot notation for versions after v0.6 (see below) and that this technique was extended to class nested within another class.

3.1.1 Parameters and Default Values

Ideally, there would be *no* hard-coded values — every value used for calculation will be parameterized, and subject to control/modification. Fortunately, Python affords a feature which specifically addresses this, optional arguments with default values:
https://stackoverflow.com/questions/9539921/how-do-i-define-a-function-with-optional-argumen

In short, rather than hard-code numbers, for example in loops, they will be assigned as default values, and thus afford the user/programmer the option of changing them when the module is called.

3.2 Implementation files and gcodepreview class

Each file will begin with a comment indicating the file type and further notes/comments on usage where appropriate:

```python
#!/usr/bin/env python
#icon "C:\Program Files\PythonSCAD\bin\openscad.exe" --trust-python
#Currently tested with https://www.pythonscad.org/downloads/
    PythonSCAD_nolibfive-2025.06.04-x86-64-Installer.exe and Python
    3.11
#gcodepreview 0.9, for use with PythonSCAD,
#if using from PythonSCAD using OpenSCAD code, see gcodepreview.
    scad

import sys

# add math functions (sqrt)
import math

# getting openscad functions into namespace
#https://github.com/gsohler/openscad/issues/39
try:
    from openscad import *
except ModuleNotFoundError as e:
    print("OpenSCAD module not loaded.")

def pygcpversion():
    thegcpversion = 0.9
    return thegcpversion
```

The OpenSCAD file must `use` the Python file (note that some test/example code is commented out):

```
//!OpenSCAD

//gcodepreview version 0.8
//
//used via include <gcodepreview.scad>;
//

use <gcodepreview.py>

module gcpversion(){
echo(pygcpversion());
}

//function myfunc(var) = gcp.myfunc(var);
//
//function getvv() = gcp.getvv();
//
//module makecube(xdim, ydim, zdim){
//gcp.makecube(xdim, ydim, zdim);
//}
//
//module placecube(){
//gcp.placecube();
//}
//
//module instantiatecube(){
//gcp.instantiatecube();
//}
//
```

If all functions are to be handled within Python, then they will need to be gathered into a class which contains them and which is initialized so as to define shared variables and initial program state, and then there will need to be objects/commands for each aspect of the program, each of which will utilise needed variables and will contain appropriate functionality. Note that they will be divided between mandatory and optional functions/variables/objects:

- Mandatory
  - stocksetup:
    - stockXwidth, stockYheight, stockZthickness, zeroheight, stockzero, retractheight
  - gcpfiles:
    - basefilename, generatedxf, generategcode
  - largesquaretool:

    * large_square_tool_num, toolradius, plunge, feed, speed
  - currenttoolnum
    * endmilltype
    * diameter
    * flute
    * shaftdiameter
    * shaftheight
    * shaftlength
    * toolnumber
    * cutcolor
    * rapidcolor
    * shaftcolor

- Optional
  - smallsquaretool:
    * small_square_tool_num, small_square_ratio
  - largeballtool:
    * large_ball_tool_num, large_ball_ratio
  - largeVtool:
    * large_V_tool_num, large_V_ratio
  - smallballtool:
    * small_ball_tool_num, small_ball_ratio
  - smallVtool:
    * small_V_tool_num, small_V_ratio
  - DTtool:
    * DT_tool_num, DT_ratio
  - KHtool:
    * KH_tool_num, KH_ratio
  - Roundovertool:
    * Roundover_tool_num, RO_ratio
  - misctool:
    * MISC_tool_num, MISC_ratio

gcodepreview init

The class which is defined is `gcodepreview` which begins with the `init` method which allows passing in and defining the variables which will be used by the other methods in this class. Part of this includes handling various definitions for Boolean values.

```
gcpy class gcodepreview:
gcpy
gcpy     def __init__(self,
gcpy                  generategcode = False,
gcpy                  generatedxf = False,
gcpy                  gcpfa = 2,
gcpy                  gcpfs = 0.125,
gcpy                  steps = 10
gcpy                  ):
gcpy         """
gcpy         Initialize gcodepreview object.
gcpy
gcpy         Parameters
gcpy         ----------
gcpy         generategcode : boolean
gcpy                         Enables writing out G-code.
gcpy         generatedxf   : boolean
gcpy                         Enables writing out DXF file(s).
gcpy
gcpy         Returns
gcpy         -------
gcpy         object
gcpy             The initialized gcodepreview object.
gcpy         """
gcpy         if generategcode == 1:
gcpy             self.generategcode = True
gcpy         elif generategcode == 0:
gcpy             self.generategcode = False
gcpy         else:
gcpy             self.generategcode = generategcode
```

```
        if generatedxf == 1:
            self.generatedxf = True
        elif generatedxf == 0:
            self.generatedxf = False
        else:
            self.generatedxf = generatedxf
# unless multiple dxfs are enabled, the check for them is of course
    False
        self.generatedxfs = False
# set up 3D previewing parameters
        fa = gcpfa
        fs = gcpfs
        self.steps = steps
# initialize the machine state
        self.mc = "Initialized"
        self.mpx = float(0)
        self.mpy = float(0)
        self.mpz = float(0)
        self.tpz = float(0)
# initialize the toolpath state
        self.retractheight = 5
# initialize the DEFAULT tool
        self.currenttoolnum = 102
        self.endmilltype = "square"
        self.diameter = 3.175
        self.flute = 12.7
        self.shaftdiameter = 3.175
        self.shaftheight = 12.7
        self.shaftlength = 19.5
        self.toolnumber = "100036"
        self.cutcolor = "green"
        self.rapidcolor = "orange"
        self.shaftcolor = "red"
# the variables for holding 3D models must be initialized as empty
    lists so as to ensure that only append or extend commands are
    used with them
        self.rapids = []
        self.toolpaths = []

#    def myfunc(self, var):
#        self.vv = var * var
#        return self.vv
#
#    def getvv(self):
#        return self.vv
#
#    def checkint(self):
#        return self.mc
#
#    def makecube(self, xdim, ydim, zdim):
#        self.c=cube([xdim, ydim, zdim])
#
#    def placecube(self):
#        show(self.c)
#
#    def instantiatecube(self):
#        return self.c
```

3.2.1 Position and Variables

In modeling the machine motion and G-code it will be necessary to have the machine track several variables for machine position, the current tool and its parameters, and the current depth in the current toolpath. This will be done using paired functions (which will set and return the matching variable) and a matching variable.

The first such variables are for XYZ position:

mpx
- mpx

mpy
- mpy

mpz
- mpz

Similarly, for some toolpaths it will be necessary to track the depth along the Z-axis as the toolpath tpzinc is cut out, or the increment which a cut advances — this is done using an internal variable, `tpzinc`.

It will further be necessary to have a variable for the current tool:

currenttoolnum
- currenttoolnum

Note that the `currenttoolnum` variable should always be accessed and used for any specification of a tool, being read in whenever a tool is to be made use of, or a parameter or aspect of the tool needs to be used in a calculation.

toolmovement

In early versions, a 3D model of the tool was available as `currenttool` itself and used where appropriate, but in v0.9, this was changed to using lists for concatenating the `hull`ed shapes of tool movements, so the module, `toolmovement` which given begin/end position returns the appropriate shape(s) as a list.

xpos ypos zpos

It will be necessary to have Python functions (`xpos`, `ypos`, and `zpos`) which return the current values of the machine position in Cartesian coordinates:

```
    def xpos(self):
        return self.mpx

    def ypos(self):
        return self.mpy

    def zpos(self):
        return self.mpz
```

Wrapping these in OpenSCAD functions allows use of this positional information from OpenSCAD:

```
function xpos() = gcp.xpos();

function ypos() = gcp.ypos();

function zpos() = gcp.zpos();
```

setxpos setypos setzpos

and in turn, functions which set the positions: `setxpos`, `setypos`, and `setzpos`.

```
    def setxpos(self, newxpos):
        self.mpx = newxpos

    def setypos(self, newypos):
        self.mpy = newypos

    def setzpos(self, newzpos):
        self.mpz = newzpos
```

Using the `set...` routines will afford a single point of control if specific actions are found to be contingent on changes to these positions.

3.2.2 Initial Modules

Initializing the machine state requires zeroing out the three machine position variables:

- mpx
- mpy
- mpz

Rather than a specific command for this, the code will be in-lined where appropriate (note that if machine initialization becomes sufficiently complex to warrant it, then a suitable command will need to be coded). Note that the variables are declared in the `__init__` of the class.

toolmovement endmilltype diameter flute ra tip

The `toolmovement` class requires that the tool be defined in terms of `endmilltype`, `diameter`, `flute` (length), `ra` (radius or angle depending on context), and `tip`, and in turn defines the tool number as described below. An interface which calls this routine based on tool number will allow a return to the previous style of usage.

toolmovement rapids toolpaths

There will be two variables to record `toolmovement`, `rapids` and `toolpaths`. Initialized as empty lists, toolmovements will be `extend`ed to the lists.

gcodepreview setupstock

3.2.2.1 setupstock

The first such setup subroutine is `gcodepreview setupstock` which is appropriately enough, to set up the stock, and perform other initializations — initially, the only thing done in Python was to set the value of the persistent (Python) variables (see `initializemachinestate()` above), but the rewritten standalone version handles all necessary actions.

gcp.setupstock

Since part of a class, it will be called as `gcp.setupstock`. It requires that the user set parameters for stock dimensions and so forth, and will create comments in the G-code (if generating that file is enabled) which incorporate the stock dimensions and its position relative to the zero as set relative to the stock.

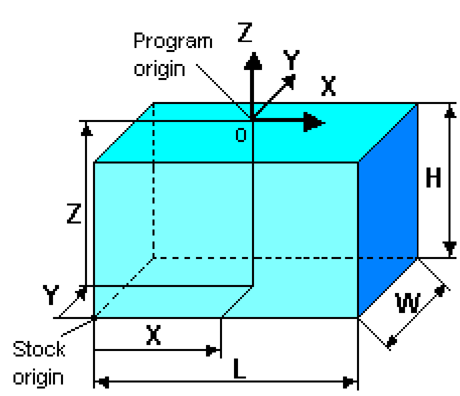
```
    def setupstock(self, stockXwidth,
                   stockYheight,
                   stockZthickness,
```

```
                 zeroheight,
                 stockzero,
                 retractheight):
        """
        Set up blank/stock for material and position/zero.

        Parameters
        ----------
        stockXwidth :   float
                        X extent/dimension
        stockYheight :  float
                        Y extent/dimension
        stockZthickness : boolean
                        Z extent/dimension
        zeroheight :    string
                        Top or Bottom, determines if Z extent will be positive or negative
        stockzero :     string
                        Lower-Left, Center-Left, Top-Left, Center, determines XY position of stock
        retractheight : float
                        Distance which tool retracts above surface of stock.

        Returns
        -------
        none
        """
        self.stockXwidth = stockXwidth
        self.stockYheight = stockYheight
        self.stockZthickness = stockZthickness
        self.zeroheight = zeroheight
        self.stockzero = stockzero
        self.retractheight = retractheight
        self.stock = cube([stockXwidth, stockYheight, stockZthickness])
```

zeroheight
stockzero

A series of `if` statements parse the `zeroheight` (Z-axis) and `stockzero` (X- and Y-axes) parameters so as to place the stock in place and suitable G-code comments are added for CutViewer. The CutViewer comments are in the form:

```
(STOCK/BLOCK, Length, Width, Height, Origin X, Origin Y, Origin Z)
```




```
        if self.zeroheight == "Top":
            if self.stockzero == "Lower-Left":
                self.stock = self.stock.translate([0, 0, -self.stockZthickness])
                if self.generategcode == True:
                    self.writegc("(stockMin:0.00mm, 0.00mm, -", str(self.stockZthickness), "mm)")
                    self.writegc("(stockMax:", str(self.stockXwidth), "mm, ", str(stockYheight), "mm, 0.00mm)")
                    self.writegc("(STOCK/BLOCK, ", str(self.stockXwidth), ", ", str(self.stockYheight), ", ", str(self.stockZthickness), ", 0.00, 0.00, ", str(self.stockZthickness), ")")
            if self.stockzero == "Center-Left":
                self.stock = self.stock.translate([0, -stockYheight / 2, -stockZthickness])
                if self.generategcode == True:
```

```
                    self.writegc("(stockMin:0.00mm, -", str(self.stockYheight/2), "mm, -", str(self.stockZthickness), "mm)")
                    self.writegc("(stockMax:", str(self.stockXwidth), "mm, ", str(self.stockYheight/2), "mm, 0.00mm)")
                    self.writegc("(STOCK/BLOCK, ", str(self.stockXwidth), ", ", str(self.stockYheight), ", ", str(self.stockZthickness), ", 0.00, ", str(self.stockYheight/2), ", ", str(self.stockZthickness), ")");
            if self.stockzero == "Top-Left":
                self.stock = self.stock.translate([0, -self.stockYheight, -self.stockZthickness])
                if self.generategcode == True:
                    self.writegc("(stockMin:0.00mm, -", str(self.stockYheight), "mm, -", str(self.stockZthickness), "mm)")
                    self.writegc("(stockMax:", str(self.stockXwidth), "mm, 0.00mm, 0.00mm)")
                    self.writegc("(STOCK/BLOCK, ", str(self.stockXwidth), ", ", str(self.stockYheight), ", ", str(self.stockZthickness), ", 0.00, ", str(self.stockYheight), ", ", str(self.stockZthickness), ")")
            if self.stockzero == "Center":
                self.stock = self.stock.translate([-self.stockXwidth / 2, -self.stockYheight / 2, -self.stockZthickness])
                if self.generategcode == True:
                    self.writegc("(stockMin: -", str(self.stockXwidth/2), ", -", str(self.stockYheight/2), "mm, -", str(self.stockZthickness), "mm)")
                    self.writegc("(stockMax:", str(self.stockXwidth/2), "mm, ", str(self.stockYheight/2), "mm, 0.00mm)")
                    self.writegc("(STOCK/BLOCK, ", str(self.stockXwidth), ", ", str(self.stockYheight), ", ", str(self.stockZthickness), ", ", str(self.stockXwidth/2), ", ", str(self.stockYheight/2), ", ", str(self.stockZthickness), ")")
        if self.zeroheight == "Bottom":
            if self.stockzero == "Lower-Left":
                 self.stock = self.stock.translate([0, 0, 0])
                 if self.generategcode == True:
                     self.writegc("(stockMin:0.00mm, 0.00mm, 0.00mm)")
                     self.writegc("(stockMax:", str(self.stockXwidth), "mm, ", str(self.stockYheight), "mm, ", str(self.stockZthickness), "mm)")
                     self.writegc("(STOCK/BLOCK, ", str(self.stockXwidth), ", ", str(self.stockYheight), ", ", str(self.stockZthickness), ", 0.00, 0.00, 0.00)")
            if self.stockzero == "Center-Left":
                self.stock = self.stock.translate([0, -self.stockYheight / 2, 0])
                if self.generategcode == True:
                    self.writegc("(stockMin:0.00mm, -", str(self.stockYheight/2), "mm, 0.00mm)")
                    self.writegc("(stockMax:", str(self.stockXwidth), "mm, ", str(self.stockYheight/2), "mm, -", str(self.stockZthickness), "mm)")
                    self.writegc("(STOCK/BLOCK, ", str(self.stockXwidth), ", ", str(self.stockYheight), ", ", str(self.stockZthickness), ", 0.00, ", str(self.stockYheight/2), ", 0.00mm)");
            if self.stockzero == "Top-Left":
                self.stock = self.stock.translate([0, -self.stockYheight, 0])
                if self.generategcode == True:
                    self.writegc("(stockMin:0.00mm, -", str(self.stockYheight), "mm, 0.00mm)")
                    self.writegc("(stockMax:", str(self.stockXwidth), "mm, 0.00mm, ", str(self.stockZthickness)
```

```
, "mm)")
                    self.writegc("(STOCK/BLOCK, ", str(self.stockXwidth), ", ", str(self.stockYheight), ", ", str(self.stockZthickness), ", 0.00, ", str(self.stockYheight), ", 0.00)")
            if self.stockzero == "Center":
                self.stock = self.stock.translate([-self.stockXwidth / 2, -self.stockYheight / 2, 0])
                if self.generategcode == True:
                    self.writegc("(stockMin: -", str(self.stockXwidth/2), ", -", str(self.stockYheight/2), "mm, 0.00mm)")
                    self.writegc("(stockMax:", str(self.stockXwidth/2), "mm, ", str(self.stockYheight/2), "mm, ", str(self.stockZthickness), "mm)")
                    self.writegc("(STOCK/BLOCK, ", str(self.stockXwidth), ", ", str(self.stockYheight), ", ", str(self.stockZthickness), ", ", str(self.stockXwidth/2), ", ", str(self.stockYheight/2), ", 0.00)")
        if self.generategcode == True:
            self.writegc("G90");
            self.writegc("G21");
```

Note that while the #102 is declared as a default tool, while it was originally necessary to call a tool change after invoking `setupstock`, in the 2024.09.03 version of PythonSCAD this requirement went away when an update which interfered with persistently setting a variable directly was fixed. The setupstock command is required if working with a 3D project, creating the block of stock which the following toolpath commands will cut away. Note that since Python in OpenPythonSCAD defers output of the 3D model, it is possible to define it once, then set up all the specifics for each possible positioning of the stock in terms of origin.

The OpenSCAD version is simply a descriptor:

```
module setupstock(stockXwidth, stockYheight, stockZthickness, zeroheight, stockzero, retractheight) {
    gcp.setupstock(stockXwidth, stockYheight, stockZthickness, zeroheight, stockzero, retractheight);
}
```

If processing G-code, the parameters passed in are necessarily different, and there is of course, no need to write out G-code.

```
    def setupcuttingarea(self, sizeX, sizeY, sizeZ, extentleft, extentfb, extentd):
#         self.initializemachinestate()
        c=cube([sizeX,sizeY,sizeZ])
        c = c.translate([extentleft,extentfb,extentd])
        self.stock = c
        self.toolpaths = []
        return c
```

3.2.3 Adjustments and Additions

For certain projects and toolpaths it will be helpful to shift the stock, and to add additional pieces to the project.

Shifting the stock is simple:

```
    def shiftstock(self, shiftX, shiftY, shiftZ):
        self.stock = self.stock.translate([shiftX, shiftY, shiftZ])
```

```
module shiftstock(shiftX, shiftY, shiftZ) {
    gcp.shiftstock(shiftX, shiftY, shiftZ);
}
```

adding stock is similar, but adds the requirement that it include options for shifting the stock:

```
    def addtostock(self, stockXwidth, stockYheight, stockZthickness,
                                 shiftX = 0,
                                 shiftY = 0,
                                 shiftZ = 0):
```

```
            addedpart = cube([stockXwidth, stockYheight,
                stockZthickness])
            addedpart = addedpart.translate([shiftX, shiftY, shiftZ])
            self.stock = self.stock.union(addedpart)
```

the OpenSCAD module is a descriptor as expected:

```
module addtostock(stockXwidth, stockYheight, stockZthickness,
    shiftX, shiftY, shiftZ) {
    gcp.addtostock(stockXwidth, stockYheight, stockZthickness,
        shiftX, shiftY, shiftZ);
}
```

3.3 Tools and Changes

Originally, it was necessary to return a shape so that modules which use a <variable>.union command would function as expected even when the 3D model created is stored in a variable.

toolmovement

Due to stack limits in OpenSCAD for the CSG tree, instead, the shapes will be stored in two variables as lists processed/created using a command toolmovement which will subsume all tool related functionality. As other routines need access to information about the current tool, appropriate routines will allow its variables will be queried.

The base/entry functionality has the instance being defined in terms of a basic set of variables (one of which is overloaded to serve multiple purposes, depending on the type of endmill).

Note that it will also be necessary to write out a tool description compatible with the program CutViewer as a G-code comment so that it may be used as a 3D previewer for the G-code for tool changes in G-code. Several forms are available as described below.

3.3.1 Numbering for Tools

Currently, the numbering scheme used is that of the various manufacturers of the tools, or descriptive short-hand numbers created for tools which lack such a designation (with a disclosure that the author is a Carbide 3D employee).

Creating any numbering scheme is like most things in life, a trade-off, balancing length and expressiveness/compleatness against simplicity and usability. The software application Carbide Create (as released by an employer of the main author) has a limit of six digits, which seems a reasonable length from a complexity/simplicity standpoint, but also potentially reasonably expressible.

It will be desirable to track the following characteristics and measurements, apportioned over the digits as follows:

$$\overbrace{\text{endmill type}}^{1}\overbrace{\text{radius/angle}}^{2-3}\overbrace{\text{cutting diameter(and tip radius for tapered ball nose)}}^{4-5}\overbrace{\text{cutting flute length}}^{6}$$

- 1st digit: endmill type:
  - 0 - "O"-flute
  - 1 - square
  - 2 - ball
  - 3 - V
  - 4 - bowl
  - 5 - tapered ball
  - 6 - roundover
  - 7 - thread-cutting
  - 8 - dovetail
  - 9 - other (e.g., keyhole, lollipop, or manufacturer number — if manufacturer number is used, then the 9 and any padding zeroes will be removed from the G-code or DXF when writing out file(s))
- 2nd and 3rd digits shape radius (ball/roundover) or angle (V), 2nd and 3rd digit together 10–99 indicate measurement in tenth of a millimeter. 2nd digit:
  - 0 - Imperial (00 indicates n/a or square)
  - any other value for both the 2nd and 3rd digits together indicate a metric measurement or an angle in degrees
- 3rd digit (if 2nd is 0 indicating Imperial)
  - 1 - 1/32$^{\text{nd}}$
  - 2 - 1/16

  - 3 - 1/8
  - 4 - 1/4
  - 5 - 5/16
  - 6 - 3/8
  - 7 - 1/2
  - 8 - 3/4
  - 9 - >1″ or other

- 4th and 5th digits cutting diameter as 2nd and 3rd above except 4th digit indicates tip radius for tapered ball nose and such tooling is only represented in Imperial measure:
- 4th digit (tapered ball nose)
  - 1 - 0.01 in (this is the 0.254mm of the #501 and 502)
  - 2 - 0.015625 in (1/64th)
  - 3 - 0.0295
  - 4 - 0.03125 in (1/32nd)
  - 5 - 0.0335
  - 6 - 0.0354
  - 7 - 0.0625 in (1/16th)
  - 8 - 0.125 in (1/8th)
  - 9 - 0.25 in (1/4)
- 6th digit cutting flute length:
  - 0 - other
  - 1 - calculate based on V angle
  - 2 - 1/16
  - 3 - 1/8
  - 4 - 1/4
  - 5 - 5/16
  - 6 - 1/2
  - 7 - 3/4
  - 8 - "long reach" or greater than 3/4″
  - 9 - calculate based on radius
- or 6th digit tip diameter for roundover tooling (added to cutting diameter to arrive at actual cutting diameter — note that these values are the same as for the tip radius of the #501 and 502)
  - 1 - 0.01 in
  - 2 - 0.015625 in (1/64th)
  - 3 - 0.0295
  - 4 - 0.03125 in (1/32nd)
  - 5 - 0.0335
  - 6 - 0.0354
  - 7 - 0.0625 in (1/16th)
  - 8 - 0.125 in (1/8th)
  - 9 - 0.25 in (1/4)

Using this technique to create tool numbers for Carbide 3D tooling we arrive at:

- Square

    #122 == 100012

    #112 == 100024

    #102 == 100036 (also #326 (Amana 46200-K))

    #201 == 100047 (also #251 and #322 (Amana 46202-K))

    #205 == 100048

    #324 == 100048 (Amana 46170-K)

- Ball

  #121 == 201012

  #111 == 202024

  #101 == 203036

  #202 == 204047

  #325 == 204048 (Amana 46376-K)

- V

  #301 == 390074

  #302 == 360071

  #327 == 360098 (Amana RC-1148)

- Single (O) flute

  #282 == 000204

  #274 == 000036

  #278 == 000047

- Tapered Ball Nose

  #501 == 530131

  #502 == 540131

(note that some dimensions were rounded off/approximated)
Extending that to the non-Carbide 3D tooling thus implemented:

- Dovetail

  814 == 814071

  45828 == 808071

- Keyhole Tool

  374 == 906043

  375 == 906053

  376 == 907040

  378 == 907050

- Roundover Tool

  56142 == 602032

  56125 == 603042

  1568 == 603032

  1570

  1572 == 604042

  1574

- Threadmill

  648 == 7

- Bowl bit

  45981

  45982

  1370

  1372

Tools which do not have calculated numbers filled in are not supported by the system as currently defined in an unambiguous fashion (instead filling in the manufacturer's tool number padded with zeros is hard-coded). Notable limitations:

- No way to indicate flute geometry beyond O-flute
- Lack of precision for metric tooling/limited support for Imperial sizes, notably, the dimensions used are scaled for smaller tooling and are not suited to larger scale tooling such as bowl bits
- No way to indicate several fairly common shapes including keyhole, lollipop, and flat-bottomed V/chamfer tools (except of course for using 9#####)

A further consideration is that it is not possible to represent tools unambiguously, so that given a tool definition it is possible to derive the manufacturer's tool number, *e.g.*,

```
self.currenttoolshape = self.toolshapes("square", 3.175, 12.7)
```

representing three different tools (Carbide 3D #201 (upcut), #251 (downcut), and #322 (Amana 46202-K)). Affording some sort of hinting to the user may be warranted, or a mechanism to allow specifying a given manufacturer tool as part of setting up a job.

A more likely scheme is that manufacturer tool numbers will be used to identify tooling, the generated number will be used internally, then the saved manufacturer number will be exported to the G-code file, or used when generating a DXF filename for a given set of tool movements.

```
    def currenttoolnumber(self):
        return(self.currenttoolnum)
```

toolchange The `toolchange` command will need to set several variables.

Mandatory variables include:

- endmilltype
  - O-flute
  - square
  - ball
  - V
  - keyhole
  - dovetail
  - roundover
  - tapered ball
- diameter
- flute

and depending on the tool geometry, several additional variables will be necessary (usually derived from `self.ra`):

- radius
- angle

an optional setting of a `toolnumber` may be useful in the future.

tool number toolchange **3.3.1.1 toolchange** This command accepts a `tool number` and assigns its characteristics as parameters. It then applies the appropriate commands for a `toolchange`. Note that it is expected that this code will be updated as needed when new tooling is introduced as additional modules which require specific tooling are added.

Note that the comments written out in G-code correspond to those used by the G-code previewing tool CutViewer (which is unfortunately, no longer readily available). Similarly, the G-code previewing functionality in this library expects that such comments will be in place so as to model the stock.

A further concern is that early versions often passed the tool into a module using a parameter. That ceased to be necessary in the 2024.09.03 version of PythonSCAD, and all modules should read the tool # from `currenttoolnumber()`.

Note that there are many varieties of tooling and not all will be directly supported, and that at need, additional tool shape support may be added under `misc`.

The original implementation created the model for the tool at the current position, and a duplicate at the end position, wrapping the twain for each end of a given movement in a `hull()` command and then applying a `union`. This approach will not work within Python, so it will be necessary to instead assign and select the tool as part of the `toolmovement` command.

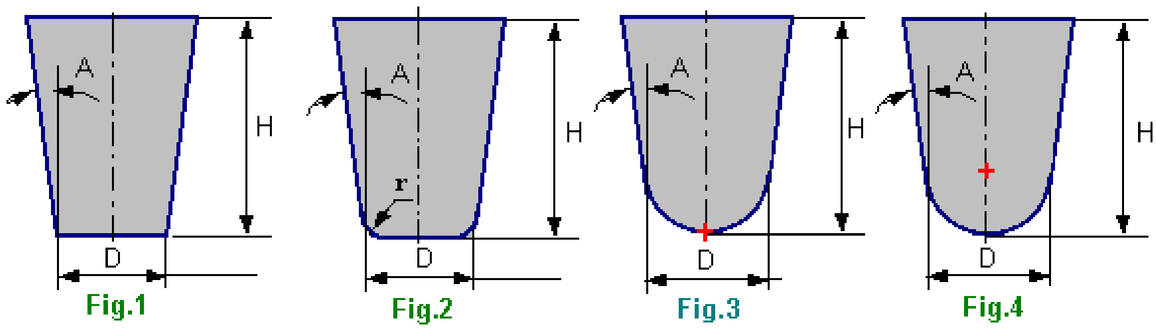
```
    def toolchange(self, tool_number, speed = 10000):
        self.currenttoolnum = tool_number

        if (self.generategcode == True):
            self.writegc("(Toolpath)")
            self.writegc("M05")
```

toolchange The Python definition for `toolchange` requires the tool number (used to write out the G-code comment description for CutViewer and also expects the speed for the current tool since this is passed into the G-code tool change command as part of the spindle on command. A simple if–then structure, the variables necessary for defining the toolshape are (re)defined each time the
toolmovement command is called so that they may be used by the command `toolmovement` for actually modeling the shapes and the path and the resultant material removal.

3.3.1.2 Square (including O-flute)

The CutViewer values include:

`TOOL/MILL, Diameter, Corner radius, Height, Taper Angle`




```
        if (tool_number == 201): #201/251/322 (Amana 46202-K) ==
            100047
            self.writegc("(TOOL/MILL,␣6.35,␣0.00,␣0.00,␣0.00)")
            self.endmilltype = "square"
            self.diameter = 6.35
            self.flute = 19.05
            self.shaftdiameter = 6.35
            self.shaftheight = 19.05
            self.shaftlength = 20.0
            self.toolnumber = "100047"
        elif (tool_number == 102): #102/326 == 100036
            self.writegc("(TOOL/MILL,␣3.175,␣0.00,␣0.00,␣0.00)")
            self.endmilltype = "square"
            self.diameter = 3.175
            self.flute = 12.7
            self.shaftdiameter = 3.175
            self.shaftheight = 12.7
            self.shaftlength = 20.0
            self.toolnumber = 100036
        elif (tool_number == 112): #112 == 100024
            self.writegc("(TOOL/MILL,␣1.5875,␣0.00,␣0.00,␣0.00)")
            self.endmilltype = "square"
            self.diameter = 1.5875
            self.flute = 6.35
            self.shaftdiameter = 3.175
            self.shaftheight = 6.35
            self.shaftlength = 12.0
            self.toolnumber = "100024"
        elif (tool_number == 122): #122 == 100012
            self.writegc("(TOOL/MILL,␣0.79375,␣0.00,␣0.00,␣0.00)")
            self.endmilltype = "square"
            self.diameter = 0.79375
            self.flute = 1.5875
            self.shaftdiameter = 3.175
            self.shaftheight = 1.5875
            self.shaftlength = 12.0
            self.toolnumber = "100012"
        elif (tool_number == 324): #324 (Amana 46170-K) == 100048
            self.writegc("(TOOL/MILL,␣6.35,␣0.00,␣0.00,␣0.00)")
            self.endmilltype = "square"
            self.diameter = 6.35
            self.flute = 22.225
            self.shaftdiameter = 6.35
            self.shaftheight = 22.225
            self.shaftlength = 20.0
            self.toolnumber = "100048"
        elif (tool_number == 205): #205 == 100048
            self.writegc("(TOOL/MILL,␣6.35,␣0.00,␣0.00,␣0.00)")
            self.endmilltype = "square"
            self.diameter = 6.35
            self.flute = 25.4
            self.shaftdiameter = 6.35
            self.shaftheight = 25.4
            self.shaftlength = 20.0
            self.toolnumber = "100048"
#
```

Making a distinction betwixt Square and O-flute tooling may be removed from a future version.

```
        elif (tool_number == 282): #282 == 000204
            self.writegc("(TOOL/MILL,␣2.0,␣0.00,␣0.00,␣0.00)")
            self.endmilltype = "O-flute"
            self.diameter = 2.0
            self.flute = 6.35
            self.shaftdiameter = 6.35
            self.shaftheight = 6.35
            self.shaftlength = 12.0
            self.toolnumber = "000204"
        elif (tool_number == 274): #274 == 000036
            self.writegc("(TOOL/MILL,␣3.175,␣0.00,␣0.00,␣0.00)")
            self.endmilltype = "O-flute"
            self.diameter = 3.175
            self.flute = 12.7
            self.shaftdiameter = 3.175
            self.shaftheight = 12.7
            self.shaftlength = 20.0
            self.toolnumber = "000036"
        elif (tool_number == 278): #278 == 000047
            self.writegc("(TOOL/MILL,␣6.35,␣0.00,␣0.00,␣0.00)")
            self.endmilltype = "O-flute"
            self.diameter = 6.35
            self.flute = 19.05
            self.shaftdiameter = 3.175
            self.shaftheight = 19.05
            self.shaftlength = 20.0
            self.toolnumber = "000047"
#
```

3.3.1.3 Ball nose (including tapered ball nose)

Additional shapes continue the `elifs`...

```
        elif (tool_number == 202): #202 == 204047
            self.writegc("(TOOL/MILL,␣6.35,␣3.175,␣0.00,␣0.00)")
            self.endmilltype = "ball"
            self.diameter = 6.35
            self.flute = 19.05
            self.shaftdiameter = 6.35
            self.shaftheight = 19.05
            self.shaftlength = 20.0
            self.toolnumber = "204047"
        elif (tool_number == 101): #101 == 203036
            self.writegc("(TOOL/MILL,␣3.175,␣1.5875,␣0.00,␣0.00)")
            self.endmilltype = "ball"
            self.diameter = 3.175
            self.flute = 12.7
            self.shaftdiameter = 3.175
            self.shaftheight = 12.7
            self.shaftlength = 20.0
            self.toolnumber = "203036"
        elif (tool_number == 111): #111 == 202024
            self.writegc("(TOOL/MILL,␣1.5875,␣0.79375,␣0.00,␣0.00)"
                )
            self.endmilltype = "ball"
            self.diameter = 1.5875
            self.flute = 6.35
            self.shaftdiameter = 3.175
            self.shaftheight = 6.35
            self.shaftlength = 20.0
            self.toolnumber = "202024"
        elif (tool_number == 121): #121 == 201012
            self.writegc("(TOOL/MILL,␣3.175,␣0.79375,␣0.00,␣0.00)")
            self.endmilltype = "ball"
            self.diameter = 0.79375
            self.flute = 1.5875
            self.shaftdiameter = 3.175
            self.shaftheight = 1.5875
            self.shaftlength = 20.0
            self.toolnumber = "201012"
        elif (tool_number == 325): #325 (Amana 46376-K) == 204048
            self.writegc("(TOOL/MILL,␣6.35,␣3.175,␣0.00,␣0.00)")
            self.endmilltype = "ball"
            self.diameter = 6.35
            self.flute = 25.4
            self.shaftdiameter = 6.35
            self.shaftheight = 25.4
            self.shaftlength = 20.0
            self.toolnumber = "204048"
```

```
#
```

3.3.1.4 V

Note that one V tool is described as an Engraver in Carbide Create. While CutViewer has specialty Tool/chamfer and Tool/drill parameters, it is possible to describe a V tool as a Tool/mill (using a very small tip radius).

```
        elif (tool_number == 301): #301 == 390074
            self.writegc("(TOOL/MILL,␣0.10,␣0.05,␣6.35,␣45.00)")
            self.endmilltype = "V"
            self.diameter = 12.7
            self.flute = 6.35
            self.angle = 90
            self.shaftdiameter = 6.35
            self.shaftheight = 6.35
            self.shaftlength = 20.0
            self.toolnumber = "390074"
        elif (tool_number == 302): #302 == 360071
            self.writegc("(TOOL/MILL,␣0.10,␣0.05,␣6.35,␣30.00)")
            self.endmilltype = "V"
            self.diameter = 12.7
            self.flute = 11.067
            self.angle = 60
            self.shaftdiameter = 6.35
            self.shaftheight = 11.067
            self.shaftlength = 20.0
            self.toolnumber = "360071"
        elif (tool_number == 390): #390 == 390032
            self.writegc("(TOOL/MILL,␣0.03,␣0.00,␣1.5875,␣45.00)")
            self.endmilltype = "V"
            self.diameter = 3.175
            self.flute = 1.5875
            self.angle = 90
            self.shaftdiameter = 3.175
            self.shaftheight = 1.5875
            self.shaftlength = 20.0
            self.toolnumber = "390032"
        elif (tool_number == 327): #327 (Amana RC-1148) == 360098
            self.writegc("(TOOL/MILL,␣0.03,␣0.00,␣13.4874,␣30.00)")
            self.endmilltype = "V"
            self.diameter = 25.4
            self.flute = 22.134
            self.angle = 60
            self.shaftdiameter = 6.35
            self.shaftheight = 22.134
            self.shaftlength = 20.0
            self.toolnumber = "360098"
        elif (tool_number == 323): #323 == 330041 30 degree V Amana
            , 45771-K
            self.writegc("(TOOL/MILL,␣0.10,␣0.05,␣11.18,␣15.00)")
            self.endmilltype = "V"
            self.diameter = 6.35
            self.flute = 11.849
            self.angle = 30
            self.shaftdiameter = 6.35
            self.shaftheight = 11.849
            self.shaftlength = 20.0
            self.toolnumber = "330041"
#
```

3.3.1.5 Keyhole

Keyhole tooling will primarily be used with a dedicated toolpath.

```
        elif (tool_number == 374): #374 == 906043
            self.writegc("(TOOL/MILL,␣9.53,␣0.00,␣3.17,␣0.00)")
            self.endmilltype = "keyhole"
            self.diameter = 9.525
            self.flute = 3.175
            self.radius = 6.35
            self.shaftdiameter = 6.35
            self.shaftheight = 3.175
            self.shaftlength = 20.0
            self.toolnumber = "906043"
        elif (tool_number == 375): #375 == 906053
            self.writegc("(TOOL/MILL,␣9.53,␣0.00,␣3.17,␣0.00)")
            self.endmilltype = "keyhole"
```

```
            self.diameter = 9.525
            self.flute = 3.175
            self.radius = 8
            self.shaftdiameter = 6.35
            self.shaftheight = 3.175
            self.shaftlength = 20.0
            self.toolnumber = "906053"
        elif (tool_number == 376): #376 == 907040
            self.writegc("(TOOL/MILL,␣12.7,␣0.00,␣4.77,␣0.00)")
            self.endmilltype = "keyhole"
            self.diameter = 12.7
            self.flute = 4.7625
            self.radius = 6.35
            self.shaftdiameter = 6.35
            self.shaftheight = 4.7625
            self.shaftlength = 20.0
            self.toolnumber = "907040"
        elif (tool_number == 378): #378 == 907050
            self.writegc("(TOOL/MILL,␣12.7,␣0.00,␣4.77,␣0.00)")
            self.endmilltype = "keyhole"
            self.diameter = 12.7
            self.flute = 4.7625
            self.radius = 8
            self.shaftdiameter = 6.35
            self.shaftheight = 4.7625
            self.shaftlength = 20.0
            self.toolnumber = "907050"
#
```

3.3.1.6 Bowl This geometry is also useful for square endmills with a radius.

```
        elif (tool_number == 45981): #45981 == 445981
#Amana Carbide Tipped Bowl & Tray 1/8 Radius x 1/2 Dia x 1/2 x 1/4
    Inch Shank
            self.writegc("(TOOL/MILL,0.03,␣0.00,␣10.00,␣30.00)")
            self.writegc("(TOOL/MILL,␣15.875,␣6.35,␣19.05,␣0.00)")
            self.endmilltype = "bowl"
            self.diameter = 12.7
            self.flute = 12.7
            self.radius = 3.175
            self.shaftdiameter = 6.35
            self.shaftheight = 12.7
            self.shaftlength = 20.0
            self.toolnumber = "445981"
        elif (tool_number == 45982):#0.507/2, 4.509
            self.writegc("(TOOL/MILL,␣15.875,␣6.35,␣19.05,␣0.00)")
            self.endmilltype = "bowl"
            self.diameter = 19.05
            self.flute = 15.875
            self.radius = 6.35
            self.shaftdiameter = 6.35
            self.shaftheight = 15.875
            self.shaftlength = 20.0
            self.toolnumber = "445982"
#        elif (tool_number == 1370): #1370 == 401370
#Whiteside Bowl & Tray Bit 1/4"SH, 1/8"R, 7/16"CD (5/16" cutting
    flute length)
            self.writegc("(TOOL/MILL,␣11.1125,␣8,␣3.175,␣0.00)")
            self.endmilltype = "bowl"
            self.diameter = 11.1125
            self.flute = 8
            self.radius = 3.175
            self.shaftdiameter = 6.35
            self.shaftheight = 8
            self.shaftlength = 20.0
            self.toolnumber = "401370"
#        elif (tool_number == 1372): #1372/45982 == 401372
#Whiteside Bowl & Tray Bit 1/4"SH, 1/4"R, 3/4"CD (5/8" cutting
    flute length)
#Amana Carbide Tipped Bowl & Tray 1/4 Radius x 3/4 Dia x 5/8 x 1/4
    Inch Shank
            self.writegc("(TOOL/MILL,␣19.5,␣15.875,␣6.35,␣0.00)")
            self.endmilltype = "bowl"
            self.diameter = 19.5
            self.flute = 15.875
            self.radius = 6.35
            self.shaftdiameter = 6.35
```

```
                self.shaftheight = 15.875
                self.shaftlength = 20.0
                self.toolnumber = "401372"
#
```

3.3.1.7 Tapered ball nose

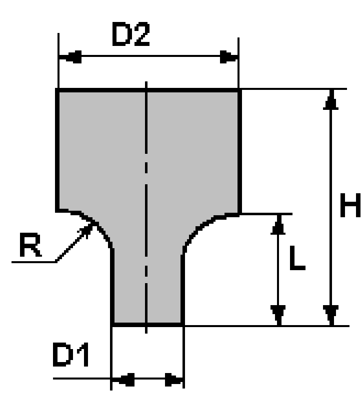
One vendor which provides such tooling is Precise Bits: https://www.precisebits.com/products/carbidebits/taperedcarve250b2f.asp&filter=7, but unfortunately, their tool numbering is ambiguous, the version of each major number (204 and 304) for their 1/4″ shank tooling which is sufficiently popular to also be offered in a ZRN coating will be used. Similarly, the #501 and #502 PCB engravers from Carbide 3D are also supported.

```
            elif (tool_number == 501): #501 == 530131
                self.writegc("(TOOL/MILL,0.03,␣0.00,␣10.00,␣30.00)")
#                self.currenttoolshape = self.toolshapes("tapered ball", 3.175, 5.561, 30, 0.254)
                self.endmilltype = "tapered␣ball"
                self.diameter = 3.175
                self.flute = 5.561
                self.angle = 30
                self.tip = 0.254
                self.shaftdiameter = 3.175
                self.shaftheight = 5.561
                self.shaftlength = 10.0
                self.toolnumber = "530131"
            elif (tool_number == 502): #502 == 540131
                self.writegc("(TOOL/MILL,0.03,␣0.00,␣10.00,␣20.00)")
#                self.currenttoolshape = self.toolshapes("tapered ball", 3.175, 4.117, 40, 0.254)
                self.endmilltype = "tapered␣ball"
                self.diameter = 3.175
                self.flute = 4.117
                self.angle = 40
                self.tip = 0.254
                self.shaftdiameter = 3.175
                self.shaftheight = 4.117
                self.shaftlength = 10.0
                self.toolnumber = "540131"
#            elif (tool_number == 204):#
#                self.writegc("()")
#                self.currenttoolshape = self.tapered_ball(1.5875, 6.35, 38.1, 3.6)
#            elif (tool_number == 304):#
#                self.writegc("()")
#                self.currenttoolshape = self.tapered_ball(3.175, 6.35, 38.1, 2.4)
#
```

3.3.1.8 Roundover (corner rounding)

Note that the parameters will need to incorporate the tip diameter into the overall diameter. CutViewer uses:

```
TOOL/CRMILL, Diameter1, Diameter2, Radius, Height, Length
```




```
            elif (tool_number == 56125):#0.508/2, 1.531 56125 == 603042
                self.writegc("(TOOL/CRMILL,␣0.508,␣6.35,␣3.175,␣7.9375,␣3.175)")
                self.endmilltype = "roundover"
                self.tip = 0.508
```

```
            self.diameter = 6.35 - self.tip
            self.flute = 8 - self.tip
            self.radius = 3.175 - self.tip
            self.shaftdiameter = 6.35
            self.shaftheight = 8
            self.shaftlength = 10.0
            self.toolnumber = "603042"
        elif (tool_number == 56142):#0.508/2, 2.921 56142 == 602032
            self.writegc("(TOOL/CRMILL, 0.508, 3.571875, 1.5875, 5.55625, 1.5875)")
            self.endmilltype = "roundover"
            self.tip = 0.508
            self.diameter = 3.175 - self.tip
            self.flute = 4.7625 - self.tip
            self.radius = 1.5875 - self.tip
            self.shaftdiameter = 3.175
            self.shaftheight = 4.7625
            self.shaftlength = 10.0
            self.toolnumber = "602032"
#        elif (tool_number == 312):#1.524/2, 3.175
#            self.writegc("(TOOL/CRMILL, Diameter1, Diameter2, Radius, Height, Length)")
#        elif (tool_number == 1568):#0.507/2, 4.509 1568 == 603032
##FIX            self.writegc("(TOOL/CRMILL, 0.17018, 9.525, 4.7625, 12.7, 4.7625)")
##            self.currenttoolshape = self.toolshapes("roundover", 3.175, 6.35, 3.175, 0.396875)
#            self.endmilltype = "roundover"
#            self.diameter = 3.175
#            self.flute = 6.35
#            self.radius = 3.175
#            self.tip = 0.396875
#            self.toolnumber = "603032"
##https://www.amanatool.com/45982-carbide-tipped-bowl-tray-1-4-radius-x-3-4-dia-x-5-8-x-1-4-inch-shank.html
#        elif (tool_number == 1570):#0.507/2, 4.509 1570 == 600002 ?!?
#            self.writegc("(TOOL/CRMILL, 0.17018, 9.525, 4.7625, 12.7, 4.7625)")
##            self.currenttoolshape = self.toolshapes("roundover", 4.7625, 9.525, 4.7625, 0.396875)
#            self.endmilltype = "roundover"
#            self.diameter = 4.7625
#            self.flute = 9.525
#            self.radius = 4.7625
#            self.tip = 0.396875
#            self.toolnumber = "600002"
#        elif (tool_number == 1572): #1572 = 604042
##FIX            self.writegc("(TOOL/CRMILL, 0.17018, 9.525, 4.7625, 12.7, 4.7625)")
##            self.currenttoolshape = self.toolshapes("roundover", 6.35, 12.7, 6.35, 0.396875)
#            self.endmilltype = "roundover"
#            self.diameter = 6.35
#            self.flute = 12.7
#            self.radius = 6.35
#            self.tip = 0.396875
#            self.toolnumber = "604042"
#        elif (tool_number == 1574): #1574 == 600062
##FIX            self.writegc("(TOOL/CRMILL, 0.17018, 9.525, 4.7625, 12.7, 4.7625)")
##            self.currenttoolshape = self.toolshapes("roundover", 9.525, 19.5, 9.515, 0.396875)
#            self.endmilltype = "roundover"
#            self.diameter = 9.525
#            self.flute = 19.5
#            self.radius = 9.515
#            self.tip = 0.396875
#            self.toolnumber = "600062"
#
```

3.3.1.9 Dovetails Unfortunately, tools which support undercuts such as dovetails are not supported by CutViewer (CAMotics will work for such tooling, at least dovetails which may be defined as "stub" endmills with a bottom diameter greater than upper diameter).

```
        elif (tool_number == 814): #814 == 814071
```

```
#Item 18J1607, 1/2" 14ř Dovetail Bit, 8mm shank
                self.writegc("(TOOL/MILL,␣12.7,␣6.367,␣12.7,␣0.00)")
            #    dt_bottomdiameter, dt_topdiameter, dt_height, dt_angle
                )
            #    https://www.leevalley.com/en-us/shop/tools/power-tool-
                accessories/router-bits/30172-dovetail-bits?item=18J1607
#                self.currenttoolshape = self.toolshapes("dovetail",
        12.7, 12.7, 14)
                self.endmilltype = "dovetail"
                self.diameter = 12.7
                self.flute = 12.7
                self.angle = 14
                self.toolnumber = "814071"
            elif (tool_number == 808079): #45828 == 808071
                self.writegc("(TOOL/MILL,␣12.7,␣6.816,␣20.95,␣0.00)")
            #    http://www.amanatool.com/45828-carbide-tipped-dovetail
                -8-deg-x-1-2-dia-x-825-x-1-4-inch-shank.html
#                self.currenttoolshape = self.toolshapes("dovetail",
        12.7, 20.955, 8)
                self.endmilltype = "dovetail"
                self.diameter = 12.7
                self.flute = 20.955
                self.angle = 8
                self.toolnumber = "808071"
#
```

Each tool must be modeled in 3D using OpenSCAD commands, but it will also be necessary to have a consistent structure for managing the various shapes and aspects of shapes.

While tool shapes were initially handled as geometric shapes stored in Python variables, processing them as such after the fashion of OpenSCAD required the use of `union()` commands and assigning a small initial object (usually a primitive placed at the origin) so that the union could take place. This has the result of creating a nested union structure in the CSG tree which can quickly become so deeply nested that it exceeds the limits set in PythonSCAD.

As was discussed in the PythonSCAD Google Group (https://groups.google.com/g/pythonscad/c/rtiYa38W8tY), if a list is used instead, then the contents of the list are added all at once at a single level when processed.

An example file which shows this concept:

```
from openscad import *
fn=200

box = cube([40,40,40])

features = []

features.append(cube([36,36,40]) + [2,2,2])
features.append(cylinder(d=20,h=5) + [20,20,-1])
features.append(cylinder(d=3,h=10) ^ [[5,35],[5,35], -1])

part = difference(box, features)

show(part)
```

As per usual, the OpenSCAD command is simply a dispatcher:

```
module toolchange(tool_number, speed){
    gcp.toolchange(tool_number, speed);
}
```

For example:

```
toolchange(small_square_tool_num, speed);
```

(the assumption is that all speed rates in a file will be the same, so as to account for the most frequent use case of a trim router with speed controlled by a dial setting and feed rates/ratios being calculated to provide the correct chipload at that setting.)

3.3.1.10 closing G-code With the tools delineated, the module is closed out and the toolchange information written into the G-code as well as the command to start the spindle at the specified speed.

```
        self.writegc("M6T", str(tool_number))
        self.writegc("M03S", str(speed))
```

3.3.2 Laser support

Two possible options for supporting a laser present themselves: color-coded DXFs or direct G-code support. An example file for the latter:
https://lasergrbl.com/test-file-and-samples/depth-of-focus-test/

```
M3 S0
S0
G0X0Y16
S1000
G1X100F1200
S0
M5 S0
M3 S0
S0
G0X0Y12
S1000
G1X100F1000
S0
M5 S0
M3 S0
S0
G0X0Y8
S1000
G1X100F800
S0
M5 S0
M3 S0
S0
G0X0Y4
S1000
G1X100F600
S0
M5 S0
M3 S0
S0
G0X0Y0
S1000
G1X100F400
S0
M5 S0
```

3.4 Shapes and tool movement

With all the scaffolding in place, it is possible to model the tool and `hull()` between copies of the 3D model of the tool, or a cross-section of it for both `cut...` and `rapid...` operations.

`cut...`
`rapid...`

The majority of commands will be more general, focusing on tooling which is generally supported by this library, moving in lines and arcs so as to describe shapes which lend themselves to representation with those tools and which match up with both toolpaths and supported geometry in Carbide Create, and the usage requirements of the typical user.

This structure has the notable advantage that if a tool shape is represented as a list and always handled thus, then representing complex shapes which need to be represented in discrete elements/parts becomes a natural thing to do and the program architecture is simpler since all possible shapes may be handled by the same code/logic with no need to identify different shapes and handle them differently.

Note that it will be preferable to use `extend` if the variable to be added contains a list rather than `append` since the former will flatten out the list and add the individual elements, so that a list remains a list of elements rather than becoming a list of lists and elements, except that there will be at least two elements to each tool model list:

- cutting *tool* shape (note that this may be either a single model, or a list of discrete slices of the tool shape)
- *shaft*

and when a cut is made by hulling each element from the cut begin position to its end position, this will be done using different colors so that the shaft rubbing may be identified on the 3D surface of the preview of the cut.

3.4.0.1 Tooling for Undercutting Toolpaths There are several notable candidates for undercutting tooling.

- Keyhole tools — intended to cut slots for retaining hardware used for picture hanging, they may be used to create slots for other purposes Note that it will be necessary to model these thrice, once for the actual keyhole cutting, second for the fluted portion of the shaft, and then the shaft should be modeled for collision https://assetssc.leevalley.com/en-gb/shop/tools/power-tool-accessories/router-bits/30113-keyhole-router-bits

- Dovetail cutters — used for the joinery of the same name, they cut a large area at the bottom which slants up to a narrower region at a defined angle
- Lollipop cutters — normally used for 3D work, as their name suggests they are essentially a (cutting) ball on a narrow stick (the tool shaft), they are mentioned here only for compleatness' sake and are not (at this time) implemented
- Threadmill — used for cutting threads, normally a single form geometry is used on a CNC.

3.4.1 Generalized commands and cuts

The first consideration is a naming convention which will allow a generalized set of associated commands to be defined. The initial version will only create OpenSCAD commands for 3D modeling and write out matching DXF files. At a later time this will be extended with G-code support.

There are three different movements in G-code which will need to be handled. Rapid commands will be used for G0 movements and will not appear in DXFs but will appear in G-code files, while straight line cut (G1) and arc (G2/G3) commands may appear in both G-code and DXF files, depending on the specific command invoked.

3.4.2 Movement and color

toolmovement shaftmovement The first command which must be defined is `toolmovement` which is used as the core of the other commands, affording a 3D model of the tool moving in a straight line. A matching `shaftmovement` command will allow modeling collision of the shaft with the stock should it occur. This differentiation raises the matter of color representation. Using a different color for the shape of the endmill when cutting and for rapid movements will similarly allow identifying instances of the tool crashing through stock at rapid speed.

```
    def setcolor(self,
                 cutcolor = "green",
                 rapidcolor = "orange",
                 shaftcolor = "red"):
        self.cutcolor = cutcolor
        self.rapidcolor = rapidcolor
        self.shaftcolor = shaftcolor
```

The possible colors are those of Web colors (https://en.wikipedia.org/wiki/Web_colors), while DXF has its own set of colors based on numbers (see table) and applying a Venn diagram and removing problematic extremes we arrive at the third column above as black and white are potentially inconsistent/confusing since at least one CAD program toggles them based on light/dark mode being applied to its interface.

Table 1: Colors in OpenSCAD and DXF

Web Colors (OpenSCAD)	DXF	Both
Black	"Black" (0)	
Red	"Red" (1)	Red
Yellow	"Yellow" (2)	Yellow
Green	"Green" (3)	Green
	"Cyan" (4)	
Blue	"Blue" (5)	Blue
	"Magenta" (6)	
White	"White" (7)	
Gray	"Dark Gray" (8)	(Dark) Gray
	"Light Gray" (9)	
Silver		
Maroon		
Olive		
Lime		
Aqua		
Teal		
Navy		
Fuchsia		
Purple		

(note that the names are not case-sensitive)

Most tools are easily implemented with concise 3D descriptions which may be connected with a simple `hull` operation. Note that extending the normal case to a pair of such operations, one for the shaft, the other for the cutting shape will markedly simplify the code, and will make it possible to color-code the shaft which may afford indication of instances of it rubbing against the stock.

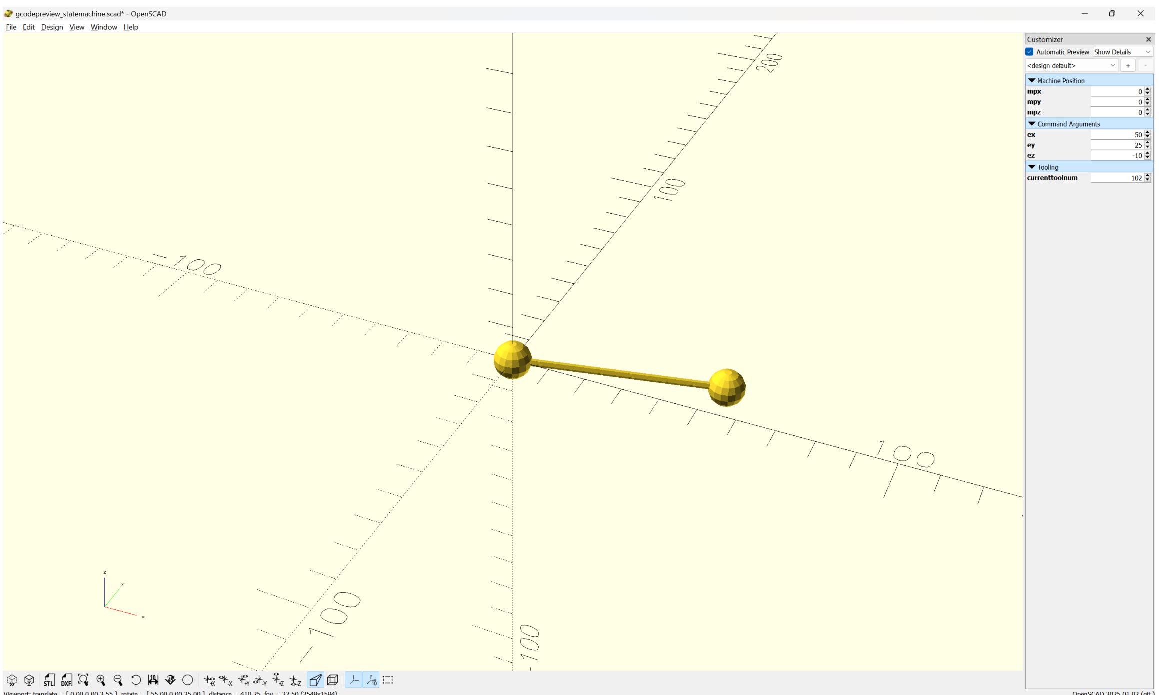
Note that the variables `self.rapids` and `self.toolpaths` are used to hold the list of accumulated 3D models of the rapid motions and cuts as elements in lists so that they may be `difference`d from the stock.

3.4.2.1 toolmovement

The `toolmovement` command incorporates the color variables to indicate cutting and differentiate rapid movements and the tool shaft.

Diagramming this is quite straight-forward — there is simply a movement made from the current position to the end. If we start at the origin, `X0`, `Y0`, `Z0`, then it is simply a straight-line movement (rapid)/cut (possibly a partial cut in the instance of a keyhole or roundover tool), and no variables change value.

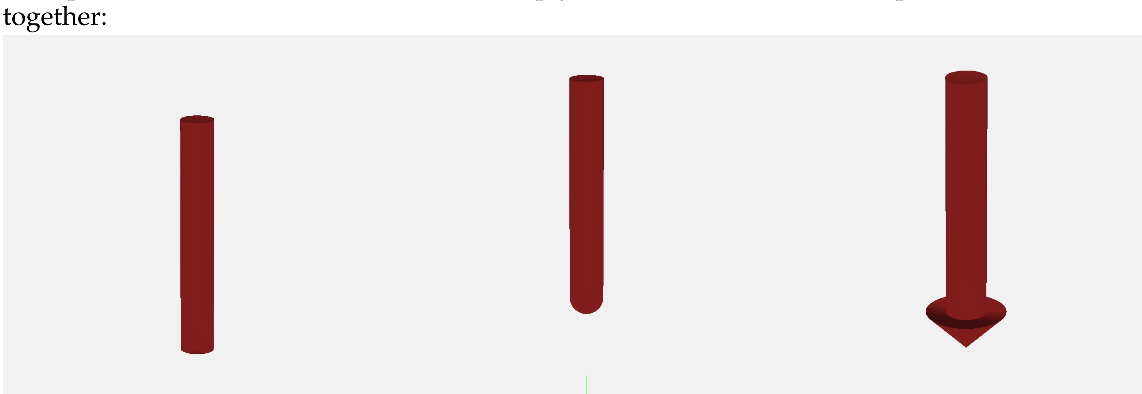
The code for diagramming this is quite straight-forward. A BlockSCAD implementation is available at: https://www.blockscad3d.com/community/projects/1894400, and the OpenSCAD version is only a little more complex (adding code to ensure positioning):




```
    def toolmovement(self, bx, by, bz, ex, ey, ez, step = 0):
        tslist = []
        if step > 0:
            steps = step
        else:
            steps = self.steps
```

3.4.2.2 Normal Tooling/toolshapes

Most tooling has quite standard shapes and are defined by their profile as defined in a class which simply defines/declares their shape and `hull()`s them together:



- Square (#201 and 102) — able to cut a flat bottom, perpendicular side and right angle, their simple and easily understood geometry makes them a standard choice
- Ballnose (#202 and 101) — rounded, they are the standard choice for concave and organic shapes
- V tooling (#301, 302 and 390) — pointed at the tip, they are available in a variety of angles and diameters and may be used for decorative V carving, or for chamfering or cutting specific angles

Note that the module for creating movement of the tool will need to handle all of the different tool shapes, generating a list of `hull()` commands which describe the 3D region which tool movement describes.

endmill square **3.4.2.3 Square (including O-flute)** The `endmill square` is a simple cylinder:

```
        if self.endmilltype == "square":
            ts = cylinder(r1=(self.diameter / 2), r2=(self.diameter
                 / 2), h=self.flute, center = False)
            tslist.append(hull(ts.translate([bx, by, bz]), ts.
               translate([ex, ey, ez])))
            return tslist

        if self.endmilltype == "O-flute":
            ts = cylinder(r1=(self.diameter / 2), r2=(self.diameter
                 / 2), h=self.flute, center = False)
            tslist.append(hull(ts.translate([bx, by, bz]), ts.
               translate([ex, ey, ez])))
            return tslist
```

ballnose **3.4.2.4 Ball nose (including tapered ball nose)** The `ballnose` is modeled as a hemisphere joined with a cylinder:

```
        if self.endmilltype == "ball":
            b = sphere(r=(self.diameter / 2))
            s = cylinder(r1=(self.diameter / 2), r2=(self.diameter
               / 2), h=self.flute, center=False)
            bs = union(b, s)
            bs = bs.translate([0, 0, (self.diameter / 2)])
            tslist.append(hull(bs.translate([bx, by, bz]), bs.
               translate([ex, ey, ez])))
            return tslist
#
```

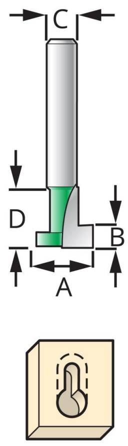
3.4.2.5 bowl The bowl tool is modeled as a series of cylinders stacked on top of each other and `hull()`ed together:

```
        if self.endmilltype == "bowl":
            inner = cylinder(r1 = self.diameter/2 - self.radius, r2
                 = self.diameter/2 - self.radius, h = self.flute)
            outer = cylinder(r1 = self.diameter/2, r2 = self.
               diameter/2, h = self.flute - self.radius)
            outer = outer.translate([0,0, self.radius])
            slices = hull(outer, inner)
#    slices = cylinder(r1 = 0.0001, r2 = 0.0001, h = 0.0001, center
   =False)
            for i in range(1, 90 - self.steps, self.steps):
                slice = cylinder(r1 = self.diameter / 2 - self.
                    radius + self.radius * Sin(i), r2 = self.
                    diameter / 2 - self.radius + self.radius * Sin(i
                    +self.steps), h = self.radius/90, center=False)
                slices = hull(slices, slice.translate([0, 0, self.
                    radius - self.radius * Cos(i+self.steps)]))
            tslist.append(hull(slices.translate([bx, by, bz]),
               slices.translate([ex, ey, ez])))
            return tslist
#
```

endmill v **3.4.2.6 V** The `endmill v` is modeled as a cylinder with a zero width base and a second cylinder for the shaft (note that Python's `math` defaults to radians, hence the need to convert from degrees):

```
        if self.endmilltype == "V":
            v = cylinder(r1=0, r2=(self.diameter / 2), h=((self.
               diameter / 2) / Tan((self.angle / 2))), center=False
               )
#              s = cylinder(r1=(self.diameter / 2), r2=(self.
   diameter / 2), h=self.flute, center=False)
#              sh = s.translate([0, 0, ((self.diameter / 2) / Tan
   ((self.angle / 2)))])
            tslist.append(hull(v.translate([bx, by, bz]), v.
               translate([ex, ey, ez])))
            return tslist
```

3.4.2.7 Keyhole Keyhole toolpaths (see: subsection 3.7.0.2.3 are intended for use with tooling which projects beyond the narrower shaft and so will cut usefully underneath the visible surface. Also described as "undercut" tooling, but see below.



Keyhole Router Bits

#	A	B	C	D
374	3/8″	1/8″	1/4″	3/8″
375	9.525mm	3.175mm	8mm	9.525mm
376	1/2″	3/16″	1/4″	1/2″
378	12.7mm	4.7625mm	8mm	12.7mm

```
        if self.endmilltype == "keyhole":
            kh = cylinder(r1=(self.diameter / 2), r2=(self.diameter
                 / 2), h=self.flute, center=False)
            sh = (cylinder(r1=(self.radius / 2), r2=(self.radius /
                2), h=self.flute*2, center=False))
            tslist.append(hull(kh.translate([bx, by, bz]), kh.
                translate([ex, ey, ez])))
            tslist.append(hull(sh.translate([bx, by, bz]), sh.
                translate([ex, ey, ez])))
            return tslist
```

3.4.2.8 Tapered ball nose The tapered ball nose tool is modeled as a sphere at the tip and a pair of cylinders, where one (a cone) describes the taper, while the other represents the shaft.

```
        if self.endmilltype == "tapered ball":
            b = sphere(r=(self.tip / 2))
            s = cylinder(r1=(self.tip / 2), r2=(self.diameter / 2),
                 h=self.flute, center=False)
            bshape = union(b, s)
            tslist.append(hull(bshape.translate([bx, by, bz]),
                bshape.translate([ex, ey, ez])))
            return tslist
```

dovetail **3.4.2.9 Dovetails** The `dovetail` is modeled as a cylinder with the differing bottom and top diameters determining the angle (though `dt_angle` is still required as a parameter)

```
        if self.endmilltype == "dovetail":
            dt = cylinder(r1=(self.diameter / 2), r2=(self.diameter
                 / 2) - self.flute * Tan(self.angle), h= self.flute,
                 center=False)
            tslist.append(hull(dt.translate([bx, by, bz]), dt.
                translate([ex, ey, ez])))
            return tslist
        if self.endmilltype == "other":
            tslist = []
#    def dovetail(self, dt_bottomdiameter, dt_topdiameter,
    dt_height, dt_angle):
#        return cylinder(r1=(dt_bottomdiameter / 2), r2=(
    dt_topdiameter / 2), h= dt_height, center=False)
```

3.4.2.10 Concave toolshapes While normal tooling may be represented with a one (or more) `hull` operation(s) betwixt two 3D toolshapes (or six in the instance of keyhole tools), concave tooling such as roundover/radius tooling require multiple sections or even slices of the tool shape to be modeled separately which are then `hull`ed together. Something of this can be seen in the manual work-around for previewing them: https://community.carbide3d.com/t/using-unsupported-tooling-in-carbide-create-roundover-cove-radius-bits/43723.

Because it is necessary to divide the tooling into vertical slices and call the hull operation for each slice the tool definitions have to be called separately in the `cut...` modules, or integrated at the lowest level.

3.4.2.11 Roundover tooling

It is not possible to represent all tools using tool changes as coded above which require using a `hull` operation between 3D representations of the tools at the beginning and end points. Tooling which cannot be so represented will be implemented separately

roundover below, see paragraph 3.4.2.10 — `roundover` tooling will need to generate a list of slices of the tool shape `hull`ed together.

```
        if self.endmilltype == "roundover":
            shaft = cylinder(self.steps, self.tip/2, self.tip/2)
            toolpath = hull(shaft.translate([bx, by, bz]), shaft.translate([ex, ey, ez]))
            shaft = cylinder(self.flute, self.diameter/2 + self.tip/2, self.diameter/2 + self.tip/2)
            toolpath = toolpath.union(hull(shaft.translate([bx, by, bz + self.radius]), shaft.translate([ex, ey, ez + self.radius])))
            tslist = [toolpath]
            slice = cylinder(0.0001, 0.0001, 0.0001)
            slices = slice
            for i in range(1, 90 - self.steps, self.steps):
                dx = self.radius*Cos(i)
                dxx = self.radius*Cos(i + self.steps)
                dzz = self.radius*Sin(i)
                dz = self.radius*Sin(i + self.steps)
                dh = dz - dzz
                slice = cylinder(r1 = self.tip/2+self.radius-dx, r2 = self.tip/2+self.radius-dxx, h = dh)
                slices = slices.union(hull(slice.translate([bx, by, bz+dz]), slice.translate([ex, ey, ez+dz])))
                tslist.append(slices)
            return tslist
#
```

Note that this routine does *not* alter the machine position variables since it may be called multiple times for a given toolpath, *e.g.*, for arcs. This command will then be called in the definitions for `rapid` and `cutline` which only differ in which variable the 3D model list is `union`ed with.

shaftmovement A similar routine will be used to handle the `shaftmovement`.

shaftmovement

3.4.2.12 shaftmovement

The `shaftmovement` command uses variables defined as part of the tool definition to determine the Z-axis position of the cylinder used to represent the shaft and its diameter and height:

```
    def shaftmovement(self, bx, by, bz, ex, ey, ez):
        tslist = []
        ts = cylinder(r1=(self.shaftdiameter / 2), r2=(self.shaftdiameter / 2), h=self.shaftlength, center = False)
        ts = ts.translate([0, 0, self.shaftheight])
        tslist.append(hull(ts.translate([bx, by, bz]), ts.translate([ex, ey, ez])))
        return tslist
```

rapid

3.4.2.13 rapid and cut (lines)

A matching pair of commands is made for these, and `rapid` is used as the basis for a series of commands which match typical usages of `G0`.

Note the addition of a Laser mode which simulates the tool having been turned off — likely further changes will be required.

```
    def rapid(self, ex, ey, ez, laser = 0):
#        print(self.rapidcolor)
        if laser == 0:
            tm = self.toolmovement(self.xpos(), self.ypos(), self.zpos(), ex, ey, ez)
            tm = color(tm, self.shaftcolor)
            ts = self.shaftmovement(self.xpos(), self.ypos(), self.zpos(), ex, ey, ez)
            ts = color(ts, self.rapidcolor)
            self.toolpaths.extend([tm, ts])
        self.setxpos(ex)
        self.setypos(ey)
        self.setzpos(ez)

    def cutline(self, ex, ey, ez):
```

```
#        print(self.cutcolor)
#        print(ex, ey, ez)
        tm = self.toolmovement(self.xpos(), self.ypos(), self.zpos(), ex, ey, ez)
        tm = color(tm, self.cutcolor)
        ts = self.shaftmovement(self.xpos(), self.ypos(), self.zpos(), ex, ey, ez)
        ts = color(ts, self.rapidcolor)
        self.setxpos(ex)
        self.setypos(ey)
        self.setzpos(ez)
        self.toolpaths.extend([tm, ts])
```

It is then possible to add specific `rapid...` commands to match typical usages of G-code. The first command needs to be a move to/from the safe Z height. In G-code this would be:

```
(Move to safe Z to avoid workholding)
G53G0Z-5.000
```

but in the 3D model, since we do not know how tall the Z-axis is, we simply move to safe height and use that as a starting point:

```
    def movetosafeZ(self):
        rapid = self.rapid(self.xpos(), self.ypos(), self.retractheight)
#        if self.generatepaths == True:
#            rapid = self.rapid(self.xpos(), self.ypos(), self.retractheight)
#            self.rapids = self.rapids.union(rapid)
#        else:
#  if (generatecode == true) {
#  //    writecomment("PREPOSITION FOR RAPID PLUNGE");Z25.650
#  //G1Z24.663F381.0, "F", str(plunge)
#        if self.generatepaths == False:
#            return rapid
#        else:
#            return cube([0.001, 0.001, 0.001])
        return rapid

    def rapidXYZ(self, ex, ey, ez):
        rapid = self.rapid(ex, ey, ez)
#        if self.generatepaths == False:
        return rapid

    def rapidXY(self, ex, ey):
        rapid = self.rapid(ex, ey, self.zpos())
#        if self.generatepaths == True:
#            self.rapids = self.rapids.union(rapid)
#        else:
#        if self.generatepaths == False:
        return rapid

    def rapidXZ(self, ex, ez):
        rapid = self.rapid(ex, self.ypos(), ez)
#        if self.generatepaths == False:
        return rapid

    def rapidYZ(self, ey, ez):
        rapid = self.rapid(self.xpos(), ey, ez)
#        if self.generatepaths == False:
        return rapid

    def rapidX(self, ex):
        rapid = self.rapid(ex, self.ypos(), self.zpos())
#        if self.generatepaths == False:
        return rapid

    def rapidY(self, ey):
        rapid = self.rapid(self.xpos(), ey, self.zpos())
#        if self.generatepaths == False:
        return rapid

    def rapidZ(self, ez):
        rapid = [self.rapid(self.xpos(), self.ypos(), ez)]
#        if self.generatepaths == True:
#            self.rapids = self.rapids.union(rapid)
#        else:
#        if self.generatepaths == False:
```

```
            return rapid
```

Note that rather than re-create the matching OpenSCAD commands as descriptors, due to the issue of redirection and return values and the possibility for errors it is more expedient to simply re-create the matching command (at least for the rapids):

```
module movetosafeZ(){
    gcp.rapid(gcp.xpos(), gcp.ypos(), retractheight);
}

module rapid(ex, ey, ez) {
    gcp.rapid(ex, ey, ez);
}

module rapidXY(ex, ey) {
    gcp.rapid(ex, ey, gcp.zpos());
}

module rapidXZ(ex, ez) {
    gcp.rapid(ex, gcp.zpos(), ez);
}

module rapidZ(ez) {
    gcp.rapid(gcp.xpos(), gcp.ypos(), ez);
}
```

Similarly, there is a series of `cutline...` commands as predicted above.

`cut...` `cutline` The Python commands `cut...` add the `currenttool` to the `toolpath` `hull`ed together at the current position and the end position of the move. For `cutline`, this is a straight-forward connection of the current (beginning) and ending coordinates:

```
    def cutlinedxf(self, ex, ey, ez):
        self.dxfline(self.currenttoolnumber(), self.xpos(), self.ypos(), ex, ey)
        self.cutline(ex, ey, ez)

    def cutlinedxfgc(self, ex, ey, ez):
        self.dxfline(self.currenttoolnumber(), self.xpos(), self.ypos(), ex, ey)
        self.writegc("G01␣X", str(ex), "␣Y", str(ey), "␣Z", str(ez))
        self.cutline(ex, ey, ez)

    def cutvertexdxf(self, ex, ey, ez):
        self.addvertex(self.currenttoolnumber(), ex, ey)
        self.writegc("G01␣X", str(ex), "␣Y", str(ey), "␣Z", str(ez))
        self.cutline(ex, ey, ez)

    def cutlineXYZwithfeed(self, ex, ey, ez, feed):
        return self.cutline(ex, ey, ez)

    def cutlineXYZ(self, ex, ey, ez):
        return self.cutline(ex, ey, ez)

    def cutlineXYwithfeed(self, ex, ey, feed):
        return self.cutline(ex, ey, self.zpos())

    def cutlineXY(self, ex, ey):
        return self.cutline(ex, ey, self.zpos())

    def cutlineXZwithfeed(self, ex, ez, feed):
        return self.cutline(ex, self.ypos(), ez)

    def cutlineXZ(self, ex, ez):
        return self.cutline(ex, self.ypos(), ez)

    def cutlineXwithfeed(self, ex, feed):
        return self.cutline(ex, self.ypos(), self.zpos())

    def cutlineX(self, ex):
        return self.cutline(ex, self.ypos(), self.zpos())

    def cutlineYZ(self, ey, ez):
        return self.cutline(self.xpos(), ey, ez)

    def cutlineYwithfeed(self, ey, feed):
```

```
        return self.cutline(self.xpos(), ey, self.zpos())

    def cutlineY(self, ey):
        return self.cutline(self.xpos(), ey, self.zpos())

    def cutlineZgcfeed(self, ez, feed):
        self.writegc("G01␣Z", str(ez), "F", str(feed))
        return self.cutline(self.xpos(), self.ypos(), ez)

    def cutlineZwithfeed(self, ez, feed):
        return self.cutline(self.xpos(), self.ypos(), ez)

    def cutlineZ(self, ez):
        return self.cutline(self.xpos(), self.ypos(), ez)
```

The matching OpenSCAD command is a descriptor:

```
module cutline(ex, ey, ez){
    gcp.cutline(ex, ey, ez);
}

module cutlinedxfgc(ex, ey, ez){
    gcp.cutlinedxfgc(ex, ey, ez);
}

module cutlineZgcfeed(ez, feed){
    gcp.cutlineZgcfeed(ez, feed);
}
```

3.4.2.14 Arcs A further consideration here is that G-code and DXF support arcs in addition to the lines already implemented. Implementing arcs wants at least the following options for quadrant and direction:

- cutarcCW — cut a partial arc described in a clock-wise direction
- cutarcCC — counter-clock-wise
- cutarcNWCW — cut the upper-left quadrant of a circle moving clockwise
- cutarcNWCC — upper-left quadrant counter-clockwise
- cutarcNECW
- cutarcNECC
- cutarcSECW
- cutarcSECC
- cutarcNECW
- cutarcNECC
- cutcircleCC — while it won't matter for generating a DXF, when G-code is implemented direction of cut will be a consideration for that
- cutcircleCW
- cutcircleCCdxf
- cutcircleCWdxf

It will be necessary to have two separate representations of arcs — the G-code and DXF may be easily and directly supported with a single command, but representing the matching tool movement in OpenSCAD will require a series of short line movements which approximate the arc cutting in each direction and at changing Z-heights so as to allow for threading and similar operations. Note that there are the following representations/interfaces for representing an arc:

- G-code — G2 (clockwise) and G3 (counter-clockwise) arcs may be specified, and since the endpoint is the positional requirement, it is most likely best to use the offset to the center (I and J), rather than the radius parameter (K) `G2/3 ...`
- DXF — `dxfarc(xcenter, ycenter, radius, anglebegin, endangle, tn)`
- approximation of arc using lines (OpenSCAD) in both clock-wise and counter-clock-wise directions

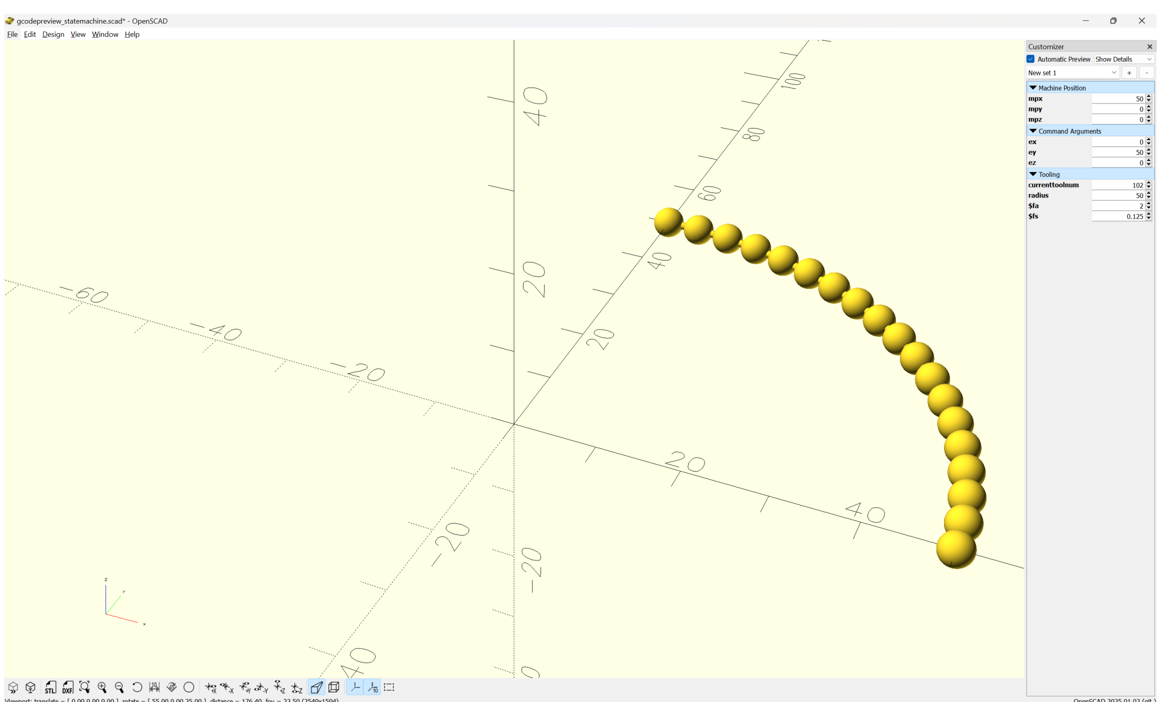
Cutting the quadrant arcs greatly simplifies the calculation and interface for the modules. A full set of 8 will be necessary, then circles will have a pair of modules (one for each cut direction) made for them.

Parameters which will need to be passed in are:

- `ex` — note that the matching origins (`bx`, `by`, `bz`) as well as the (current) toolnumber are accessed using the appropriate commands for machine position
- `ey`
- `ez` — allowing a different Z position will make possible threading and similar helical toolpaths
- `xcenter` — the center position will be specified as an absolute position which will require calculating the offset when it is used for G-code's `IJ`, for which `xctr`/`yctr` are suggested
- `ycenter`
- `radius` — while this could be calculated, passing it in as a parameter is both convenient and (potentially) could be used as a check on the other parameters
- `tpzreldim` — the relative depth (or increase in height) of the current cutting motion

`cutarcCW` `cutarcCC` Since OpenSCAD does not have an arc movement command it is necessary to iterate through a loop: `cutarcCW` (clockwise) or `cutarcCC` (counterclockwise) to handle the drawing and processing of the `cutline()` toolpaths as short line segments which additionally affords a single point of control for adding additional features such as allowing the depth to vary as one cuts along an arc (the line version is used rather than shape so as to capture the changing machine positions with each step through the loop). Note that the definition matches the `DXF` definition of defining the center position with a matching radius, but it will be necessary to move the tool to the actual origin, and to calculate the end position when writing out a `G2`/`G3` arc.

This brings to the fore the fact that at its heart, this program is simply graphing math in 3D using tools (as presaged by the book series *Make:Geometry/Trigonometry/Calculus*). This is clear in a depiction of the algorithm for the `cutarcCC`/`CW` commands, where the `x` value is the cos of the `radius` and the `y` value the sin:



The code for which makes this obvious:

```
/* [Machine Position] */
mpx = 0;
/* [Machine Position] */
mpy = 0;
/* [Machine Position] */
mpz = 0;

/* [Command Arguments] */
ex = 50;
/* [Command Arguments] */
ey = 25;
/* [Command Arguments] */
ez = -10;

/* [Tooling] */
```

```
currenttoolnum = 102;

machine_extents();

radius = 50;
$fa = 2;
$fs = 0.125;

plot_arc(radius, 0, 0, 0, radius, 0, 0, 0, radius, 0, 90, 5);

module plot_arc(bx, by, bz, ex, ey, ez, acx, acy, radius, barc, earc, inc){
for (i = [barc : inc : earc-inc]) {
  union(){
    hull()
    {
      translate([acx + cos(i)*radius,
                 acy + sin(i)*radius,
                 0]){
        sphere(r=0.5);
      }
      translate([acx + cos(i+inc)*radius,
                 acy + sin(i+inc)*radius,
                 0]){
        sphere(r=0.5);
      }
    }
      translate([acx + cos(i)*radius,
                 acy + sin(i)*radius,
                 0]){
      sphere(r=2);
    }
      translate([acx + cos(i+inc)*radius,
                 acy + sin(i+inc)*radius,
                 0]){
      sphere(r=2);
    }
  }
}
}

module machine_extents(){
translate([-200, -200, 20]){
  cube([0.001, 0.001, 0.001], center=true);
}
translate([200, 200, 20]){
  cube([0.001, 0.001, 0.001], center=true);
}
}
```

Note that it is necessary to move to the beginning cutting position before calling, and that it is necessary to pass in the relative change in Z position/depth. (Previous iterations calculated the increment of change outside the loop, but it is more workable to do so inside.)

```
    def cutarcCC(self, barc, earc, xcenter, ycenter, radius,
        tpzreldim, stepsizearc=1):
        tpzinc = tpzreldim / (earc - barc)
        i = barc
        while i < earc:
            self.cutline(xcenter + radius * Cos(math.radians(i)),
                ycenter + radius * Sin(math.radians(i)), self.zpos()
                +tpzinc)
            i += stepsizearc
        self.setxpos(xcenter + radius * Cos(math.radians(earc)))
        self.setypos(ycenter + radius * Sin(math.radians(earc)))

    def cutarcCW(self, barc, earc, xcenter, ycenter, radius,
        tpzreldim, stepsizearc=1):
#        print(str(self.zpos()))
#        print(str(ez))
#        print(str(barc - earc))
#        tpzinc = ez - self.zpos() / (barc - earc)
#        print(str(tzinc))
#        global toolpath
#        print("Entering n toolpath")
        tpzinc = tpzreldim / (barc - earc)
#        cts = self.currenttoolshape
#        toolpath = cts
```

```
#        toolpath = toolpath.translate([self.xpos(), self.ypos(), self.zpos()])
#        toolpath = []
        i = barc
        while i > earc:
            self.cutline(xcenter + radius * Cos(math.radians(i)),
                ycenter + radius * Sin(math.radians(i)), self.zpos()
                +tpzinc)
#            self.setxpos(xcenter + radius * Cos(math.radians(i)))
#            self.setypos(ycenter + radius * Sin(math.radians(i)))
#            print(str(self.xpos()), str(self.ypos(), str(self.zpos())))
#            self.setzpos(self.zpos()+tpzinc)
            i += abs(stepsizearc) * -1
#        self.dxfarc(self.currenttoolnumber(), xcenter, ycenter, radius, barc, earc)
#        if self.generatepaths == True:
#            print("Unioning n toolpath")
#            self.toolpaths = self.toolpaths.union(toolpath)
#        else:
        self.setxpos(xcenter + radius * Cos(math.radians(earc)))
        self.setypos(ycenter + radius * Sin(math.radians(earc)))
#        self.toolpaths.extend(toolpath)
#        if self.generatepaths == False:
#        return toolpath
#        else:
#            return cube([0.01, 0.01, 0.01])
```

Note that it will be necessary to add versions which write out a matching DXF element:

```
    def cutarcCWdxf(self, barc, earc, xcenter, ycenter, radius,
        tpzreldim, stepsizearc=1):
        self.cutarcCW(barc, earc, xcenter, ycenter, radius,
            tpzreldim, stepsizearc=1)
        self.dxfarc(self.currenttoolnumber(), xcenter, ycenter,
            radius, earc, barc)
#        if self.generatepaths == False:
#        return toolpath
#        else:
#            return cube([0.01, 0.01, 0.01])

    def cutarcCCdxf(self, barc, earc, xcenter, ycenter, radius,
        tpzreldim, stepsizearc=1):
        self.cutarcCC(barc, earc, xcenter, ycenter, radius,
            tpzreldim, stepsizearc=1)
        self.dxfarc(self.currenttoolnumber(), xcenter, ycenter,
            radius, barc, earc)
```

Matching OpenSCAD modules are easily made:

```
module cutarcCC(barc, earc, xcenter, ycenter, radius, tpzreldim){
    gcp.cutarcCC(barc, earc, xcenter, ycenter, radius, tpzreldim);
}

module cutarcCW(barc, earc, xcenter, ycenter, radius, tpzreldim){
    gcp.cutarcCW(barc, earc, xcenter, ycenter, radius, tpzreldim);
}
```

An alternate interface which matches how G2/G3 arcs are programmed in G-code is a useful option:

```
    def cutquarterCCNE(self, ex, ey, ez, radius):
        if self.zpos() == ez:
            tpzinc = 0
        else:
            tpzinc = (ez - self.zpos()) / 90
#        print("tpzinc ", tpzinc)
        i = 1
        while i < 91:
            self.cutline(ex + radius * Cos(i), ey - radius + radius
                * Sin(i), self.zpos()+tpzinc)
            i += 1

    def cutquarterCCNW(self, ex, ey, ez, radius):
        if self.zpos() == ez:
            tpzinc = 0
        else:
```

```
            tpzinc = (ez - self.zpos()) / 90
#            tpzinc = (self.zpos() + ez) / 90
        print("tpzinc ", tpzinc)
        i = 91
        while i < 181:
            self.cutline(ex + radius + radius * Cos(i), ey + radius * Sin(i), self.zpos()+tpzinc)
            i += 1

    def cutquarterCCSW(self, ex, ey, ez, radius):
        if self.zpos() == ez:
            tpzinc = 0
        else:
            tpzinc = (ez - self.zpos()) / 90
#            tpzinc = (self.zpos() + ez) / 90
        print("tpzinc ", tpzinc)
        i = 181
        while i < 271:
            self.cutline(ex + radius * Cos(i), ey + radius + radius * Sin(i), self.zpos()+tpzinc)
            i += 1

    def cutquarterCCSE(self, ex, ey, ez, radius):
        if self.zpos() == ez:
            tpzinc = 0
        else:
            tpzinc = (ez - self.zpos()) / 90
#            tpzinc = (self.zpos() + ez) / 90
#        print("tpzinc ", tpzinc)
        i = 271
        while i < 361:
            self.cutline(ex - radius + radius * Cos(i), ey + radius * Sin(i), self.zpos()+tpzinc)
            i += 1

    def cutquarterCCNEdxf(self, ex, ey, ez, radius):
        self.cutquarterCCNE(ex, ey, ez, radius)
        self.dxfarc(self.currenttoolnumber(), ex, ey - radius, radius,  0, 90)

    def cutquarterCCNWdxf(self, ex, ey, ez, radius):
        self.cutquarterCCNW(ex, ey, ez, radius)
        self.dxfarc(self.currenttoolnumber(), ex + radius, ey, radius, 90, 180)

    def cutquarterCCSWdxf(self, ex, ey, ez, radius):
        self.cutquarterCCSW(ex, ey, ez, radius)
        self.dxfarc(self.currenttoolnumber(), ex, ey + radius, radius, 180, 270)

    def cutquarterCCSEdxf(self, ex, ey, ez, radius):
        self.cutquarterCCSE(ex, ey, ez, radius)
        self.dxfarc(self.currenttoolnumber(), ex - radius, ey, radius, 270, 360)
```

```
module cutquarterCCNE(ex, ey, ez, radius){
     gcp.cutquarterCCNE(ex, ey, ez, radius);
}

module cutquarterCCNW(ex, ey, ez, radius){
     gcp.cutquarterCCNW(ex, ey, ez, radius);
}

module cutquarterCCSW(ex, ey, ez, radius){
    gcp.cutquarterCCSW(ex, ey, ez, radius);
}

module cutquarterCCSE(self, ex, ey, ez, radius){
    gcp.cutquarterCCSE(ex, ey, ez, radius);
}

module cutquarterCCNEdxf(ex, ey, ez, radius){
     gcp.cutquarterCCNEdxf(ex, ey, ez, radius);
}

module cutquarterCCNWdxf(ex, ey, ez, radius){
     gcp.cutquarterCCNWdxf(ex, ey, ez, radius);
```

```
}

module cutquarterCCSWdxf(ex, ey, ez, radius){
    gcp.cutquarterCCSWdxf(ex, ey, ez, radius);
}

module cutquarterCCSEdxf(self, ex, ey, ez, radius){
    gcp.cutquarterCCSEdxf(ex, ey, ez, radius);
}
```

3.4.3 tooldiameter

It will also be necessary to be able to provide the diameter of the current tool. Arguably, this would be much easier using an object-oriented programming style/dot notation.

One aspect of tool parameters which will need to be supported is shapes which create different profiles based on how deeply the tool is cutting into the surface of the material at a given point. To accommodate this, it will be necessary to either track the thickness of uncut material at any given point, or, to specify the depth of cut as a parameter.

`tool diameter`

The public-facing OpenSCAD code, `tool diameter` simply calls the matching OpenSCAD module which wraps the Python code:

```
function tool_diameter(td_tool, td_depth) = otool_diameter(td_tool,
    td_depth);
```

`tool diameter`

the Python code, `tool diameter` returns appropriate values based on the specified tool number and depth:

```
    def tool_diameter(self, ptd_tool, ptd_depth):
# Square 122, 112, 102, 201
        if ptd_tool == 122:
            return 0.79375
        if ptd_tool == 112:
            return 1.5875
        if ptd_tool == 102:
            return 3.175
        if ptd_tool == 201:
            return 6.35
# Ball 121, 111, 101, 202
        if ptd_tool == 122:
            if ptd_depth > 0.396875:
                return 0.79375
            else:
                return ptd_tool
        if ptd_tool == 112:
            if ptd_depth > 0.79375:
                return 1.5875
            else:
                return ptd_tool
        if ptd_tool == 101:
            if ptd_depth > 1.5875:
                return 3.175
            else:
                return ptd_tool
        if ptd_tool == 202:
            if ptd_depth > 3.175:
                return 6.35
            else:
                return ptd_tool
# V 301, 302, 390
        if ptd_tool == 301:
            return ptd_tool
        if ptd_tool == 302:
            return ptd_tool
        if ptd_tool == 390:
            return ptd_tool
# Keyhole
        if ptd_tool == 374:
            if ptd_depth < 3.175:
                return 9.525
            else:
                return 6.35
        if ptd_tool == 375:
            if ptd_depth < 3.175:
                return 9.525
            else:
                return 8
```

```
        if ptd_tool == 376:
            if ptd_depth < 4.7625:
                return 12.7
            else:
                return 6.35
        if ptd_tool == 378:
            if ptd_depth < 4.7625:
                return 12.7
            else:
                return 8
# Dovetail
        if ptd_tool == 814:
            if ptd_depth > 12.7:
                return 6.35
            else:
                return ptd_tool
        if ptd_tool == 808079:
            if ptd_depth > 20.95:
                return 6.816
            else:
                return ptd_tool
# Bowl Bit
#https://www.amanatool.com/45982-carbide-tipped-bowl-tray-1-4-
    radius-x-3-4-dia-x-5-8-x-1-4-inch-shank.html
        if ptd_tool == 45982:
            if ptd_depth > 6.35:
                return 15.875
            else:
                return ptd_tool
# Tapered Ball Nose
        if ptd_tool == 204:
            if ptd_depth > 6.35:
                return ptd_tool
        if ptd_tool == 304:
            if ptd_depth > 6.35:
                return ptd_tool
            else:
                return ptd_tool
```

`tool radius` Since it is often necessary to utilise the radius of the tool, an additional command, `tool radius` to return this value is worthwhile:

```
    def tool_radius(self, ptd_tool, ptd_depth):
        tr = self.tool_diameter(ptd_tool, ptd_depth)/2
        return tr
```

(Note that where values are not fully calculated values currently the passed in tool number (`ptd tool`)is returned which will need to be replaced with code which calculates the appropriate values.)

3.4.4 Feeds and Speeds

`feed` `plunge` `speed` There are several possibilities for handling feeds and speeds. Currently, base values for `feed`, `plunge`, and `speed` are used, which may then be adjusted using various `<tooldescriptor>_ratio` values, as an acknowledgement of the likelihood of a trim router being used as a spindle, the assumption is that the `speed` will remain unchanged.

The tools which need to be calculated thus are those in addition to the `large_square` tool:

- `small_square_ratio`
- `small_ball_ratio`
- `large_ball_ratio`
- `small_V_ratio`
- `large_V_ratio`
- `KH_ratio`
- `DT_ratio`

3.5 Difference of Stock, Rapids, and Toolpaths

At the end of cutting it will be necessary to subtract the accumulated toolpaths and rapids from the stock.

For Python, the initial 3D model is stored in the variable `stock`:

```
    def stockandtoolpaths(self, option = "stockandtoolpaths"):
        if option == "stock":
            show(self.stock)
        elif option == "toolpaths":
            show(self.toolpaths)
        elif option == "rapids":
            show(self.rapids)
        else:
            part = self.stock.difference(self.rapids)
            part = self.stock.difference(self.toolpaths)
            show(part)
```

Note that because of the differences in behaviour between OpenPythonSCAD (the `show()` command results in an explicit display of the requested element) and OpenSCAD (there is an implicit mechanism where the 3D element whihc is returned is displayed), the most expedient mechanism is to have an explicit Python command which returns the 3D model:

```
    def returnstockandtoolpaths(self):
        part = self.stock.difference(self.toolpaths)
        return part
```

and then make use of that specific command for OpenSCAD:

```
module stockandtoolpaths(){
    gcp.returnstockandtoolpaths();
}
```

forgoing the options of showing toolpaths and/or rapids separately.

3.6 Output files

The `gcodepreview` class will write out DXF and/or G-code files.

3.6.1 Python and OpenSCAD File Handling

`writeln` The class `gcodepreview` will need additional commands for opening files. The original implementation in RapSCAD used a command `writeln` — fortunately, this command is easily re-created in Python, though it is made as a separate file for each sort of file which may be opened. Note that the `dxf` commands will be wrapped up with `if/elif` blocks which will write to additional file(s) based on tool number as set up above.

```
    def writegc(self, *arguments):
        if self.generategcode == True:
            line_to_write = ""
            for element in arguments:
                line_to_write += element
            self.gc.write(line_to_write)
            self.gc.write("\n")

    def writedxf(self, toolnumber, *arguments):
#        global dxfclosed
        line_to_write = ""
        for element in arguments:
            line_to_write += element
        if self.generatedxf == True:
            if self.dxfclosed == False:
                self.dxf.write(line_to_write)
                self.dxf.write("\n")
        if self.generatedxfs == True:
            self.writedxfs(toolnumber, line_to_write)

    def writedxfs(self, toolnumber, line_to_write):
#        print("Processing writing toolnumber", toolnumber)
#        line_to_write = ""
#        for element in arguments:
#            line_to_write += element
        if (toolnumber == 0):
            return
        elif self.generatedxfs == True:
            if (self.large_square_tool_num == toolnumber):
                self.dxflgsq.write(line_to_write)
                self.dxflgsq.write("\n")
            if (self.small_square_tool_num == toolnumber):
                self.dxfsmsq.write(line_to_write)
                self.dxfsmsq.write("\n")
```

```
            if (self.large_ball_tool_num == toolnumber):
                self.dxflgbl.write(line_to_write)
                self.dxflgbl.write("\n")
            if (self.small_ball_tool_num == toolnumber):
                self.dxfsmbl.write(line_to_write)
                self.dxfsmbl.write("\n")
            if (self.large_V_tool_num == toolnumber):
                self.dxflgV.write(line_to_write)
                self.dxflgV.write("\n")
            if (self.small_V_tool_num == toolnumber):
                self.dxfsmV.write(line_to_write)
                self.dxfsmV.write("\n")
            if (self.DT_tool_num == toolnumber):
                self.dxfDT.write(line_to_write)
                self.dxfDT.write("\n")
            if (self.KH_tool_num == toolnumber):
                self.dxfKH.write(line_to_write)
                self.dxfKH.write("\n")
            if (self.Roundover_tool_num == toolnumber):
                self.dxfRt.write(line_to_write)
                self.dxfRt.write("\n")
            if (self.MISC_tool_num == toolnumber):
                self.dxfMt.write(line_to_write)
                self.dxfMt.write("\n")
```

which commands will accept a series of arguments and then write them out to a file object for the appropriate file. Note that the DXF files for specific tools will expect that the tool numbers be set in the matching variables from the template. Further note that while it is possible to use tools which are not so defined, the toolpaths will not be written into DXF files for any tool numbers which do not match the variables from the template (but will appear in the main `.dxf`).

opengcodefile opendxffile

For writing to files it will be necessary to have commands for opening the files: `opengcodefile` and `opendxffile` which will set the associated defaults. There is a separate function for each type of file, and for DXFs, there are multiple file instances, one for each combination of different type and size of tool which it is expected a project will work with. Each such file will be suffixed with the tool number.

There will need to be matching OpenSCAD modules for the Python functions:

```
module opendxffile(basefilename){
    gcp.opendxffile(basefilename);
}

module opendxffiles(Base_filename, large_square_tool_num,
    small_square_tool_num, large_ball_tool_num, small_ball_tool_num,
    large_V_tool_num, small_V_tool_num, DT_tool_num, KH_tool_num,
    Roundover_tool_num, MISC_tool_num) {
    gcp.opendxffiles(Base_filename, large_square_tool_num,
        small_square_tool_num, large_ball_tool_num,
        small_ball_tool_num, large_V_tool_num, small_V_tool_num,
        DT_tool_num, KH_tool_num, Roundover_tool_num, MISC_tool_num)
        ;
}
```

opengcodefile

With matching OpenSCAD commands: `opengcodefile` for OpenSCAD:

```
module opengcodefile(basefilename, currenttoolnum, toolradius,
    plunge, feed, speed) {
    gcp.opengcodefile(basefilename, currenttoolnum, toolradius,
        plunge, feed, speed);
}
```

and Python:

```
    def opengcodefile(self, basefilename = "export",
                      currenttoolnum = 102,
                      toolradius = 3.175,
                      plunge = 400,
                      feed = 1600,
                      speed = 10000
                      ):
        self.basefilename = basefilename
        self.currenttoolnum = currenttoolnum
        self.toolradius = toolradius
        self.plunge = plunge
        self.feed = feed
        self.speed = speed
        if self.generategcode == True:
```

```
            self.gcodefilename = basefilename + ".nc"
            self.gc = open(self.gcodefilename, "w")
            self.writegc("(Design␣File:␣" + self.basefilename + ")")

    def opendxffile(self, basefilename = "export"):
        self.basefilename = basefilename
#         global generatedxfs
#         global dxfclosed
        self.dxfclosed = False
        self.dxfcolor = "Black"
        if self.generatedxf == True:
            self.generatedxfs = False
            self.dxffilename = basefilename + ".dxf"
            self.dxf = open(self.dxffilename, "w")
            self.dxfpreamble(-1)

    def opendxffiles(self, basefilename = "export",
                     large_square_tool_num = 0,
                     small_square_tool_num = 0,
                     large_ball_tool_num = 0,
                     small_ball_tool_num = 0,
                     large_V_tool_num = 0,
                     small_V_tool_num = 0,
                     DT_tool_num = 0,
                     KH_tool_num = 0,
                     Roundover_tool_num = 0,
                     MISC_tool_num = 0):
#         global generatedxfs
        self.basefilename = basefilename
        self.generatedxfs = True
        self.large_square_tool_num = large_square_tool_num
        self.small_square_tool_num = small_square_tool_num
        self.large_ball_tool_num = large_ball_tool_num
        self.small_ball_tool_num = small_ball_tool_num
        self.large_V_tool_num = large_V_tool_num
        self.small_V_tool_num = small_V_tool_num
        self.DT_tool_num = DT_tool_num
        self.KH_tool_num = KH_tool_num
        self.Roundover_tool_num = Roundover_tool_num
        self.MISC_tool_num = MISC_tool_num
        if self.generatedxf == True:
            if (large_square_tool_num > 0):
                self.dxflgsqfilename = basefilename + str(large_square_tool_num) + ".dxf"
#                 print("Opening ", str(self.dxflgsqfilename))
                self.dxflgsq = open(self.dxflgsqfilename, "w")
            if (small_square_tool_num > 0):
#                 print("Opening small square")
                self.dxfsmsqfilename = basefilename + str(small_square_tool_num) + ".dxf"
                self.dxfsmsq = open(self.dxfsmsqfilename, "w")
            if (large_ball_tool_num > 0):
#                 print("Opening large ball")
                self.dxflgblfilename = basefilename + str(large_ball_tool_num) + ".dxf"
                self.dxflgbl = open(self.dxflgblfilename, "w")
            if (small_ball_tool_num > 0):
#                 print("Opening small ball")
                self.dxfsmblfilename = basefilename + str(small_ball_tool_num) + ".dxf"
                self.dxfsmbl = open(self.dxfsmblfilename, "w")
            if (large_V_tool_num > 0):
#                 print("Opening large V")
                self.dxflgVfilename = basefilename + str(large_V_tool_num) + ".dxf"
                self.dxflgV = open(self.dxflgVfilename, "w")
            if (small_V_tool_num > 0):
#                 print("Opening small V")
                self.dxfsmVfilename = basefilename + str(small_V_tool_num) + ".dxf"
                self.dxfsmV = open(self.dxfsmVfilename, "w")
            if (DT_tool_num > 0):
#                 print("Opening DT")
                self.dxfDTfilename = basefilename + str(DT_tool_num) + ".dxf"
                self.dxfDT = open(self.dxfDTfilename, "w")
            if (KH_tool_num > 0):
```

```
#                   print("Opening KH")
                    self.dxfKHfilename = basefilename + str(KH_tool_num) + ".dxf"
                    self.dxfKH = open(self.dxfKHfilename, "w")
                if (Roundover_tool_num > 0):
#                   print("Opening Rt")
                    self.dxfRtfilename = basefilename + str(Roundover_tool_num) + ".dxf"
                    self.dxfRt = open(self.dxfRtfilename, "w")
                if (MISC_tool_num > 0):
#                   print("Opening Mt")
                    self.dxfMtfilename = basefilename + str(MISC_tool_num) + ".dxf"
                    self.dxfMt = open(self.dxfMtfilename, "w")
```

For each DXF file, there will need to be a Preamble in addition to opening the file in the file system:

```
                if (large_square_tool_num > 0):
                    self.dxfpreamble(large_square_tool_num)
                if (small_square_tool_num > 0):
                    self.dxfpreamble(small_square_tool_num)
                if (large_ball_tool_num > 0):
                    self.dxfpreamble(large_ball_tool_num)
                if (small_ball_tool_num > 0):
                    self.dxfpreamble(small_ball_tool_num)
                if (large_V_tool_num > 0):
                    self.dxfpreamble(large_V_tool_num)
                if (small_V_tool_num > 0):
                    self.dxfpreamble(small_V_tool_num)
                if (DT_tool_num > 0):
                    self.dxfpreamble(DT_tool_num)
                if (KH_tool_num > 0):
                    self.dxfpreamble(KH_tool_num)
                if (Roundover_tool_num > 0):
                    self.dxfpreamble(Roundover_tool_num)
                if (MISC_tool_num > 0):
                    self.dxfpreamble(MISC_tool_num)
```

Note that the commands which interact with files include checks to see if said files are being generated.

Future considerations:

- Multiple Preview Modes:
- Fast Preview: Write all movements with both begin and end positions into a list for a specific tool — as this is done, check for a previous movement between those positions and compare depths and tool number — keep only the deepest movement for a given tool.
- Motion Preview: Work up a 3D model of the machine and actually show the stock in relation to it,

3.6.2 DXF Overview

Elements in DXFs are represented as lines or arcs. A minimal file showing both:

```
0
SECTION
2
ENTITIES
0
LWPOLYLINE
90
2
70
0
43
0
10
-31.375
20
-34.9152
10
-31.375
20
-18.75
0
ARC
```

```
10
-54.75
20
-37.5
40
4
50
0
51
90
0
ENDSEC
0
EOF
```

3.6.2.1 Writing to DXF files When the command to open `.dxf` files is called it is passed all of the variables for the various tool types/sizes, and based on a value being greater than zero, the matching file is opened, and in addition, the main DXF which is always written to is opened as well. On the gripping hand, each element which may be written to a DXF file will have a user module as well as an internal module which will be called by it so as to write to the file for the current tool. It will be necessary for the `dxfwrite` command to evaluate the tool number which is passed in, and to use an appropriate command or set of commands to then write out to the appropriate file for a given tool (if positive) or not do anything (if zero), and to write to the master file if a negative value is passed in (this allows the various DXF template commands to be written only once and then called at need).

`dxfwrite`

Each tool has a matching command for each tool/size combination:

- Ball nose, large (lgbl) `writedxflgbl`
- Ball nose, small (smbl) `writedxfsmbl`
- Square, large (lgsq) `writedxflgsq`
- Square, small (smsq) `writedxfsmsq`
- V, large (lgV) `writedxflgV`
- V, small (smV) `writedxfsmV`
- Keyhole (KH) `writedxfKH`
- Dovetail (DT) `writedxfDT`

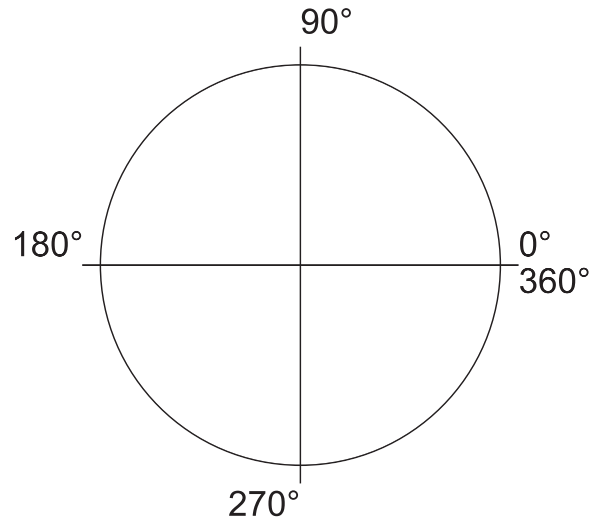
`dxfpreamble` This module requires that the tool number be passed in, and after writing out `dxfpreamble`, that value will be used to write out to the appropriate file with a series of `if` statements.

```
gcpy      def dxfpreamble(self, tn):
gcpy #        self.writedxf(tn, str(tn))
gcpy          self.writedxf(tn, "0")
gcpy          self.writedxf(tn, "SECTION")
gcpy          self.writedxf(tn, "2")
gcpy          self.writedxf(tn, "ENTITIES")
```

3.6.2.1.1 DXF Lines and Arcs There are several elements which may be written to a DXF:

- a line `dxfline`
- connected lines `beginpolyline/addvertex/closepolyline`
- arc `dxfarc`
- circle — a notable option would be for the arc to close on itself, creating a circle `dxfcircle`

DXF orders arcs counter-clockwise:



Note that arcs of greater than 90 degrees are not rendered accurately (in certain applications at least), so, for the sake of precision, they should be limited to a swing of 90 degrees or less. Further note that 4 arcs may be stitched together to make a circle:

```
dxfarc(10, 10, 5,   0,  90, small_square_tool_num);
dxfarc(10, 10, 5,  90, 180, small_square_tool_num);
dxfarc(10, 10, 5, 180, 270, small_square_tool_num);
dxfarc(10, 10, 5, 270, 360, small_square_tool_num);
```

The DXF file format supports colors defined by AutoCAD's indexed color system:

Color Code	Color Name
0	Black (or Foreground)
1	Red
2	Yellow
3	Green
4	Cyan
5	Blue
6	Magenta
7	White (or Background)
8	Dark Gray
9	Light Gray

Color codes 10–255 represent additional colors, with hues varying based on RGB values. Obviously, a command to manage adding the color commands would be:

```
    def setdxfcolor(self, color):
        self.dxfcolor = color
        self.cutcolor = color

    def writedxfcolor(self, tn):
            self.writedxf(tn, "8")
            if (self.dxfcolor == "Black"):
                self.writedxf(tn, "Layer_Black")
            if (self.dxfcolor == "Red"):
                self.writedxf(tn, "Layer_Red")
            if (self.dxfcolor == "Yellow"):
                self.writedxf(tn, "Layer_Yellow")
            if (self.dxfcolor == "Green"):
                self.writedxf(tn, "Layer_Green")
            if (self.dxfcolor == "Cyan"):
                self.writedxf(tn, "Layer_Cyan")
            if (self.dxfcolor == "Blue"):
                self.writedxf(tn, "Layer_Blue")
            if (self.dxfcolor == "Magenta"):
                self.writedxf(tn, "Layer_Magenta")
            if (self.dxfcolor == "White"):
                self.writedxf(tn, "Layer_White")
            if (self.dxfcolor == "Dark␣Gray"):
                self.writedxf(tn, "Layer_Dark_Gray")
            if (self.dxfcolor == "Light␣Gray"):
                self.writedxf(tn, "Layer_Light_Gray")

            self.writedxf(tn, "62")
            if (self.dxfcolor == "Black"):
                self.writedxf(tn, "0")
            if (self.dxfcolor == "Red"):
                self.writedxf(tn, "1")
```

```
            if (self.dxfcolor == "Yellow"):
                self.writedxf(tn, "2")
            if (self.dxfcolor == "Green"):
                self.writedxf(tn, "3")
            if (self.dxfcolor == "Cyan"):
                self.writedxf(tn, "4")
            if (self.dxfcolor == "Blue"):
                self.writedxf(tn, "5")
            if (self.dxfcolor == "Magenta"):
                self.writedxf(tn, "6")
            if (self.dxfcolor == "White"):
                self.writedxf(tn, "7")
            if (self.dxfcolor == "Dark␣Gray"):
                self.writedxf(tn, "8")
            if (self.dxfcolor == "Light␣Gray"):
                self.writedxf(tn, "9")
```

```
module setdxfcolor(color){
    gcp.setdxfcolor(color);
}
```

A further refinement would be to connect multiple line segments/arcs into a larger polyline, but since most CAM tools implicitly join elements on import, that is not necessary.

There are three possible interactions for DXF elements and toolpaths:

- describe the motion of the tool
- define a perimeter of an area which will be cut by a tool
- define a centerpoint for a specialty toolpath such as Drill or Keyhole

and it is possible that multiple such elements could be instantiated for a given toolpath.

When writing out to a DXF file there is a pair of commands, a public facing command which takes in a tool number in addition to the coordinates which then writes out to the main DXF file and then calls an internal command to which repeats the call with the tool number so as to write it out to the matching file.

```
    def dxfline(self, tn, xbegin, ybegin, xend, yend):
        self.writedxf(tn, "0")
        self.writedxf(tn, "LINE")
#
        self.writedxfcolor(tn)
#
        self.writedxf(tn, "10")
        self.writedxf(tn, str(xbegin))
        self.writedxf(tn, "20")
        self.writedxf(tn, str(ybegin))
        self.writedxf(tn, "30")
        self.writedxf(tn, "0.0")
        self.writedxf(tn, "11")
        self.writedxf(tn, str(xend))
        self.writedxf(tn, "21")
        self.writedxf(tn, str(yend))
        self.writedxf(tn, "31")
        self.writedxf(tn, "0.0")
```

In addition to `dxfline` which allows creating a line without consideration of context, there is also a `dxfpolyline` which will create a continuous/joined sequence of line segments which requires beginning it, adding vertexes, and then when done, ending the sequence.

First, begin the polyline:

```
    def beginpolyline(self, tn):#, xbegin, ybegin
        self.writedxf(tn, "0")
        self.writedxf(tn, "POLYLINE")
        self.writedxf(tn, "8")
        self.writedxf(tn, "default")
        self.writedxf(tn, "66")
        self.writedxf(tn, "1")
#
        self.writedxfcolor(tn)
#
#        self.writedxf(tn, "10")
#        self.writedxf(tn, str(xbegin))
#        self.writedxf(tn, "20")
#        self.writedxf(tn, str(ybegin))
#        self.writedxf(tn, "30")
```

```
#        self.writedxf(tn, "0.0")
        self.writedxf(tn, "70")
        self.writedxf(tn, "0")
```

then add as many vertexes as are wanted:

```
    def addvertex(self, tn, xend, yend):
        self.writedxf(tn, "0")
        self.writedxf(tn, "VERTEX")
        self.writedxf(tn, "8")
        self.writedxf(tn, "default")
        self.writedxf(tn, "70")
        self.writedxf(tn, "32")
        self.writedxf(tn, "10")
        self.writedxf(tn, str(xend))
        self.writedxf(tn, "20")
        self.writedxf(tn, str(yend))
        self.writedxf(tn, "30")
        self.writedxf(tn, "0.0")
```

then end the sequence:

```
    def closepolyline(self, tn):
        self.writedxf(tn, "0")
        self.writedxf(tn, "SEQEND")
```

For arcs, there are specific commands for writing out the DXF and G-code files. Note that for the G-code version it will be necessary to calculate the end-position, and to determine if the arc is clockwise or no (G2 vs. G3).

```
    def dxfarc(self, tn, xcenter, ycenter, radius, anglebegin,
        endangle):
        if (self.generatedxf == True):
            self.writedxf(tn, "0")
            self.writedxf(tn, "ARC")
#
            self.writedxfcolor(tn)
#
            self.writedxf(tn, "10")
            self.writedxf(tn, str(xcenter))
            self.writedxf(tn, "20")
            self.writedxf(tn, str(ycenter))
            self.writedxf(tn, "40")
            self.writedxf(tn, str(radius))
            self.writedxf(tn, "50")
            self.writedxf(tn, str(anglebegin))
            self.writedxf(tn, "51")
            self.writedxf(tn, str(endangle))

    def gcodearc(self, tn, xcenter, ycenter, radius, anglebegin,
        endangle):
        if (self.generategcode == True):
            self.writegc(tn, "(0)")
```

The various textual versions are quite obvious, and due to the requirements of G-code, it is straight-forward to include the G-code in them if it is wanted.

```
    def cutarcNECCdxf(self, ex, ey, ez, xcenter, ycenter, radius):
#        global toolpath
#        toolpath = self.currenttool()
#        toolpath = toolpath.translate([self.xpos(), self.ypos(),
    self.zpos()])
        self.dxfarc(self.currenttoolnumber(), xcenter, ycenter,
            radius, 0, 90)
        if (self.zpos == ez):
            self.settzpos(0)
        else:
            self.settzpos((self.zpos()-ez)/90)
#        self.setxpos(ex)
#        self.setypos(ey)
#        self.setzpos(ez)
#        if self.generatepaths == True:
#            print("Unioning cutarcNECCdxf toolpath")
        self.arcloop(1, 90, xcenter, ycenter, radius)
#            self.toolpaths = self.toolpaths.union(toolpath)
#        else:
```

```
#            toolpath = self.arcloop(1, 90, xcenter, ycenter, radius)
#            print("Returning cutarcNECCdxf toolpath")
        return toolpath

    def cutarcNWCCdxf(self, ex, ey, ez, xcenter, ycenter, radius):
#        global toolpath
#        toolpath = self.currenttool()
#        toolpath = toolpath.translate([self.xpos(), self.ypos(), self.zpos()])
        self.dxfarc(self.currenttoolnumber(), xcenter, ycenter, radius, 90, 180)
        if (self.zpos == ez):
            self.settzpos(0)
        else:
            self.settzpos((self.zpos()-ez)/90)
#        self.setxpos(ex)
#        self.setypos(ey)
#        self.setzpos(ez)
#        if self.generatepaths == True:
#            self.arcloop(91, 180, xcenter, ycenter, radius)
#            self.toolpaths = self.toolpaths.union(toolpath)
#        else:
        toolpath = self.arcloop(91, 180, xcenter, ycenter, radius)
        return toolpath

    def cutarcSWCCdxf(self, ex, ey, ez, xcenter, ycenter, radius):
#        global toolpath
#        toolpath = self.currenttool()
#        toolpath = toolpath.translate([self.xpos(), self.ypos(), self.zpos()])
        self.dxfarc(self.currenttoolnumber(), xcenter, ycenter, radius, 180, 270)
        if (self.zpos == ez):
            self.settzpos(0)
        else:
            self.settzpos((self.zpos()-ez)/90)
#        self.setxpos(ex)
#        self.setypos(ey)
#        self.setzpos(ez)
        if self.generatepaths == True:
            self.arcloop(181, 270, xcenter, ycenter, radius)
#            self.toolpaths = self.toolpaths.union(toolpath)
        else:
            toolpath = self.arcloop(181, 270, xcenter, ycenter, radius)
            return toolpath

    def cutarcSECCdxf(self, ex, ey, ez, xcenter, ycenter, radius):
#        global toolpath
#        toolpath = self.currenttool()
#        toolpath = toolpath.translate([self.xpos(), self.ypos(), self.zpos()])
        self.dxfarc(self.currenttoolnumber(), xcenter, ycenter, radius, 270, 360)
        if (self.zpos == ez):
            self.settzpos(0)
        else:
            self.settzpos((self.zpos()-ez)/90)
#        self.setxpos(ex)
#        self.setypos(ey)
#        self.setzpos(ez)
        if self.generatepaths == True:
            self.arcloop(271, 360, xcenter, ycenter, radius)
#            self.toolpaths = self.toolpaths.union(toolpath)
        else:
            toolpath = self.arcloop(271, 360, xcenter, ycenter, radius)
            return toolpath

    def cutarcNECWdxf(self, ex, ey, ez, xcenter, ycenter, radius):
#        global toolpath
#        toolpath = self.currenttool()
#        toolpath = toolpath.translate([self.xpos(), self.ypos(), self.zpos()])
        self.dxfarc(self.currenttoolnumber(), xcenter, ycenter, radius, 0, 90)
        if (self.zpos == ez):
```

```
            self.settzpos(0)
        else:
            self.settzpos((self.zpos()-ez)/90)
#        self.setxpos(ex)
#        self.setypos(ey)
#        self.setzpos(ez)
        if self.generatepaths == True:
            self.narcloop(89, 0, xcenter, ycenter, radius)
#            self.toolpaths = self.toolpaths.union(toolpath)
        else:
            toolpath = self.narcloop(89, 0, xcenter, ycenter, radius)
            return toolpath

    def cutarcSECWdxf(self, ex, ey, ez, xcenter, ycenter, radius):
#        global toolpath
#        toolpath = self.currenttool()
#        toolpath = toolpath.translate([self.xpos(), self.ypos(), self.zpos()])
        self.dxfarc(self.currenttoolnumber(), xcenter, ycenter, radius, 270, 360)
        if (self.zpos == ez):
            self.settzpos(0)
        else:
            self.settzpos((self.zpos()-ez)/90)
#        self.setxpos(ex)
#        self.setypos(ey)
#        self.setzpos(ez)
        if self.generatepaths == True:
            self.narcloop(359, 270, xcenter, ycenter, radius)
#            self.toolpaths = self.toolpaths.union(toolpath)
        else:
            toolpath = self.narcloop(359, 270, xcenter, ycenter, radius)
            return toolpath

    def cutarcSWCWdxf(self, ex, ey, ez, xcenter, ycenter, radius):
#        global toolpath
#        toolpath = self.currenttool()
#        toolpath = toolpath.translate([self.xpos(), self.ypos(), self.zpos()])
        self.dxfarc(self.currenttoolnumber(), xcenter, ycenter, radius, 180, 270)
        if (self.zpos == ez):
            self.settzpos(0)
        else:
            self.settzpos((self.zpos()-ez)/90)
#        self.setxpos(ex)
#        self.setypos(ey)
#        self.setzpos(ez)
        if self.generatepaths == True:
            self.narcloop(269, 180, xcenter, ycenter, radius)
#            self.toolpaths = self.toolpaths.union(toolpath)
        else:
            toolpath = self.narcloop(269, 180, xcenter, ycenter, radius)
            return toolpath

    def cutarcNWCWdxf(self, ex, ey, ez, xcenter, ycenter, radius):
#        global toolpath
#        toolpath = self.currenttool()
#        toolpath = toolpath.translate([self.xpos(), self.ypos(), self.zpos()])
        self.dxfarc(self.currenttoolnumber(), xcenter, ycenter, radius, 90, 180)
        if (self.zpos == ez):
            self.settzpos(0)
        else:
            self.settzpos((self.zpos()-ez)/90)
#        self.setxpos(ex)
#        self.setypos(ey)
#        self.setzpos(ez)
        if self.generatepaths == True:
            self.narcloop(179, 90, xcenter, ycenter, radius)
#            self.toolpaths = self.toolpaths.union(toolpath)
        else:
            toolpath = self.narcloop(179, 90, xcenter, ycenter, radius)
```

```
            return toolpath
```

Using such commands to create a circle is quite straight-forward:

```
cutarcNECCdxf(-stockXwidth/4, stockYheight/4+stockYheight/16, -stockZthickness, -stockXwidth/4, stockYhe
cutarcNWCCdxf(-(stockXwidth/4+stockYheight/16), stockYheight/4, -stockZthickness, -stockXwidth/4, stockY
cutarcSWCCdxf(-stockXwidth/4, stockYheight/4-stockYheight/16, -stockZthickness, -stockXwidth/4, stockYhe
cutarcSECCdxf(-(stockXwidth/4-stockYheight/16), stockYheight/4, -stockZthickness, -stockXwidth/4, stockY
```

```
    def arcCCgc(self, ex, ey, ez, xcenter, ycenter, radius):
        self.writegc("G03␣X", str(ex), "␣Y", str(ey), "␣Z", str(ez)
            , "␣R", str(radius))

    def arcCWgc(self, ex, ey, ez, xcenter, ycenter, radius):
        self.writegc("G02␣X", str(ex), "␣Y", str(ey), "␣Z", str(ez)
            , "␣R", str(radius))
```

The above commands may be called if G-code is also wanted with writing out G-code added:

```
    def cutarcNECCdxfgc(self, ex, ey, ez, xcenter, ycenter, radius)
        :
        self.arcCCgc(ex, ey, ez, xcenter, ycenter, radius)
        if self.generatepaths == True:
            self.cutarcNECCdxf(ex, ey, ez, xcenter, ycenter, radius
                )
        else:
            return self.cutarcNECCdxf(ex, ey, ez, xcenter, ycenter,
                radius)

    def cutarcNWCCdxfgc(self, ex, ey, ez, xcenter, ycenter, radius)
        :
        self.arcCCgc(ex, ey, ez, xcenter, ycenter, radius)
        if self.generatepaths == False:
            return self.cutarcNWCCdxf(ex, ey, ez, xcenter, ycenter,
                radius)

    def cutarcSWCCdxfgc(self, ex, ey, ez, xcenter, ycenter, radius)
        :
        self.arcCCgc(ex, ey, ez, xcenter, ycenter, radius)
        if self.generatepaths == False:
            return self.cutarcSWCCdxf(ex, ey, ez, xcenter, ycenter,
                radius)

    def cutarcSECCdxfgc(self, ex, ey, ez, xcenter, ycenter, radius)
        :
        self.arcCCgc(ex, ey, ez, xcenter, ycenter, radius)
        if self.generatepaths == False:
            return self.cutarcSECCdxf(ex, ey, ez, xcenter, ycenter,
                radius)

    def cutarcNECWdxfgc(self, ex, ey, ez, xcenter, ycenter, radius)
        :
        self.arcCWgc(ex, ey, ez, xcenter, ycenter, radius)
        if self.generatepaths == False:
            return self.cutarcNECWdxf(ex, ey, ez, xcenter, ycenter,
                radius)

    def cutarcSECWdxfgc(self, ex, ey, ez, xcenter, ycenter, radius)
        :
        self.arcCWgc(ex, ey, ez, xcenter, ycenter, radius)
        if self.generatepaths == False:
            return self.cutarcSECWdxf(ex, ey, ez, xcenter, ycenter,
                radius)

    def cutarcSWCWdxfgc(self, ex, ey, ez, xcenter, ycenter, radius)
        :
        self.arcCWgc(ex, ey, ez, xcenter, ycenter, radius)
        if self.generatepaths == False:
            return self.cutarcSWCWdxf(ex, ey, ez, xcenter, ycenter,
                radius)

    def cutarcNWCWdxfgc(self, ex, ey, ez, xcenter, ycenter, radius)
        :
        self.arcCWgc(ex, ey, ez, xcenter, ycenter, radius)
        if self.generatepaths == False:
            return self.cutarcNWCWdxf(ex, ey, ez, xcenter, ycenter,
                radius)
```

```
gcpscad module cutarcNECCdxfgc(ex, ey, ez, xcenter, ycenter, radius){
gcpscad     gcp.cutarcNECCdxfgc(ex, ey, ez, xcenter, ycenter, radius);
gcpscad }
gcpscad
gcpscad module cutarcNWCCdxfgc(ex, ey, ez, xcenter, ycenter, radius){
gcpscad     gcp.cutarcNWCCdxfgc(ex, ey, ez, xcenter, ycenter, radius);
gcpscad }
gcpscad
gcpscad module cutarcSWCCdxfgc(ex, ey, ez, xcenter, ycenter, radius){
gcpscad     gcp.cutarcSWCCdxfgc(ex, ey, ez, xcenter, ycenter, radius);
gcpscad }
gcpscad
gcpscad module cutarcSECCdxfgc(ex, ey, ez, xcenter, ycenter, radius){
gcpscad     gcp.cutarcSECCdxfgc(ex, ey, ez, xcenter, ycenter, radius);
gcpscad }
```

3.6.3 G-code Overview

The G-code commands and their matching modules may include (but are not limited to):

Command/Module	G-code
opengcodefile(s)(...); setupstock(...)	(export.nc) (stockMin: -109.5, -75mm, -8.35mm) (stockMax:109.5mm, 75mm, 0.00mm) (STOCK/BLOCK, 219, 150, 8.35, 109.5, 75, 8.35) G90 G21
movetosafez()	(Move to safe Z to avoid workholding) G53G0Z-5.000
toolchange(...);	(TOOL/MILL, 3.17, 0.00, 0.00, 0.00) M6T102 M03S16000

cutoneaxis_setfeed(...);
 	
cutwithfeed(...);
	(PREPOSITION FOR RAPID PLUNGE)
	G0X0Y0
	Z0.25
	G1Z0F100
	G1 X109.5 Y75 Z-8.35F400
	Z9

closegcodefile();	M05 M02

Conversely, the G-code commands which are supported are generated by the following modules:

G-code	Command/Module
(Design File:) (stockMin:0.00mm, -152.40mm, -34.92mm) (stockMax:109.50mm, -77.40mm, 0.00mm) (STOCK/BLOCK, 109.50, 75.00, 34.92, 0.00, 152.40, 34.92) G90 G21	opengcodefile(s)(...); setupstock(.
(Move to safe Z to avoid workholding) G53G0Z-5.000	movetosafez()
(Toolpath: Contour Toolpath 1) M05 (TOOL/MILL, 3.17, 0.00, 0.00, 0.00) M6T102 M03S10000	toolchange(...);
(PREPOSITION FOR RAPID PLUNGE)	writecomment(...)
G0X0.000Y-152.400 Z0.250	rapid(...) rapid(...)
G1Z-1.000F203.2 X109.500Y-77.400F508.0 X57.918Y16.302Z-0.726 Y22.023Z-1.023 X61.190Z-0.681 Y21.643 X57.681 Z12.700	cutwithfeed(...); cutwithfeed(...);
M05 M02	closegcodefile();

The implication here is that it should be possible to read in a G-code file, and for each line/command instantiate a matching command so as to create a 3D model/preview of the file. This is addressed by making specialized commands for movement which correspond to the various axis combinations (XYZ, XY, XZ, YZ, X, Y, Z).

A further consideration is that rather than hard-coding all possibilities or any changes, having an option for a "post-processor" will be far more flexible.

Described at: https://carbide3d.com/hub/faq/create-pro-custom-post-processor/ the necessary hooks would be:

- onOpen
- onClose
- onSection (which is where tool changes are defined, since "section" in this case is segmented per tool)

closegcodefile
closedxffile

3.6.3.1 Closings At the end of the program it will be necessary to close each file using the commands: `closegcodefile`, and `closedxffile`. In some instances it may be necessary to write additional information, depending on the file format. Note that these commands will need to be within the `gcodepreview class`.

```
    def dxfpostamble(self, tn):
#        self.writedxf(tn, str(tn))
        self.writedxf(tn, "0")
        self.writedxf(tn, "ENDSEC")
        self.writedxf(tn, "0")
        self.writedxf(tn, "EOF")
```

```
    def gcodepostamble(self):
        self.writegc("Z12.700")
        self.writegc("M05")
        self.writegc("M02")
```

dxfpostamble

It will be necessary to call the `dxfpostamble` (with appropriate checks and trappings so as to ensure that each `dxf` file is ended and closed so as to be valid.

```
    def closegcodefile(self):
        if self.generategcode == True:
            self.gcodepostamble()
            self.gc.close()

```

```
    def closedxffile(self):
        if self.generatedxf == True:
#            global dxfclosed
            self.dxfpostamble(-1)
#            self.dxfclosed = True
            self.dxf.close()

    def closedxffiles(self):
        if self.generatedxfs == True:
            if (self.large_square_tool_num > 0):
                self.dxfpostamble(self.large_square_tool_num)
            if (self.small_square_tool_num > 0):
                self.dxfpostamble(self.small_square_tool_num)
            if (self.large_ball_tool_num > 0):
                self.dxfpostamble(self.large_ball_tool_num)
            if (self.small_ball_tool_num > 0):
                self.dxfpostamble(self.small_ball_tool_num)
            if (self.large_V_tool_num > 0):
                self.dxfpostamble(self.large_V_tool_num)
            if (self.small_V_tool_num > 0):
                self.dxfpostamble(self.small_V_tool_num)
            if (self.DT_tool_num > 0):
                self.dxfpostamble(self.DT_tool_num)
            if (self.KH_tool_num > 0):
                self.dxfpostamble(self.KH_tool_num)
            if (self.Roundover_tool_num > 0):
                self.dxfpostamble(self.Roundover_tool_num)
            if (self.MISC_tool_num > 0):
                self.dxfpostamble(self.MISC_tool_num)

            if (self.large_square_tool_num > 0):
                self.dxflgsq.close()
            if (self.small_square_tool_num > 0):
                self.dxfsmsq.close()
            if (self.large_ball_tool_num > 0):
                self.dxflgbl.close()
            if (self.small_ball_tool_num > 0):
                self.dxfsmbl.close()
            if (self.large_V_tool_num > 0):
                self.dxflgV.close()
            if (self.small_V_tool_num > 0):
                self.dxfsmV.close()
            if (self.DT_tool_num > 0):
                self.dxfDT.close()
            if (self.KH_tool_num > 0):
                self.dxfKH.close()
            if (self.Roundover_tool_num > 0):
                self.dxfRt.close()
            if (self.MISC_tool_num > 0):
                self.dxfMt.close()
```

closegcodefile closedxffile dxfpostamble

The commands: `closegcodefile`, and `closedxffile` are used to close the files at the end of a program. For efficiency, each references the command: `dxfpostamble` which when called provides the boilerplate needed at the end of their respective files.

```
module closegcodefile(){
    gcp.closegcodefile();
}

module closedxffiles(){
    gcp.closedxffiles();
}

module closedxffile(){
    gcp.closedxffile();
}
```

3.7 Cutting shapes and expansion

Certain basic shapes (arcs, circles, rectangles), will be incorporated in the main code. Other shapes will be added as they are developed, and of course the user is free to develop their own systems.

It is most expedient to test out new features in a new/separate file insofar as the file structures will allow (tool definitions for example will need to consolidated in 3.3.1.1) which will need to be included in the projects which will make use of said features until such time as they are added into the main `gcodepreview.scad` file.

A basic requirement for two-dimensional regions will be to define them so as to cut them out. Two different geometric treatments will be necessary: modeling the geometry which defines the region to be cut out (output as a DXF); and modeling the movement of the tool, the toolpath which will be used in creating the 3D model and outputting the G-code.

3.7.0.1 Building blocks The outlines of shapes will be defined using:

- lines — `dxfline`
- arcs — `dxfarc`

It may be that splines or Bézier curves will be added as well.

3.7.0.2 List of shapes In the TUG presentation/paper: http://tug.org/TUGboat/tb40-2/tb125adams-3d.pdf a list of 2D shapes was put forward — which of these will need to be created, or if some more general solution will be put forward is uncertain. For the time being, shapes will be implemented on an as-needed basis, as modified by the interaction with the requirements of toolpaths. Shapes for which code exists (or is trivially coded) are indicated by Forest Green — for those which have sub-classes, if all are feasible only the higher level is so called out.

- 0
  - circle — `dxfcircle`
  - ellipse (oval) (requires some sort of non-arc curve)
    - egg-shaped
  - annulus (one circle within another, forming a ring) — handled by nested circles
  - superellipse (see astroid below)
- 1
  - cone with rounded end (arc)—see also "sector" under 3 below
- 2
  - semicircle/circular/half-circle segment (arc and a straight line); see also sector below
  - arch—curve possibly smoothly joining a pair of straight lines with a flat bottom
  - lens/vesica piscis (two convex curves)
  - lune/crescent (one convex, one concave curve)
  - heart (two curves)
  - tomoe (comma shape)—non-arc curves
- 3
  - triangle
    - equilateral
    - isosceles
    - right triangle
    - scalene
  - (circular) sector (two straight edges, one convex arc)
    - quadrant (90°)
    - sextants (60°)
    - octants (45°)
  - deltoid curve (three concave arcs)
  - Reuleaux triangle (three convex arcs)
  - arbelos (one convex, two concave arcs)
  - two straight edges, one concave arc—an example is the hyperbolic sector[1]
  - two convex, one concave arc
- 4
  - rectangle (including square) — `dxfrectangle`, `dxfrectangleround`
  - parallelogram
  - rhombus
  - trapezoid/trapezium
  - kite
  - ring/annulus segment (straight line, concave arc, straight line, convex arc)

[1] en.wikipedia.org/wiki/Hyperbolic_sector and www.reddit.com/r/Geometry/comments/bkbzgh/is_there_a_name_for_a_3_pointed_figure_with_two

- astroid (four concave arcs)
- salinon (four semicircles)
- three straight lines and one concave arc

Note that most shapes will also exist in a rounded form where sharp angles/points are replaced by arcs/portions of circles, with the most typical being `dxfrectangleround`.

Is the list of shapes for which there are not widely known names interesting for its lack of notoriety?

- two straight edges, one concave arc—oddly, an asymmetric form (hyperbolic sector) has a name, but not the symmetrical—while the colloquial/prosaic "arrowhead" was considered, it was rejected as being better applied to the shape below. (It's also the shape used for the spaceship in the game Asteroids (or Hyperspace), but that is potentially confusing with astroid.) At the conference, Dr. Knuth suggested "dart" as a suitable term.
- two convex, one concave arc—with the above named, the term "arrowhead" is freed up to use as the name for this shape.
- three straight lines and one concave arc.

The first in particular is sorely needed for this project (it's the result of inscribing a circle in a square or other regular geometric shape). Do these shapes have names in any other languages which might be used instead?

These shapes will then be used in constructing toolpaths. The program Carbide Create has toolpath types and options which are as follows:

- Contour — No Offset — the default, this is already supported in the existing code
- Contour — Outside Offset
- Contour — Inside Offset
- Pocket — such toolpaths/geometry should include the rounding of the tool at the corners, c.f., `dxfrectangleround`
- Drill — note that this is implemented as the plunging of a tool centered on a circle and normally that circle is the same diameter as the tool which is used.
- Keyhole — also beginning from a circle, the command for this also models the areas which should be cleared for the sake of reducing wear on the tool and ensuring chip clearance

Some further considerations:

- relationship of geometry to toolpath — arguably there should be an option for each toolpath (we will use Carbide Create as a reference implementation) which is to be supported. Note that there are several possibilities: modeling the tool movement, describing the outline which the tool will cut, modeling a reference shape for the toolpath
- tool geometry — support is included for specialty tooling such as dovetail cutters allowing one to to get an accurate 3D model, including for tooling which undercuts since they cannot be modeled in Carbide Create.
- Starting and Max Depth — are there CAD programs which will make use of Z-axis information in a DXF? — would it be possible/necessary to further differentiate the DXF geometry? (currently written out separately for each toolpath in addition to one combined file) — would supporting layers be an option?

3.7.0.2.1 circles Circles are made up of a series of arcs:

```
    def dxfcircle(self, tool_num, xcenter, ycenter, radius):
        self.dxfarc(tool_num, xcenter, ycenter, radius,  0, 90)
        self.dxfarc(tool_num, xcenter, ycenter, radius, 90, 180)
        self.dxfarc(tool_num, xcenter, ycenter, radius, 180, 270)
        self.dxfarc(tool_num, xcenter, ycenter, radius, 270, 360)
```

Actually cutting the circle is much the same, with the added consideration of entry point if Z height is not above the surface of the stock/already removed material, directionality (counterclockwise vs. clockwise), and depth (beginning and end depths must be specified which should allow usage of this for thread-cutting and similar purposes).

Center is specified, but the actual entry point is the right-most edge.

```
    def cutcircleCC(self, xcenter, ycenter, bz, ez, radius):
        self.setzpos(bz)
        self.cutquarterCCNE(xcenter, ycenter + radius, self.zpos()
            + ez/4, radius)
```

```
        self.cutquarterCCNW(xcenter - radius, ycenter, self.zpos() + ez/4, radius)
        self.cutquarterCCSW(xcenter, ycenter - radius, self.zpos() + ez/4, radius)
        self.cutquarterCCSE(xcenter + radius, ycenter, self.zpos() + ez/4, radius)

    def cutcircleCCdxf(self, xcenter, ycenter, bz, ez, radius):
        self.cutcircleCC(self, xcenter, ycenter, bz, ez, radius)
        self.dxfcircle(self, tool_num, xcenter, ycenter, radius)
```

A Drill toolpath is a simple plunge operation which will have a matching circle to define it.

3.7.0.2.2 rectangles There are two obvious forms for rectangles, square cornered and rounded:

```
    def dxfrectangle(self, tool_num, xorigin, yorigin, xwidth, yheight, corners = "Square", radius = 6):
        if corners == "Square":
            self.dxfline(tool_num, xorigin, yorigin, xorigin + xwidth, yorigin)
            self.dxfline(tool_num, xorigin + xwidth, yorigin, xorigin + xwidth, yorigin + yheight)
            self.dxfline(tool_num, xorigin + xwidth, yorigin + yheight, xorigin, yorigin + yheight)
            self.dxfline(tool_num, xorigin, yorigin + yheight, xorigin, yorigin)
        elif corners == "Fillet":
            self.dxfrectangleround(tool_num, xorigin, yorigin, xwidth, yheight, radius)
        elif corners == "Chamfer":
            self.dxfrectanglechamfer(tool_num, xorigin, yorigin, xwidth, yheight, radius)
        elif corners == "Flipped Fillet":
            self.dxfrectangleflippedfillet(tool_num, xorigin, yorigin, xwidth, yheight, radius)
```

Note that the rounded shape below would be described as a rectangle with the “Fillet” corner treatment in Carbide Create.

```
    def dxfrectangleround(self, tool_num, xorigin, yorigin, xwidth, yheight, radius):
# begin section
        self.writedxf(tool_num, "0")
        self.writedxf(tool_num, "SECTION")
        self.writedxf(tool_num, "2")
        self.writedxf(tool_num, "ENTITIES")
        self.writedxf(tool_num, "0")
        self.writedxf(tool_num, "LWPOLYLINE")
        self.writedxf(tool_num, "5")
        self.writedxf(tool_num, "4E")
        self.writedxf(tool_num, "100")
        self.writedxf(tool_num, "AcDbEntity")
        self.writedxf(tool_num, "8")
        self.writedxf(tool_num, "0")
        self.writedxf(tool_num, "6")
        self.writedxf(tool_num, "ByLayer")
#
        self.writedxfcolor(tool_num)
#
        self.writedxf(tool_num, "370")
        self.writedxf(tool_num, "-1")
        self.writedxf(tool_num, "100")
        self.writedxf(tool_num, "AcDbPolyline")
        self.writedxf(tool_num, "90")
        self.writedxf(tool_num, "8")
        self.writedxf(tool_num, "70")
        self.writedxf(tool_num, "1")
        self.writedxf(tool_num, "43")
        self.writedxf(tool_num, "0")
#1 upper right corner before arc (counter-clockwise)
        self.writedxf(tool_num, "10")
        self.writedxf(tool_num, str(xorigin + xwidth))
        self.writedxf(tool_num, "20")
        self.writedxf(tool_num, str(yorigin + yheight - radius))
        self.writedxf(tool_num, "42")
        self.writedxf(tool_num, "0.414213562373095")
```

```
#2 upper right corner after arc
        self.writedxf(tool_num, "10")
        self.writedxf(tool_num, str(xorigin + xwidth - radius))
        self.writedxf(tool_num, "20")
        self.writedxf(tool_num, str(yorigin + yheight))
#3 upper left corner before arc (counter-clockwise)
        self.writedxf(tool_num, "10")
        self.writedxf(tool_num, str(xorigin + radius))
        self.writedxf(tool_num, "20")
        self.writedxf(tool_num, str(yorigin + yheight))
        self.writedxf(tool_num, "42")
        self.writedxf(tool_num, "0.414213562373095")
#4 upper left corner after arc
        self.writedxf(tool_num, "10")
        self.writedxf(tool_num, str(xorigin))
        self.writedxf(tool_num, "20")
        self.writedxf(tool_num, str(yorigin + yheight - radius))
#5 lower left corner before arc (counter-clockwise)
        self.writedxf(tool_num, "10")
        self.writedxf(tool_num, str(xorigin))
        self.writedxf(tool_num, "20")
        self.writedxf(tool_num, str(yorigin + radius))
        self.writedxf(tool_num, "42")
        self.writedxf(tool_num, "0.414213562373095")
#6 lower left corner after arc
        self.writedxf(tool_num, "10")
        self.writedxf(tool_num, str(xorigin + radius))
        self.writedxf(tool_num, "20")
        self.writedxf(tool_num, str(yorigin))
#7 lower right corner before arc (counter-clockwise)
        self.writedxf(tool_num, "10")
        self.writedxf(tool_num, str(xorigin + xwidth - radius))
        self.writedxf(tool_num, "20")
        self.writedxf(tool_num, str(yorigin))
        self.writedxf(tool_num, "42")
        self.writedxf(tool_num, "0.414213562373095")
#8 lower right corner after arc
        self.writedxf(tool_num, "10")
        self.writedxf(tool_num, str(xorigin + xwidth))
        self.writedxf(tool_num, "20")
        self.writedxf(tool_num, str(yorigin + radius))
# end current section
        self.writedxf(tool_num, "0")
        self.writedxf(tool_num, "SEQEND")
```

So we add the balance of the corner treatments which are decorative (and easily implemented).

Chamfer:

```
    def dxfrectanglechamfer(self, tool_num, xorigin, yorigin, xwidth, yheight, radius):
        self.dxfline(tool_num, xorigin + radius, yorigin, xorigin, yorigin + radius)
        self.dxfline(tool_num, xorigin, yorigin + yheight - radius, xorigin + radius, yorigin + yheight)
        self.dxfline(tool_num, xorigin + xwidth - radius, yorigin + yheight, xorigin + xwidth, yorigin + yheight - radius)
        self.dxfline(tool_num, xorigin + xwidth - radius, yorigin, xorigin + xwidth, yorigin + radius)

        self.dxfline(tool_num, xorigin + radius, yorigin, xorigin + xwidth - radius, yorigin)
        self.dxfline(tool_num, xorigin + xwidth, yorigin + radius, xorigin + xwidth, yorigin + yheight - radius)
        self.dxfline(tool_num, xorigin + xwidth - radius, yorigin + yheight, xorigin + radius, yorigin + yheight)
        self.dxfline(tool_num, xorigin, yorigin + yheight - radius, xorigin, yorigin + radius)
```

Flipped Fillet:

```
    def dxfrectangleflippedfillet(self, tool_num, xorigin, yorigin, xwidth, yheight, radius):
        self.dxfarc(tool_num, xorigin, yorigin, radius,  0, 90)
        self.dxfarc(tool_num, xorigin + xwidth, yorigin, radius, 90, 180)
        self.dxfarc(tool_num, xorigin + xwidth, yorigin + yheight, radius, 180, 270)
```

```
        self.dxfarc(tool_num, xorigin, yorigin + yheight, radius,
            270, 360)

        self.dxfline(tool_num, xorigin + radius, yorigin, xorigin +
             xwidth - radius, yorigin)
        self.dxfline(tool_num, xorigin + xwidth, yorigin + radius,
            xorigin + xwidth, yorigin + yheight - radius)
        self.dxfline(tool_num, xorigin + xwidth - radius, yorigin +
             yheight, xorigin + radius, yorigin + yheight)
        self.dxfline(tool_num, xorigin, yorigin + yheight - radius,
             xorigin, yorigin + radius)
```

Cutting rectangles while writing out their perimeter in the DXF files (so that they may be assigned a matching toolpath in a traditional CAM program upon import) will require the origin coordinates, height and width and depth of the pocket, and the tool # so that the corners may have a radius equal to the tool which is used. Whether a given module is an interior pocket or an outline (interior or exterior) will be determined by the specifics of the module and its usage/positioning, with `outline` being added to those modules which cut perimeter.

A further consideration is that cut orientation as an option should be accounted for if writing out G-code, as well as stepover, and the nature of initial entry (whether ramping in would be implemented, and if so, at what angle). Advanced toolpath strategies such as trochoidal milling could also be implemented.

`cutrectangle` The routine `cutrectangle` cuts the outline of a rectangle creating rounded corners.

```
    def cutrectangle(self, tool_num, bx, by, bz, xwidth, yheight,
        zdepth):
        self.cutline(bx, by, bz)
        self.cutline(bx, by, bz - zdepth)
        self.cutline(bx + xwidth, by, bz - zdepth)
        self.cutline(bx + xwidth, by + yheight, bz - zdepth)
        self.cutline(bx, by + yheight, bz - zdepth)
        self.cutline(bx, by, bz - zdepth)

    def cutrectangledxf(self, tool_num, bx, by, bz, xwidth, yheight
        , zdepth):
        self.cutrectangle(tool_num, bx, by, bz, xwidth, yheight,
            zdepth)
        self.dxfrectangle(tool_num, bx, by, xwidth, yheight, "
            Square")
```

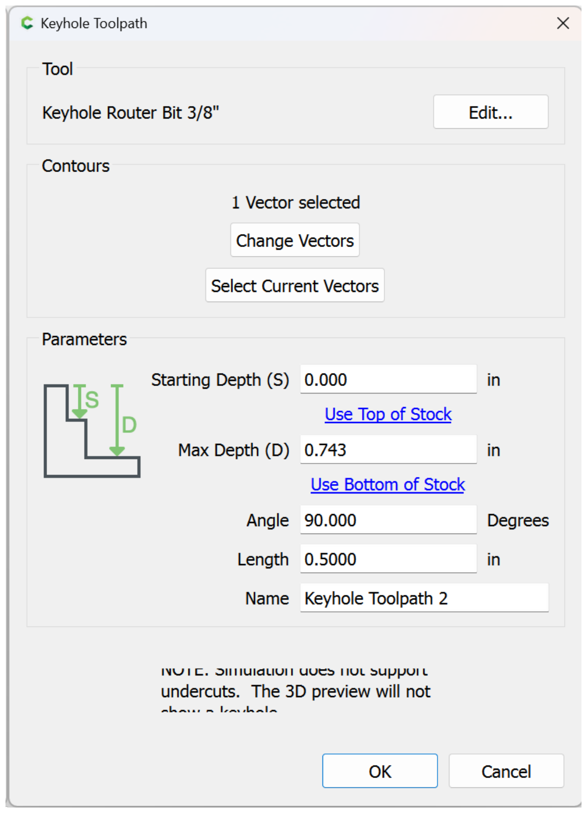
The rounded forms instantiate a radius:

```
    def cutrectangleround(self, tool_num, bx, by, bz, xwidth,
        yheight, zdepth, radius):
#        self.rapid(bx + radius, by, bz)
        self.cutline(bx + radius, by, bz + zdepth)
        self.cutline(bx + xwidth - radius, by, bz + zdepth)
        self.cutquarterCCSE(bx + xwidth, by + radius, bz + zdepth,
            radius)
        self.cutline(bx + xwidth, by + yheight - radius, bz +
            zdepth)
        self.cutquarterCCNE(bx + xwidth - radius, by + yheight, bz
            + zdepth, radius)
        self.cutline(bx + radius, by + yheight, bz + zdepth)
        self.cutquarterCCNW(bx, by + yheight - radius, bz + zdepth,
             radius)
        self.cutline(bx, by + radius, bz + zdepth)
        self.cutquarterCCSW(bx + radius, by, bz + zdepth, radius)

    def cutrectanglerounddxf(self, tool_num, bx, by, bz, xwidth,
        yheight, zdepth, radius):
        self.cutrectangleround(tool_num, bx, by, bz, xwidth,
            yheight, zdepth, radius)
        self.dxfrectangleround(tool_num, bx, by, xwidth, yheight,
            radius)
```

3.7.0.2.3 Keyhole toolpath and undercut tooling The first topologically unusual toolpath is `cutkeyhole toolpath` `cutkeyhole toolpath` — where other toolpaths have a direct correspondence between the associated geometry and the area cut, that Keyhole toolpaths may be used with tooling which undercuts and which will result in the creation of two different physical physical regions: the visible surface matching the union of the tool perimeter at the entry point and the linear movement of the shaft and the larger region of the tool perimeter at the depth which the tool is plunged to and moved along.

Tooling for such toolpaths is defined at paragraph 3.4.0.1

The interface which is being modeled is that of Carbide Create:



Hence the parameters:

- Starting Depth == `kh_start_depth`
- Max Depth == `kh_max_depth`
- Angle == `kht_direction`
- Length == `kh_distance`
- Tool == `kh_tool_num`

Due to the possibility of rotation, for the in-between positions there are more cases than one would think — for each quadrant there are the following possibilities:

- one node on the clockwise side is outside of the quadrant
- two nodes on the clockwise side are outside of the quadrant
- all nodes are w/in the quadrant
- one node on the counter-clockwise side is outside of the quadrant
- two nodes on the counter-clockwise side are outside of the quadrant

Supporting all of these would require trigonometric comparisons in the `if...else` blocks, so only the 4 quadrants, N, S, E, and W will be supported in the initial version. This will be done by wrapping the command with a version which only accepts those options:

```
    def cutkeyholegcdxf(self, kh_tool_num, kh_start_depth,
        kh_max_depth, kht_direction, kh_distance):
        if (kht_direction == "N"):
            toolpath = self.cutKHgcdxf(kh_tool_num, kh_start_depth,
                kh_max_depth, 90, kh_distance)
        elif (kht_direction == "S"):
            toolpath = self.cutKHgcdxf(kh_tool_num, kh_start_depth,
                kh_max_depth, 270, kh_distance)
        elif (kht_direction == "E"):
            toolpath = self.cutKHgcdxf(kh_tool_num, kh_start_depth,
                kh_max_depth, 0, kh_distance)
        elif (kht_direction == "W"):
            toolpath = self.cutKHgcdxf(kh_tool_num, kh_start_depth,
                kh_max_depth, 180, kh_distance)
#          if self.generatepaths == True:
#              self.toolpaths = union([self.toolpaths, toolpath])
        return toolpath
#          else:
#              return cube([0.01, 0.01, 0.01])
```

```
module cutkeyholegcdxf(kh_tool_num, kh_start_depth, kh_max_depth,
    kht_direction, kh_distance){
    gcp.cutkeyholegcdxf(kh_tool_num, kh_start_depth, kh_max_depth,
        kht_direction, kh_distance);
}
```

cutKHgcdxf The original version of the command, `cutKHgcdxf` retains an interface which allows calling it for arbitrary beginning and ending points of an arc.

Note that code is still present for the partial calculation of one quadrant (for the case of all nodes within the quadrant). The first task is to place a circle at the origin which is invariant of angle:

```
    def cutKHgcdxf(self, kh_tool_num, kh_start_depth, kh_max_depth,
        kh_angle, kh_distance):
        oXpos = self.xpos()
        oYpos = self.ypos()
        self.dxfKH(kh_tool_num, self.xpos(), self.ypos(),
            kh_start_depth, kh_max_depth, kh_angle, kh_distance)
        toolpath = self.cutline(self.xpos(), self.ypos(), -
            kh_max_depth)
        self.setxpos(oXpos)
        self.setypos(oYpos)
#         if self.generatepaths == False:
        return toolpath
#         else:
#             return cube([0.001, 0.001, 0.001])
```

```
    def dxfKH(self, kh_tool_num, oXpos, oYpos, kh_start_depth,
        kh_max_depth, kh_angle, kh_distance):
#         oXpos = self.xpos()
#         oYpos = self.ypos()
#Circle at entry hole
        self.dxfarc(kh_tool_num, oXpos, oYpos, self.tool_radius(
            kh_tool_num, 7), 0, 90)
        self.dxfarc(kh_tool_num, oXpos, oYpos, self.tool_radius(
            kh_tool_num, 7), 90, 180)
        self.dxfarc(kh_tool_num, oXpos, oYpos, self.tool_radius(
            kh_tool_num, 7), 180, 270)
        self.dxfarc(kh_tool_num, oXpos, oYpos, self.tool_radius(
            kh_tool_num, 7), 270, 360)
```

Then it will be necessary to test for each possible case in a series of If Else blocks:

```
#pre-calculate needed values
        r = self.tool_radius(kh_tool_num, 7)
#         print(r)
        rt = self.tool_radius(kh_tool_num, 1)
#         print(rt)
        ro = math.sqrt((self.tool_radius(kh_tool_num, 1))**2-(self.
            tool_radius(kh_tool_num, 7))**2)
#         print(ro)
        angle = math.degrees(math.acos(ro/rt))
#Outlines of entry hole and slot
        if (kh_angle == 0):
#Lower left of entry hole
            self.dxfarc(kh_tool_num, self.xpos(), self.ypos(), self
                .tool_radius(kh_tool_num, 1), 180, 270)
#Upper left of entry hole
            self.dxfarc(kh_tool_num, self.xpos(), self.ypos(), self
                .tool_radius(kh_tool_num, 1), 90, 180)
#Upper right of entry hole
#             self.dxfarc(kh_tool_num, self.xpos(), self.ypos(), rt,
    41.810, 90)
            self.dxfarc(kh_tool_num, self.xpos(), self.ypos(), rt,
                angle, 90)
#Lower right of entry hole
            self.dxfarc(kh_tool_num, self.xpos(), self.ypos(), rt,
                270, 360-angle)
#             self.dxfarc(kh_tool_num, self.xpos(), self.ypos(),
    self.tool_radius(kh_tool_num, 1), 270, 270+math.acos(math.
    radians(self.tool_diameter(kh_tool_num, 5)/self.tool_diameter(
    kh_tool_num, 1))))
#Actual line of cut
#             self.dxfline(kh_tool_num, self.xpos(), self.ypos(),
    self.xpos()+kh_distance, self.ypos())
```

```
#upper right of end of slot (kh_max_depth+4.36))/2
            self.dxfarc(kh_tool_num, self.xpos()+kh_distance, self.ypos(), self.tool_diameter(kh_tool_num, (kh_max_depth+4.36))/2, 0, 90)
#lower right of end of slot
            self.dxfarc(kh_tool_num, self.xpos()+kh_distance, self.ypos(), self.tool_diameter(kh_tool_num, (kh_max_depth+4.36))/2, 270, 360)
#upper right slot
            self.dxfline(kh_tool_num, self.xpos()+ro, self.ypos()-(self.tool_diameter(kh_tool_num, 7)/2), self.xpos()+kh_distance, self.ypos()-(self.tool_diameter(kh_tool_num, 7)/2))
#            self.dxfline(kh_tool_num, self.xpos()+(math.sqrt((self.tool_diameter(kh_tool_num, 1)^2)-(self.tool_diameter(kh_tool_num, 5)^2))/2), self.ypos()+self.tool_diameter(kh_tool_num, (kh_max_depth))/2, ( (kh_max_depth-6.34))/2)^2-(self.tool_diameter(kh_tool_num, (kh_max_depth-6.34))/2)^2, self.xpos()+kh_distance, self.ypos()+self.tool_diameter(kh_tool_num, (kh_max_depth))/2, kh_tool_num)
#end position at top of slot
#lower right slot
            self.dxfline(kh_tool_num, self.xpos()+ro, self.ypos()+(self.tool_diameter(kh_tool_num, 7)/2), self.xpos()+kh_distance, self.ypos()+(self.tool_diameter(kh_tool_num, 7)/2))
#        dxfline(kh_tool_num, self.xpos()+(math.sqrt((self.tool_diameter(kh_tool_num, 1)^2)-(self.tool_diameter(kh_tool_num, 5)^2))/2), self.ypos()-self.tool_diameter(kh_tool_num, (kh_max_depth))/2, ( (kh_max_depth-6.34))/2)^2-(self.tool_diameter(kh_tool_num, (kh_max_depth-6.34))/2)^2, self.xpos()+kh_distance, self.ypos()-self.tool_diameter(kh_tool_num, (kh_max_depth))/2, KH_tool_num)
#end position at top of slot
#    hull(){
#      translate([xpos(), ypos(), zpos()]){
#        keyhole_shaft(6.35, 9.525);
#      }
#      translate([xpos(), ypos(), zpos()-kh_max_depth]){
#        keyhole_shaft(6.35, 9.525);
#      }
#    }
#    hull(){
#      translate([xpos(), ypos(), zpos()-kh_max_depth]){
#        keyhole_shaft(6.35, 9.525);
#      }
#      translate([xpos()+kh_distance, ypos(), zpos()-kh_max_depth]){
#        keyhole_shaft(6.35, 9.525);
#      }
#    }
#    cutwithfeed(getxpos(), getypos(), -kh_max_depth, feed);
#    cutwithfeed(getxpos()+kh_distance, getypos(), -kh_max_depth, feed);
#    setxpos(getxpos()-kh_distance);
#  } else if (kh_angle > 0 && kh_angle < 90) {
#//echo(kh_angle);
#  dxfarc(getxpos(), getypos(), tool_diameter(KH_tool_num, (kh_max_depth))/2, 90+kh_angle, 180+kh_angle, KH_tool_num);
#  dxfarc(getxpos(), getypos(), tool_diameter(KH_tool_num, (kh_max_depth))/2, 180+kh_angle, 270+kh_angle, KH_tool_num);
#dxfarc(getxpos(), getypos(), tool_diameter(KH_tool_num, (kh_max_depth))/2, kh_angle+asin((tool_diameter(KH_tool_num, (kh_max_depth+4.36))/2)/(tool_diameter(KH_tool_num, (kh_max_depth))/2)), 90+kh_angle, KH_tool_num);
#dxfarc(getxpos(), getypos(), tool_diameter(KH_tool_num, (kh_max_depth))/2, 270+kh_angle, 360+kh_angle-asin((tool_diameter(KH_tool_num, (kh_max_depth+4.36))/2)/(tool_diameter(KH_tool_num, (kh_max_depth))/2)), KH_tool_num);
#dxfarc(getxpos()+(kh_distance*cos(kh_angle)),
#  getypos()+(kh_distance*sin(kh_angle)), tool_diameter(KH_tool_num, (kh_max_depth+4.36))/2, 0+kh_angle, 90+kh_angle, KH_tool_num);
#dxfarc(getxpos()+(kh_distance*cos(kh_angle)), getypos()+(kh_distance*sin(kh_angle)), tool_diameter(KH_tool_num, (kh_max_depth+4.36))/2, 270+kh_angle, 360+kh_angle, KH_tool_num);
#dxfline( getxpos()+tool_diameter(KH_tool_num, (kh_max_depth))/2*cos(kh_angle+asin((tool_diameter(KH_tool_num, (kh_max_depth+4.36))/2)/(tool_diameter(KH_tool_num, (kh_max_depth))/2))),
```

```
# getypos()+tool_diameter(KH_tool_num, (kh_max_depth))/2*sin(kh_angle+asin((tool_diameter(KH_tool_num, (kh_max_depth+4.36))/2)/(tool_diameter(KH_tool_num, (kh_max_depth))/2))),
# getxpos()+(kh_distance*cos(kh_angle))-((tool_diameter(KH_tool_num, (kh_max_depth+4.36))/2)*sin(kh_angle)),
# getypos()+(kh_distance*sin(kh_angle))+((tool_diameter(KH_tool_num, (kh_max_depth+4.36))/2)*cos(kh_angle)), KH_tool_num);
#//echo("a", tool_diameter(KH_tool_num, (kh_max_depth+4.36))/2);
#//echo("c", tool_diameter(KH_tool_num, (kh_max_depth))/2);
#echo("Aangle", asin((tool_diameter(KH_tool_num, (kh_max_depth+4.36))/2)/(tool_diameter(KH_tool_num, (kh_max_depth))/2)));
#//echo(kh_angle);
# cutwithfeed(getxpos()+(kh_distance*cos(kh_angle)), getypos()+(kh_distance*sin(kh_angle)), -kh_max_depth, feed);
#            toolpath = toolpath.union(self.cutline(self.xpos()+kh_distance, self.ypos(), -kh_max_depth))
        elif (kh_angle == 90):
#Lower left of entry hole
            self.dxfarc(kh_tool_num, oXpos, oYpos, self.tool_radius(kh_tool_num, 1), 180, 270)
#Lower right of entry hole
            self.dxfarc(kh_tool_num, oXpos, oYpos, self.tool_radius(kh_tool_num, 1), 270, 360)
#left slot
            self.dxfline(kh_tool_num, oXpos-r, oYpos+ro, oXpos-r, oYpos+kh_distance)
#right slot
            self.dxfline(kh_tool_num, oXpos+r, oYpos+ro, oXpos+r, oYpos+kh_distance)
#upper left of end of slot
            self.dxfarc(kh_tool_num, oXpos, oYpos+kh_distance, r, 90, 180)
#upper right of end of slot
            self.dxfarc(kh_tool_num, oXpos, oYpos+kh_distance, r, 0, 90)
#Upper right of entry hole
            self.dxfarc(kh_tool_num, oXpos, oYpos, rt, 0, 90-angle)
#Upper left of entry hole
            self.dxfarc(kh_tool_num, oXpos, oYpos, rt, 90+angle, 180)
#            toolpath = toolpath.union(self.cutline(oXpos, oYpos+kh_distance, -kh_max_depth))
        elif (kh_angle == 180):
#Lower right of entry hole
            self.dxfarc(kh_tool_num, oXpos, oYpos, self.tool_radius(kh_tool_num, 1), 270, 360)
#Upper right of entry hole
            self.dxfarc(kh_tool_num, oXpos, oYpos, self.tool_radius(kh_tool_num, 1), 0, 90)
#Upper left of entry hole
            self.dxfarc(kh_tool_num, oXpos, oYpos, rt, 90, 180-angle)
#Lower left of entry hole
            self.dxfarc(kh_tool_num, oXpos, oYpos, rt, 180+angle, 270)
#upper slot
            self.dxfline(kh_tool_num, oXpos-ro, oYpos-r, oXpos-kh_distance, oYpos-r)
#lower slot
            self.dxfline(kh_tool_num, oXpos-ro, oYpos+r, oXpos-kh_distance, oYpos+r)
#upper left of end of slot
            self.dxfarc(kh_tool_num, oXpos-kh_distance, oYpos, r, 90, 180)
#lower left of end of slot
            self.dxfarc(kh_tool_num, oXpos-kh_distance, oYpos, r, 180, 270)
#            toolpath = toolpath.union(self.cutline(oXpos-kh_distance, oYpos, -kh_max_depth))
        elif (kh_angle == 270):
#Upper left of entry hole
            self.dxfarc(kh_tool_num, oXpos, oYpos, self.tool_radius(kh_tool_num, 1), 90, 180)
#Upper right of entry hole
            self.dxfarc(kh_tool_num, oXpos, oYpos, self.tool_radius(kh_tool_num, 1), 0, 90)
#left slot
            self.dxfline(kh_tool_num, oXpos-r, oYpos-ro, oXpos-r,
```

```
                oYpos-kh_distance)
#right slot
            self.dxfline(kh_tool_num, oXpos+r, oYpos-ro, oXpos+r, oYpos-kh_distance)
#lower left of end of slot
            self.dxfarc(kh_tool_num, oXpos, oYpos-kh_distance, r, 180, 270)
#lower right of end of slot
            self.dxfarc(kh_tool_num, oXpos, oYpos-kh_distance, r, 270, 360)
#lower right of entry hole
            self.dxfarc(kh_tool_num, oXpos, oYpos, rt, 180, 270-angle)
#lower left of entry hole
            self.dxfarc(kh_tool_num, oXpos, oYpos, rt, 270+angle, 360)
#            toolpath = toolpath.union(self.cutline(oXpos, oYpos-kh_distance, -kh_max_depth))
#        print(self.zpos())
#        self.setxpos(oXpos)
#        self.setypos(oYpos)
#        if self.generatepaths == False:
#            return toolpath

#  } else if (kh_angle == 90) {
#    //Lower left of entry hole
#    dxfarc(getxpos(), getypos(), 9.525/2, 180, 270, KH_tool_num);
#    //Lower right of entry hole
#    dxfarc(getxpos(), getypos(), 9.525/2, 270, 360, KH_tool_num);
#    //Upper right of entry hole
#    dxfarc(getxpos(), getypos(), 9.525/2, 0, acos(tool_diameter(KH_tool_num, 5)/tool_diameter(KH_tool_num, 1)), KH_tool_num);
#    //Upper left of entry hole
#    dxfarc(getxpos(), getypos(), 9.525/2, 180-acos(tool_diameter(KH_tool_num, 5)/tool_diameter(KH_tool_num, 1)), 180, KH_tool_num);
#    //Actual line of cut
#    dxfline(getxpos(), getypos(), getxpos(), getypos()+kh_distance);
#    //upper right of slot
#    dxfarc(getxpos(), getypos()+kh_distance, tool_diameter(KH_tool_num, (kh_max_depth+4.36))/2, 0, 90, KH_tool_num);
#    //upper left of slot
#    dxfarc(getxpos(), getypos()+kh_distance, tool_diameter(KH_tool_num, (kh_max_depth+6.35))/2, 90, 180, KH_tool_num);
#    //right of slot
#    dxfline(
#        getxpos()+tool_diameter(KH_tool_num, (kh_max_depth))/2,
#        getypos()+(math.sqrt((tool_diameter(KH_tool_num, 1)^2)-(tool_diameter(KH_tool_num, 5)^2))/2), //( (kh_max_depth-6.34))/2)^2-(tool_diameter(KH_tool_num, (kh_max_depth-6.34))/2)^2,
#        getxpos()+tool_diameter(KH_tool_num, (kh_max_depth))/2,
#    //end position at top of slot
#        getypos()+kh_distance,
#        KH_tool_num);
#    dxfline(getxpos()-tool_diameter(KH_tool_num, (kh_max_depth))/2, getypos()+(math.sqrt((tool_diameter(KH_tool_num, 1)^2)-(tool_diameter(KH_tool_num, 5)^2))/2), getxpos()-tool_diameter(KH_tool_num, (kh_max_depth+6.35))/2, getypos()+kh_distance, KH_tool_num);
#    hull(){
#      translate([xpos(), ypos(), zpos()]){
#        keyhole_shaft(6.35, 9.525);
#      }
#      translate([xpos(), ypos(), zpos()-kh_max_depth]){
#        keyhole_shaft(6.35, 9.525);
#      }
#    }
#    hull(){
#      translate([xpos(), ypos(), zpos()-kh_max_depth]){
#        keyhole_shaft(6.35, 9.525);
#      }
#      translate([xpos(), ypos()+kh_distance, zpos()-kh_max_depth]){
#        keyhole_shaft(6.35, 9.525);
#      }
#    }
#    cutwithfeed(getxpos(), getypos(), -kh_max_depth, feed);
```

```
gcpy #     cutwithfeed(getxpos(), getypos()+kh_distance, -kh_max_depth, feed);
gcpy #     setypos(getypos()-kh_distance);
gcpy #   } else if (kh_angle == 180) {
gcpy #     //Lower right of entry hole
gcpy #     dxfarc(getxpos(), getypos(), 9.525/2, 270, 360, KH_tool_num);
gcpy #     //Upper right of entry hole
gcpy #     dxfarc(getxpos(), getypos(), 9.525/2, 0, 90, KH_tool_num);
gcpy #     //Upper left of entry hole
gcpy #     dxfarc(getxpos(), getypos(), 9.525/2, 90, 90+acos(tool_diameter(KH_tool_num, 5)/tool_diameter(KH_tool_num, 1)), KH_tool_num);
gcpy #     //Lower left of entry hole
gcpy #     dxfarc(getxpos(), getypos(), 9.525/2, 270-acos(tool_diameter(KH_tool_num, 5)/tool_diameter(KH_tool_num, 1)), 270, KH_tool_num);
gcpy #     //upper left of slot
gcpy #     dxfarc(getxpos()-kh_distance, getypos(), tool_diameter(KH_tool_num, (kh_max_depth+6.35))/2, 90, 180, KH_tool_num);
gcpy #     //lower left of slot
gcpy #     dxfarc(getxpos()-kh_distance, getypos(), tool_diameter(KH_tool_num, (kh_max_depth+6.35))/2, 180, 270, KH_tool_num);
gcpy #     //Actual line of cut
gcpy #     dxfline(getxpos(), getypos(), getxpos()-kh_distance, getypos());
gcpy #     //upper left slot
gcpy #     dxfline(
gcpy #         getxpos()-(math.sqrt((tool_diameter(KH_tool_num, 1)^2)-(tool_diameter(KH_tool_num, 5)^2))/2),
gcpy #         getypos()+tool_diameter(KH_tool_num, (kh_max_depth))/2, //( (kh_max_depth-6.34))/2)^2-(tool_diameter(KH_tool_num, (kh_max_depth-6.34))/2)^2,
gcpy #         getxpos()-kh_distance,
gcpy #     //end position at top of slot
gcpy #         getypos()+tool_diameter(KH_tool_num, (kh_max_depth))/2,
gcpy #         KH_tool_num);
gcpy #     //lower right slot
gcpy #     dxfline(
gcpy #         getxpos()-(math.sqrt((tool_diameter(KH_tool_num, 1)^2)-(tool_diameter(KH_tool_num, 5)^2))/2),
gcpy #         getypos()-tool_diameter(KH_tool_num, (kh_max_depth))/2, //( (kh_max_depth-6.34))/2)^2-(tool_diameter(KH_tool_num, (kh_max_depth-6.34))/2)^2,
gcpy #         getxpos()-kh_distance,
gcpy #     //end position at top of slot
gcpy #         getypos()-tool_diameter(KH_tool_num, (kh_max_depth))/2,
gcpy #         KH_tool_num);
gcpy #     hull(){
gcpy #       translate([xpos(), ypos(), zpos()]){
gcpy #         keyhole_shaft(6.35, 9.525);
gcpy #       }
gcpy #       translate([xpos(), ypos(), zpos()-kh_max_depth]){
gcpy #         keyhole_shaft(6.35, 9.525);
gcpy #       }
gcpy #     }
gcpy #     hull(){
gcpy #       translate([xpos(), ypos(), zpos()-kh_max_depth]){
gcpy #         keyhole_shaft(6.35, 9.525);
gcpy #       }
gcpy #       translate([xpos()-kh_distance, ypos(), zpos()-kh_max_depth]){
gcpy #         keyhole_shaft(6.35, 9.525);
gcpy #       }
gcpy #     }
gcpy #     cutwithfeed(getxpos(), getypos(), -kh_max_depth, feed);
gcpy #     cutwithfeed(getxpos()-kh_distance, getypos(), -kh_max_depth, feed);
gcpy #     setxpos(getxpos()+kh_distance);
gcpy #   } else if (kh_angle == 270) {
gcpy #     //Upper right of entry hole
gcpy #     dxfarc(getxpos(), getypos(), 9.525/2, 0, 90, KH_tool_num);
gcpy #     //Upper left of entry hole
gcpy #     dxfarc(getxpos(), getypos(), 9.525/2, 90, 180, KH_tool_num);
gcpy #     //lower right of slot
gcpy #     dxfarc(getxpos(), getypos()-kh_distance, tool_diameter(KH_tool_num, (kh_max_depth+4.36))/2, 270, 360, KH_tool_num);
gcpy #     //lower left of slot
gcpy #     dxfarc(getxpos(), getypos()-kh_distance, tool_diameter(
```

```
KH_tool_num, (kh_max_depth+4.36))/2, 180, 270, KH_tool_num);
#    //Actual line of cut
#    dxfline(getxpos(), getypos(), getxpos(), getypos()-kh_distance);
#    //right of slot
#    dxfline(
#        getxpos()+tool_diameter(KH_tool_num, (kh_max_depth))/2,
#        getypos()-(math.sqrt((tool_diameter(KH_tool_num, 1)^2)-(tool_diameter(KH_tool_num, 5)^2))/2), //( (kh_max_depth-6.34))/2)^2-(tool_diameter(KH_tool_num, (kh_max_depth-6.34))/2)^2,
#        getxpos()+tool_diameter(KH_tool_num, (kh_max_depth))/2,
#    //end position at top of slot
#        getypos()-kh_distance,
#        KH_tool_num);
#    //left of slot
#    dxfline(
#        getxpos()-tool_diameter(KH_tool_num, (kh_max_depth))/2,
#        getypos()-(math.sqrt((tool_diameter(KH_tool_num, 1)^2)-(tool_diameter(KH_tool_num, 5)^2))/2), //( (kh_max_depth-6.34))/2)^2-(tool_diameter(KH_tool_num, (kh_max_depth-6.34))/2)^2,
#        getxpos()-tool_diameter(KH_tool_num, (kh_max_depth))/2,
#    //end position at top of slot
#        getypos()-kh_distance,
#        KH_tool_num);
#    //Lower right of entry hole
#    dxfarc(getxpos(), getypos(), 9.525/2, 360-acos(tool_diameter(KH_tool_num, 5)/tool_diameter(KH_tool_num, 1)), 360, KH_tool_num);
#    //Lower left of entry hole
#    dxfarc(getxpos(), getypos(), 9.525/2, 180, 180+acos(tool_diameter(KH_tool_num, 5)/tool_diameter(KH_tool_num, 1)), KH_tool_num);
#    hull(){
#      translate([xpos(), ypos(), zpos()]){
#        keyhole_shaft(6.35, 9.525);
#      }
#      translate([xpos(), ypos(), zpos()-kh_max_depth]){
#        keyhole_shaft(6.35, 9.525);
#      }
#    }
#    hull(){
#      translate([xpos(), ypos(), zpos()-kh_max_depth]){
#        keyhole_shaft(6.35, 9.525);
#      }
#      translate([xpos(), ypos()-kh_distance, zpos()-kh_max_depth]){
#        keyhole_shaft(6.35, 9.525);
#      }
#    }
#    cutwithfeed(getxpos(), getypos(), -kh_max_depth, feed);
#    cutwithfeed(getxpos(), getypos()-kh_distance, -kh_max_depth, feed);
#    setypos(getypos()+kh_distance);
#  }
#}
```

3.7.0.2.4 Dovetail joinery and tooling One focus of this project from the beginning has been cutting joinery. The first such toolpath to be developed is half-blind dovetails, since they are intrinsically simple to calculate since their geometry is dictated by the geometry of the tool.

BlocksCAD project page at: https://www.blockscad3d.com/community/projects/1941456 and discussion at: https://community.carbide3d.com/t/tool-paths-for-different-sized-dovetail-bit 89098

Making such cuts will require dovetail tooling such as:

- 808079 https://www.amanatool.com/45828-carbide-tipped-dovetail-8-deg-x-1-2-dia-x-825-x-1 html
- 814 https://www.leevalley.com/en-us/shop/tools/power-tool-accessories/router-bits/30172-dovetail-bits?item=18J1607

Two commands are required:

```
    def cut_pins(self, Joint_Width, stockZthickness, Number_of_Dovetails, Spacing, Proportion, DTT_diameter, DTT_angle):
        DTO = Tan(math.radians(DTT_angle)) * (stockZthickness * Proportion)
```

```
        DTR = DTT_diameter/2 - DTO
        cpr = self.rapidXY(0, stockZthickness + Spacing/2)
        ctp = self.cutlinedxfgc(self.xpos(), self.ypos(), -stockZthickness * Proportion)
#        ctp = ctp.union(self.cutlinedxfgc(Joint_Width / (Number_of_Dovetails * 2), self.ypos(), -stockZthickness * Proportion))
        i = 1
        while i < Number_of_Dovetails * 2:
#            print(i)
            ctp = ctp.union(self.cutlinedxfgc(i * (Joint_Width / (Number_of_Dovetails * 2)), self.ypos(), -stockZthickness * Proportion))
            ctp = ctp.union(self.cutlinedxfgc(i * (Joint_Width / (Number_of_Dovetails * 2)), (stockZthickness + Spacing) + (stockZthickness * Proportion) - (DTT_diameter/2), -(stockZthickness * Proportion)))
            ctp = ctp.union(self.cutlinedxfgc(i * (Joint_Width / (Number_of_Dovetails * 2)), stockZthickness + Spacing/2, -(stockZthickness * Proportion)))
            ctp = ctp.union(self.cutlinedxfgc((i + 1) * (Joint_Width / (Number_of_Dovetails * 2)), stockZthickness + Spacing/2,-(stockZthickness * Proportion)))
            self.dxfrectangleround(self.currenttoolnumber(),
                i * (Joint_Width / (Number_of_Dovetails * 2))-DTR,
                stockZthickness + (Spacing/2) - DTR,
                DTR * 2,
                (stockZthickness * Proportion) + Spacing/2 + DTR * 2 - (DTT_diameter/2),
                DTR)
            i += 2
        self.rapidZ(0)
        return ctp
```

and

```
    def cut_tails(self, Joint_Width, stockZthickness, Number_of_Dovetails, Spacing, Proportion, DTT_diameter, DTT_angle):
        DTO = Tan(math.radians(DTT_angle)) * (stockZthickness * Proportion)
        DTR = DTT_diameter/2 - DTO
        cpr = self.rapidXY(0, 0)
        ctp = self.cutlinedxfgc(self.xpos(), self.ypos(), -stockZthickness * Proportion)
        ctp = ctp.union(self.cutlinedxfgc(
            Joint_Width / (Number_of_Dovetails * 2) - (DTT_diameter - DTO),
            self.ypos(),
            -stockZthickness * Proportion))
        i = 1
        while i < Number_of_Dovetails * 2:
            ctp = ctp.union(self.cutlinedxfgc(
                i * (Joint_Width / (Number_of_Dovetails * 2)) - (DTT_diameter - DTO),
                stockZthickness * Proportion - DTT_diameter / 2,
                -(stockZthickness * Proportion)))
            ctp = ctp.union(self.cutarcCWdxf(180, 90,
                i * (Joint_Width / (Number_of_Dovetails * 2)),
                stockZthickness * Proportion - DTT_diameter / 2,
#                self.ypos(),
                DTT_diameter - DTO, 0, 1))
            ctp = ctp.union(self.cutarcCWdxf(90, 0,
                i * (Joint_Width / (Number_of_Dovetails * 2)),
                stockZthickness * Proportion - DTT_diameter / 2,
                DTT_diameter - DTO, 0, 1))
            ctp = ctp.union(self.cutlinedxfgc(
                i * (Joint_Width / (Number_of_Dovetails * 2)) + (DTT_diameter - DTO),
                0,
                -(stockZthickness * Proportion)))
            ctp = ctp.union(self.cutlinedxfgc(
                (i + 2) * (Joint_Width / (Number_of_Dovetails * 2)) - (DTT_diameter - DTO),
                0,
                -(stockZthickness * Proportion)))
            i += 2
```

```
        self.rapidZ(0)
        self.rapidXY(0, 0)
        ctp = ctp.union(self.cutlinedxfgc(self.xpos(), self.ypos(), -stockZthickness * Proportion))
        self.dxfarc(self.currenttoolnumber(), 0, 0, DTR, 180, 270)
        self.dxfline(self.currenttoolnumber(), -DTR, 0, -DTR, stockZthickness + DTR)
        self.dxfarc(self.currenttoolnumber(), 0, stockZthickness + DTR, DTR, 90, 180)
        self.dxfline(self.currenttoolnumber(), 0, stockZthickness + DTR * 2, Joint_Width, stockZthickness + DTR * 2)
        i = 0
        while i < Number_of_Dovetails * 2:
            ctp = ctp.union(self.cutline(i * (Joint_Width / (Number_of_Dovetails * 2)), stockZthickness + DTO, -(stockZthickness * Proportion)))
            ctp = ctp.union(self.cutline((i+2) * (Joint_Width / (Number_of_Dovetails * 2)), stockZthickness + DTO, -(stockZthickness * Proportion)))
            ctp = ctp.union(self.cutline((i+2) * (Joint_Width / (Number_of_Dovetails * 2)), 0, -(stockZthickness * Proportion)))
            self.dxfarc(self.currenttoolnumber(), i * (Joint_Width / (Number_of_Dovetails * 2)), 0, DTR, 270, 360)
            self.dxfline(self.currenttoolnumber(),
                i * (Joint_Width / (Number_of_Dovetails * 2)) + DTR,
                0,
                i * (Joint_Width / (Number_of_Dovetails * 2)) + DTR, stockZthickness * Proportion - DTT_diameter / 2)
            self.dxfarc(self.currenttoolnumber(), (i + 1) * (Joint_Width / (Number_of_Dovetails * 2)), stockZthickness * Proportion - DTT_diameter / 2, (Joint_Width / (Number_of_Dovetails * 2)) - DTR, 90, 180)
            self.dxfarc(self.currenttoolnumber(), (i + 1) * (Joint_Width / (Number_of_Dovetails * 2)), stockZthickness * Proportion - DTT_diameter / 2, (Joint_Width / (Number_of_Dovetails * 2)) - DTR, 0, 90)
            self.dxfline(self.currenttoolnumber(),
                (i + 2) * (Joint_Width / (Number_of_Dovetails * 2)) - DTR,
                0,
                (i + 2) * (Joint_Width / (Number_of_Dovetails * 2)) - DTR, stockZthickness * Proportion - DTT_diameter / 2)
            self.dxfarc(self.currenttoolnumber(), (i + 2) * (Joint_Width / (Number_of_Dovetails * 2)), 0, DTR, 180, 270)
            i += 2
        self.dxfarc(self.currenttoolnumber(), Joint_Width, stockZthickness + DTR, DTR, 0, 90)
        self.dxfline(self.currenttoolnumber(), Joint_Width + DTR, stockZthickness + DTR, Joint_Width + DTR, 0)
        self.dxfarc(self.currenttoolnumber(), Joint_Width, 0, DTR, 270, 360)
        return ctp
```

which are used as:

```
toolpaths = gcp.cut_pins(stockXwidth, stockZthickness, Number_of_Dovetails, Spacing, Proportion, DTT_dia
```

```
toolpaths.append(gcp.cut_tails(stockXwidth, stockZthickness, Number_of_Dovetails, Spacing, Proportion, I
```

Future versions may adjust the parameters passed in, having them calculate from the specifications for the currently active dovetail tool.

3.7.0.2.5 Full-blind box joints

BlocksCAD project page at: https://www.blockscad3d.com/community/projects/1943966 and discussion at: https://community.carbide3d.com/t/full-blind-box-joints-in-carbide-create/53329

Full-blind box joints will require 3 separate tools:

- small V tool — this will be needed to make a cut along the edge of the joint
- small square tool — this should be the same diameter as the small V tool

- large V tool — this will facilitate the stock being of a greater thickness and avoid the need to make multiple cuts to cut the blind miters at the ends of the joint

Two different versions of the commands will be necessary, one for each orientation:

- horizontal
- vertical

and then the internal commands for each side will in turn need separate versions:

```
    def Full_Blind_Finger_Joint_square(self, bx, by, orientation, side, width, thickness, Number_of_Pins, largeVdiameter, smallDiameter, normalormirror = "Default"):
    # Joint_Orientation = "Horizontal" "Even" == "Lower", "Odd" == "Upper"
    # Joint_Orientation = "Vertical" "Even" == "Left", "Odd" == "Right"
        if (orientation == "Vertical"):
            if (normalormirror == "Default" and side != "Both"):
                if (side == "Left"):
                    normalormirror = "Even"
                if (side == "Right"):
                    normalormirror = "Odd"
        if (orientation == "Horizontal"):
            if (normalormirror == "Default" and side != "Both"):
                if (side == "Lower"):
                    normalormirror = "Even"
                if (side == "Upper"):
                    normalormirror = "Odd"
        Finger_Width = ((Number_of_Pins * 2) - 1) * smallDiameter * 1.1
        Finger_Origin = width/2 - Finger_Width/2
        rapid = self.rapidZ(0)
        self.setdxfcolor("Cyan")
        rapid = rapid.union(self.rapidXY(bx, by))
        toolpath = (self.Finger_Joint_square(bx, by, orientation, side, width, thickness, Number_of_Pins, Finger_Origin, smallDiameter))
        if (orientation == "Vertical"):
            if (side == "Both"):
                toolpath = self.cutrectanglerounddxf(self.currenttoolnum, bx - (thickness - smallDiameter/2), by-smallDiameter/2, 0, (thickness * 2) - smallDiameter, width+smallDiameter, (smallDiameter / 2) / Tan(math.radians(45)), smallDiameter/2)
            if (side == "Left"):
                toolpath = self.cutrectanglerounddxf(self.currenttoolnum, bx - (smallDiameter/2), by-smallDiameter/2, 0, thickness, width+smallDiameter, ((smallDiameter / 2) / Tan(math.radians(45))), smallDiameter/2)
            if (side == "Right"):
                toolpath = self.cutrectanglerounddxf(self.currenttoolnum, bx - (thickness - smallDiameter/2), by-smallDiameter/2, 0, thickness, width+smallDiameter, ((smallDiameter / 2) / Tan(math.radians(45))), smallDiameter/2)
        toolpath = toolpath.union(self.Finger_Joint_square(bx, by, orientation, side, width, thickness, Number_of_Pins, Finger_Origin, smallDiameter))
        if (orientation == "Horizontal"):
            if (side == "Both"):
                toolpath = self.cutrectanglerounddxf(
                    self.currenttoolnum,
                    bx-smallDiameter/2,
                    by - (thickness - smallDiameter/2),
                    0,
                    width+smallDiameter,
                    (thickness * 2) - smallDiameter,
                    (smallDiameter / 2) / Tan(math.radians(45)),
                    smallDiameter/2)
            if (side == "Lower"):
                toolpath = self.cutrectanglerounddxf(
                    self.currenttoolnum,
                    bx - (smallDiameter/2),
                    by - smallDiameter/2,
                    0,
```

```
                    width+smallDiameter,
                    thickness,
                    ((smallDiameter / 2) / Tan(math.radians(45))),
                    smallDiameter/2)
            if (side == "Upper"):
                toolpath = self.cutrectangleroundddxf(
                    self.currenttoolnum,
                    bx - smallDiameter/2,
                    by - (thickness - smallDiameter/2),
                    0,
                    width+smallDiameter,
                    thickness,
                    ((smallDiameter / 2) / Tan(math.radians(45))),
                    smallDiameter/2)
        toolpath = toolpath.union(self.Finger_Joint_square(bx, by, orientation, side, width, thickness, Number_of_Pins, Finger_Origin, smallDiameter))
        return toolpath

    def Finger_Joint_square(self, bx, by, orientation, side, width, thickness, Number_of_Pins, Finger_Origin, smallDiameter, normalormirror = "Default"):
        jointdepth = -(thickness - (smallDiameter / 2) / Tan(math.radians(45)))
    # Joint_Orientation = "Horizontal" "Even" == "Lower", "Odd" == "Upper"
    # Joint_Orientation = "Vertical" "Even" == "Left", "Odd" == "Right"
        if (orientation == "Vertical"):
            if (normalormirror == "Default" and side != "Both"):
                if (side == "Left"):
                     normalormirror = "Even"
                if (side == "Right"):
                     normalormirror = "Odd"
        if (orientation == "Horizontal"):
            if (normalormirror == "Default" and side != "Both"):
                if (side == "Lower"):
                     normalormirror = "Even"
                if (side == "Upper"):
                     normalormirror = "Odd"
        radius = smallDiameter/2
        jointwidth = thickness - smallDiameter
        toolpath = self.currenttool()
        rapid = self.rapidZ(0)
        self.setdxfcolor("Blue")
        toolpath = toolpath.union(self.cutlineZgcfeed(jointdepth,1000))
        self.beginpolyline(self.currenttool())
        if (orientation == "Vertical"):
            rapid = rapid.union(self.rapidXY(bx, by + Finger_Origin))
            self.addvertex(self.currenttoolnumber(), self.xpos(), self.ypos())
            toolpath = toolpath.union(self.cutlineZgcfeed(jointdepth,1000))
            i = 0
            while i <= Number_of_Pins - 1:
                if (side == "Right"):
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos(), self.ypos() + smallDiameter + radius/5, jointdepth))
                if (side == "Left" or side == "Both"):
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos(), self.ypos() + radius, jointdepth))
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos() + jointwidth, self.ypos(), jointdepth))
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos(), self.ypos() + radius/5, jointdepth))
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos() - jointwidth, self.ypos(), jointdepth))
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos(), self.ypos() + radius, jointdepth))
                if (side == "Left"):
```

```
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos(), self.ypos() + smallDiameter + radius/5, jointdepth))
                if (side == "Right" or side == "Both"):
                    if (i < (Number_of_Pins - 1)):
#                        print(i)
                        toolpath = toolpath.union(self.cutvertexdxf(self.xpos(), self.ypos() + radius, jointdepth))
                        toolpath = toolpath.union(self.cutvertexdxf(self.xpos() - jointwidth, self.ypos(), jointdepth))
                        toolpath = toolpath.union(self.cutvertexdxf(self.xpos(), self.ypos() + radius/5, jointdepth))
                        toolpath = toolpath.union(self.cutvertexdxf(self.xpos() + jointwidth, self.ypos(), jointdepth))
                        toolpath = toolpath.union(self.cutvertexdxf(self.xpos(), self.ypos() + radius, jointdepth))
                i += 1
#    Joint_Orientation = "Horizontal" "Even" == "Lower", "Odd" == "Upper"
        if (orientation == "Horizontal"):
            rapid = rapid.union(self.rapidXY(bx + Finger_Origin, by))
            self.addvertex(self.currenttoolnumber(), self.xpos(), self.ypos())
            toolpath = toolpath.union(self.cutlineZgcfeed(jointdepth,1000))
            i = 0
            while i <= Number_of_Pins - 1:
                if (side == "Upper"):
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos() + smallDiameter + radius/5, self.ypos(), jointdepth))
                if (side == "Lower" or side == "Both"):
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos() + radius, self.ypos(), jointdepth))
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos(), self.ypos() + jointwidth, jointdepth))
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos() + radius/5, self.ypos(), jointdepth))
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos(), self.ypos() - jointwidth, jointdepth))
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos() + radius, self.ypos(), jointdepth))
                if (side == "Lower"):
                    toolpath = toolpath.union(self.cutvertexdxf(self.xpos() + smallDiameter + radius/5, self.ypos(), jointdepth))
                if (side == "Upper" or side == "Both"):
                    if (i < (Number_of_Pins - 1)):
#                        print(i)
                        toolpath = toolpath.union(self.cutvertexdxf(self.xpos() + radius, self.ypos(), jointdepth))
                        toolpath = toolpath.union(self.cutvertexdxf(self.xpos(), self.ypos() - jointwidth, jointdepth))
                        toolpath = toolpath.union(self.cutvertexdxf(self.xpos() + radius/5, self.ypos(), jointdepth))
                        toolpath = toolpath.union(self.cutvertexdxf(self.xpos(), self.ypos() + jointwidth, jointdepth))
                        toolpath = toolpath.union(self.cutvertexdxf(self.xpos() + radius, self.ypos(), jointdepth))
                i += 1
        self.closepolyline(self.currenttoolnumber())
        return toolpath
```

```python

    def Full_Blind_Finger_Joint_smallV(self, bx, by, orientation, side, width, thickness, Number_of_Pins, largeVdiameter, smallDiameter):
        rapid = self.rapidZ(0)
#        rapid = rapid.union(self.rapidXY(bx, by))
        self.setdxfcolor("Red")
        if (orientation == "Vertical"):
            rapid = rapid.union(self.rapidXY(bx, by - smallDiameter/6))
            toolpath = self.cutlineZgcfeed(-thickness,1000)
            toolpath = self.cutlinedxfgc(bx, by + width + smallDiameter/6, - thickness)
        if (orientation == "Horizontal"):
            rapid = rapid.union(self.rapidXY(bx - smallDiameter/6, by))
            toolpath = self.cutlineZgcfeed(-thickness,1000)
            toolpath = self.cutlinedxfgc(bx + width + smallDiameter/6, by, -thickness)
#            rapid = self.rapidZ(0)

        return toolpath

    def Full_Blind_Finger_Joint_largeV(self, bx, by, orientation, side, width, thickness, Number_of_Pins, largeVdiameter, smallDiameter):
        radius = smallDiameter/2
        rapid = self.rapidZ(0)
        Finger_Width = ((Number_of_Pins * 2) - 1) * smallDiameter * 1.1
        Finger_Origin = width/2 - Finger_Width/2
#        rapid = rapid.union(self.rapidXY(bx, by))
# Joint_Orientation = "Horizontal" "Even" == "Lower", "Odd" == "Upper"
# Joint_Orientation = "Vertical" "Even" == "Left", "Odd" == "Right"
        if (orientation == "Vertical"):
            rapid = rapid.union(self.rapidXY(bx, by))
            toolpath = self.cutlineZgcfeed(-thickness,1000)
            toolpath = toolpath.union(self.cutlinedxfgc(bx, by + Finger_Origin, -thickness))
            rapid = self.rapidZ(0)
            rapid = rapid.union(self.rapidXY(bx, by + width - Finger_Origin))
            self.setdxfcolor("Blue")
            toolpath = toolpath.union(self.cutlineZgcfeed(-thickness,1000))
            toolpath = toolpath.union(self.cutlinedxfgc(bx, by + width, -thickness))
            if (side == "Left" or side == "Both"):
                rapid = self.rapidZ(0)
                self.setdxfcolor("Dark Gray")
                rapid = rapid.union(self.rapidXY(bx+thickness-(smallDiameter / 2) / Tan(math.radians(45)), by - radius/2))
                toolpath = toolpath.union(self.cutlineZgcfeed(-(smallDiameter / 2) / Tan(math.radians(45)),10000))
                toolpath = toolpath.union(self.cutlinedxfgc(bx+thickness-(smallDiameter / 2) / Tan(math.radians(45)), by + width + radius/2, -(smallDiameter / 2) / Tan(math.radians(45))))
                rapid = self.rapidZ(0)
                self.setdxfcolor("Green")
                rapid = rapid.union(self.rapidXY(bx+thickness/2, by+width))
                toolpath = toolpath.union(self.cutlineZgcfeed(-thickness/2,1000))
                toolpath = toolpath.union(self.cutlinedxfgc(bx+thickness/2, by + width -thickness, -thickness/2))
                rapid = self.rapidZ(0)
                rapid = rapid.union(self.rapidXY(bx+thickness/2, by))
                toolpath = toolpath.union(self.cutlineZgcfeed(-thickness/2,1000))
                toolpath = toolpath.union(self.cutlinedxfgc(bx+thickness/2, by +thickness, -thickness/2))
```

```
            if (side == "Right" or side == "Both"):
                rapid = self.rapidZ(0)
                self.setdxfcolor("Dark␣Gray")
                rapid = rapid.union(self.rapidXY(bx-(thickness-(smallDiameter / 2) / Tan(math.radians(45))), by - radius/2))
                toolpath = toolpath.union(self.cutlineZgcfeed(-(smallDiameter / 2) / Tan(math.radians(45)),10000))
                toolpath = toolpath.union(self.cutlinedxfgc(bx-(thickness-(smallDiameter / 2) / Tan(math.radians(45))), by + width + radius/2, -(smallDiameter / 2) / Tan(math.radians(45))))
                rapid = self.rapidZ(0)
                self.setdxfcolor("Green")
                rapid = rapid.union(self.rapidXY(bx-thickness/2, by+width))
                toolpath = toolpath.union(self.cutlineZgcfeed(-thickness/2,1000))
                toolpath = toolpath.union(self.cutlinedxfgc(bx-thickness/2, by + width -thickness, -thickness/2))
                rapid = self.rapidZ(0)
                rapid = rapid.union(self.rapidXY(bx-thickness/2, by))
                toolpath = toolpath.union(self.cutlineZgcfeed(-thickness/2,1000))
                toolpath = toolpath.union(self.cutlinedxfgc(bx-thickness/2, by +thickness, -thickness/2))
    # Joint_Orientation = "Horizontal" "Even" == "Lower", "Odd" == "Upper"
        if (orientation == "Horizontal"):
            rapid = rapid.union(self.rapidXY(bx, by))
            self.setdxfcolor("Blue")
            toolpath = self.cutlineZgcfeed(-thickness,1000)
            toolpath = toolpath.union(self.cutlinedxfgc(bx + Finger_Origin, by, -thickness))
            rapid = rapid.union(self.rapidZ(0))
            rapid = rapid.union(self.rapidXY(bx + width - Finger_Origin, by))
            toolpath = toolpath.union(self.cutlineZgcfeed(-thickness,1000))
            toolpath = toolpath.union(self.cutlinedxfgc(bx + width, by, -thickness))
            if (side == "Lower" or side == "Both"):
                rapid = self.rapidZ(0)
                self.setdxfcolor("Dark␣Gray")
                rapid = rapid.union(self.rapidXY(bx - radius, by+thickness-(smallDiameter / 2) / Tan(math.radians(45))))
                toolpath = toolpath.union(self.cutlineZgcfeed(-(smallDiameter / 2) / Tan(math.radians(45)),10000))
                toolpath = toolpath.union(self.cutlinedxfgc(bx + width + radius, by+thickness-(smallDiameter / 2) / Tan(math.radians(45)), -(smallDiameter / 2) / Tan(math.radians(45))))
                rapid = self.rapidZ(0)
                self.setdxfcolor("Green")
                rapid = rapid.union(self.rapidXY(bx+width, by+thickness/2))
                toolpath = toolpath.union(self.cutlineZgcfeed(-thickness/2,1000))
                toolpath = toolpath.union(self.cutlinedxfgc(bx + width -thickness, by+thickness/2, -thickness/2))
                rapid = self.rapidZ(0)
                rapid = rapid.union(self.rapidXY(bx, by+thickness/2))
                toolpath = toolpath.union(self.cutlineZgcfeed(-thickness/2,1000))
                toolpath = toolpath.union(self.cutlinedxfgc(bx + thickness, by+thickness/2, -thickness/2))
            if (side == "Upper" or side == "Both"):
                rapid = self.rapidZ(0)
                self.setdxfcolor("Dark␣Gray")
                rapid = rapid.union(self.rapidXY(bx - radius, by-(thickness-(smallDiameter / 2) / Tan(math.radians(45)))))
```

```
                toolpath = toolpath.union(self.cutlineZgcfeed(-(smallDiameter / 2) / Tan(math.radians(45)),10000))
                toolpath = toolpath.union(self.cutlinedxfgc(bx + width + radius, by-(thickness-(smallDiameter / 2) / Tan(math.radians(45))), -(smallDiameter / 2) / Tan(math.radians(45))))
                rapid = self.rapidZ(0)
                self.setdxfcolor("Green")
                rapid = rapid.union(self.rapidXY(bx+width, by-thickness/2))
                toolpath = toolpath.union(self.cutlineZgcfeed(-thickness/2,1000))
                toolpath = toolpath.union(self.cutlinedxfgc(bx + width -thickness, by-thickness/2, -thickness/2))
                rapid = self.rapidZ(0)
                rapid = rapid.union(self.rapidXY(bx, by-thickness/2))
                toolpath = toolpath.union(self.cutlineZgcfeed(-thickness/2,1000))
                toolpath = toolpath.union(self.cutlinedxfgc(bx + thickness, by-thickness/2, -thickness/2))
        rapid = self.rapidZ(0)
        return toolpath

    def Full_Blind_Finger_Joint(self, bx, by, orientation, side, width, thickness, largeVdiameter, smallDiameter, normalormirror = "Default", squaretool = 102, smallV = 390, largeV = 301):
        Number_of_Pins = int(((width - thickness * 2) / (smallDiameter * 2.2) / 2) + 0.0) * 2 + 1
#         print("Number of Pins: ",Number_of_Pins)
        self.movetosafeZ()
        self.toolchange(squaretool, 17000)
        toolpath = self.Full_Blind_Finger_Joint_square(bx, by, orientation, side, width, thickness, Number_of_Pins, largeVdiameter, smallDiameter)
        self.movetosafeZ()
        self.toolchange(smallV, 17000)
        toolpath = toolpath.union(self.Full_Blind_Finger_Joint_smallV(bx, by, orientation, side, width, thickness, Number_of_Pins, largeVdiameter, smallDiameter))
        self.toolchange(largeV, 17000)
        toolpath = toolpath.union(self.Full_Blind_Finger_Joint_largeV(bx, by, orientation, side, width, thickness, Number_of_Pins, largeVdiameter, smallDiameter))
        return toolpath
```

3.8 (Reading) G-code Files

With all other features in place, it becomes possible to read in a G-code file and then create a 3D preview of how it will cut.

First, a template file will be necessary:

```
from openscad import *
#nimport("https://raw.githubusercontent.com/WillAdams/gcodepreview/refs/heads/main/gcodepreview.py")

from gcodepreview import *

gc_file = "filename_of_G-code_file_to_process.nc"

gcp = gcodepreview(False, False)

gcp.previewgcodefile(gc_file)
```

previewgcodefile Which simply needs to call the previewgcodefile command:

```
    def previewgcodefile(self, gc_file):
        gc_file = open(gc_file, 'r')
        gcfilecontents = []
        with gc_file as file:
            for line in file:
                command = line
```

```
                gcfilecontents.append(line)

        numlinesfound = 0
        for line in gcfilecontents:
#            print(line)
            if line[:10] == "(stockMin:":
                subdivisions = line.split()
                extentleft = float(subdivisions[0][10:-3])
                extentfb = float(subdivisions[1][:-3])
                extentd = float(subdivisions[2][:-3])
                numlinesfound = numlinesfound + 1
            if line[:13] == "(STOCK/BLOCK,":
                subdivisions = line.split()
                sizeX = float(subdivisions[0][13:-1])
                sizeY = float(subdivisions[1][:-1])
                sizeZ = float(subdivisions[4][:-1])
                numlinesfound = numlinesfound + 1
            if line[:3] == "G21":
                units = "mm"
                numlinesfound = numlinesfound + 1
            if numlinesfound >=3:
                break
#            print(numlinesfound)

        self.setupcuttingarea(sizeX, sizeY, sizeZ, extentleft, extentfb, extentd)

        commands = []
        for line in gcfilecontents:
            Xc = 0
            Yc = 0
            Zc = 0
            Fc = 0
            Xp = 0.0
            Yp = 0.0
            Zp = 0.0
            if line == "G53G0Z-5.000\n":
                 self.movetosafeZ()
            if line[:3] == "M6T":
                tool = int(line[3:])
                self.toolchange(tool)
            if line[:2] == "G0":
                machinestate = "rapid"
            if line[:2] == "G1":
                machinestate = "cutline"
            if line[:2] == "G0" or line[:2] == "G1" or line[:1] == "X" or line[:1] == "Y" or line[:1] == "Z":
                if "F" in line:
                    Fplus = line.split("F")
                    Fc = 1
                    fr = float(Fplus[1])
                    line = Fplus[0]
                if "Z" in line:
                    Zplus = line.split("Z")
                    Zc = 1
                    Zp = float(Zplus[1])
                    line = Zplus[0]
                if "Y" in line:
                    Yplus = line.split("Y")
                    Yc = 1
                    Yp = float(Yplus[1])
                    line = Yplus[0]
                if "X" in line:
                    Xplus = line.split("X")
                    Xc = 1
                    Xp = float(Xplus[1])
                if Zc == 1:
                    if Yc == 1:
                        if Xc == 1:
                            if machinestate == "rapid":
                                command = "rapidXYZ(" + str(Xp) + ", " + str(Yp) + ", " + str(Zp) + ")"
                                self.rapidXYZ(Xp, Yp, Zp)
                            else:
                                command = "cutlineXYZ(" + str(Xp) + ", " + str(Yp) + ", " + str(Zp) + ")"
```

```
                                self.cutlineXYZ(Xp, Yp, Zp)
                        else:
                            if machinestate == "rapid":
                                command = "rapidYZ(" + str(Yp) + ", " + str(Zp) + ")"
                                self.rapidYZ(Yp, Zp)
                            else:
                                command = "cutlineYZ(" + str(Yp) + ", " + str(Zp) + ")"
                                self.cutlineYZ(Yp, Zp)
                    else:
                        if Xc == 1:
                            if machinestate == "rapid":
                                command = "rapidXZ(" + str(Xp) + ", " + str(Zp) + ")"
                                self.rapidXZ(Xp, Zp)
                            else:
                                command = "cutlineXZ(" + str(Xp) + ", " + str(Zp) + ")"
                                self.cutlineXZ(Xp, Zp)
                        else:
                            if machinestate == "rapid":
                                command = "rapidZ(" + str(Zp) + ")"
                                self.rapidZ(Zp)
                            else:
                                command = "cutlineZ(" + str(Zp) + ")"
                                self.cutlineZ(Zp)
                else:
                    if Yc == 1:
                        if Xc == 1:
                            if machinestate == "rapid":
                                command = "rapidXY(" + str(Xp) + ", " + str(Yp) + ")"
                                self.rapidXY(Xp, Yp)
                            else:
                                command = "cutlineXY(" + str(Xp) + ", " + str(Yp) + ")"
                                self.cutlineXY(Xp, Yp)
                        else:
                            if machinestate == "rapid":
                                command = "rapidY(" + str(Yp) + ")"
                                self.rapidY(Yp)
                            else:
                                command = "cutlineY(" + str(Yp) + ")"
                                self.cutlineY(Yp)
                    else:
                        if Xc == 1:
                            if machinestate == "rapid":
                                command = "rapidX(" + str(Xp) + ")"
                                self.rapidX(Xp)
                            else:
                                command = "cutlineX(" + str(Xp) + ")"
                                self.cutlineX(Xp)
                commands.append(command)
#                print(line)
#                print(command)
#                print(machinestate, Xc, Yc, Zc)
#                print(Xp, Yp, Zp)
#                print("/n")

#        for command in commands:
#            print(command)

#        show(self.stockandtoolpaths())
        self.stockandtoolpaths()
```

4 Notes

4.1 Other Resources

4.1.1 Coding Style

A notable influence on the coding style in this project is John Ousterhout's *A Philosophy of Software Design*[SoftwareDesign]. Complexity is managed by the overall design and structure of the code, structuring it so that each component may be worked with on an individual basis, hiding the maximum information, and exposing the maximum functionality, with names selected so as to express their functionality/usage.

Red Flags to avoid include:

- Shallow Module
- Information Leakage
- Temporal Decomposition
- Overexposure
- Pass-Through Method
- Repetition
- Special-General Mixture
- Conjoined Methods
- Comment Repeats Code
- Implementation Documentation Contaminates Interface
- Vague Name
- Hard to Pick Name
- Hard to Describe
- Nonobvious Code

4.1.2 Coding References

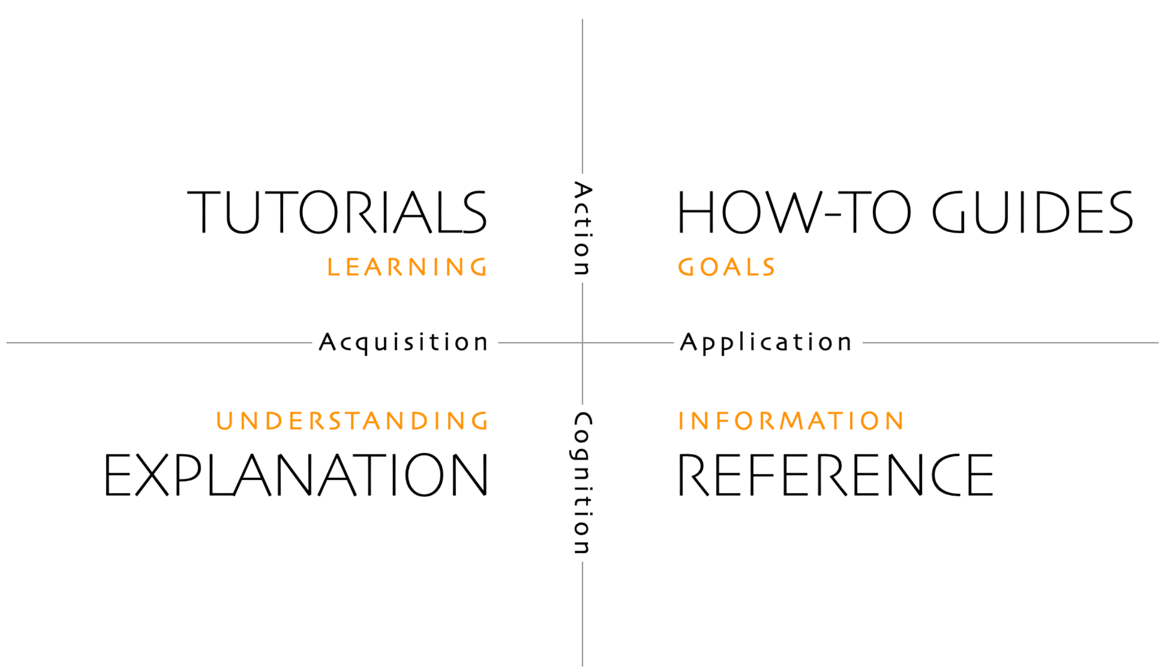
https://thewhitetulip.gitbook.io/py/06-file-handling

4.1.3 Documentation Style

https://diataxis.fr/ (originally developed at: https://docs.divio.com/documentation-system/) — divides documentation along two axes:

- Action (Practical) vs. Cognition (Theoretical)
- Acquisition (Studying) vs. Application (Working)

resulting in a matrix of:



where:

1. readme.md — (Overview) Explanation (understanding-oriented)
2. Templates — Tutorials (learning-oriented)
3. gcodepreview — How-to Guides (problem-oriented)
4. Index — Reference (information-oriented)

Straddling the boundary between coding and documenation are `docstrings` and general coding style with the latter discussed at: https://peps.python.org/pep-0008/

4.1.4 Holidays

Holidays are from https://nationaltoday.com/

4.1.5 DXFs

http://www.paulbourke.net/dataformats/dxf/
https://paulbourke.net/dataformats/dxf/min3d.html

4.2 Future

4.2.1 Images

Would it be helpful to re-create code algorithms/sections using OpenSCAD Graph Editor so as to represent/illustrate the program?

4.2.2 Bézier curves in 2 dimensions

Take a Bézier curve definition and approximate it as arcs and write them into a DXF?
https://pomax.github.io/bezierinfo/
https://ciechanow.ski/curves-and-surfaces/
https://www.youtube.com/watch?v=aVwxzDHniEw
c.f., https://linuxcnc.org/docs/html/gcode/g-code.html#gcode:g5

4.2.3 Bézier curves in 3 dimensions

One question is how many Bézier curves would it be necessary to have to define a surface in 3 dimensions. Attributes for this which are desirable/necessary:

- concise — a given Bézier curve should be represented by just the point coordinates, so two on-curve points, two off-curve points, each with a pair of coordinates
- For a given shape/region it will need to be possible to have a matching definition exactly match up with it so that one could piece together a larger more complex shape from smaller/simpler regions
- similarly it will be necessary for it to be possible to sub-divide a defined region — for example it should be possible if one had 4 adjacent regions, then the four quadrants at the intersection of the four regions could be used to construct a new region — is it possible to derive a new Bézier curve from half of two other curves?

For the three planes:

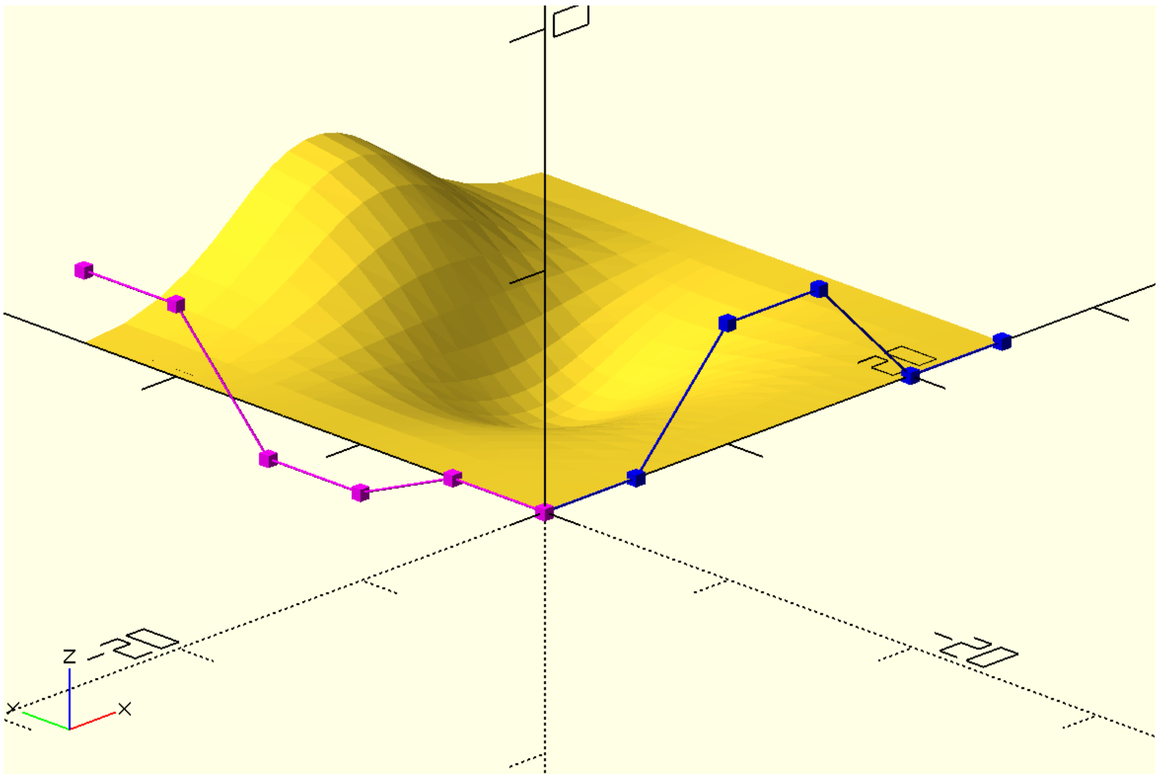
- XY
- XZ
- ZY

it should be possible to have three Bézier curves (left-most/right-most or front-back or top/bottom for two, and a mid-line for the third), so a region which can be so represented would be definable by:

```
3 planes * 3 Béziers * (2 on-curve + 2 off-curve points) == 36 coordinate pairs
```

which is a marked contrast to representations such as:
https://github.com/DavidPhillipOster/Teapot
and regions which could not be so represented could be sub-divided until the representation is workable.

Or, it may be that fewer (only two?) curves are needed:



https://pages.mtu.edu/~shene/COURSES/cs3621/NOTES/notes.html

c.f., https://github.com/BelfrySCAD/BOSL2/wiki/nurbs.scad and https://old.reddit.com/r/OpenPythonSCAD/comments/1gjcz4z/pythonscad_will_get_a_new_spline_function/

4.2.4 Mathematics

https://elementsofprogramming.com/

References

[ConstGeom] Walmsley, Brian. *Construction Geometry*. 2d ed., Centennial College Press, 1981.

[MkCalc] Horvath, Joan, and Rich Cameron. *Make: Calculus: Build models to learn, visualize, and explore*. First edition., Make: Community LLC, 2022.

[MkGeom] Horvath, Joan, and Rich Cameron. *Make: Geometry: Learn by 3D Printing, Coding and Exploring*. First edition., Make: Community LLC, 2021.

[MkTrig] Horvath, Joan, and Rich Cameron. *Make: Trigonometry: Build your way from triangles to analytic geometry*. First edition., Make: Community LLC, 2023.

[PractShopMath] Begnal, Tom. *Practical Shop Math: Simple Solutions to Workshop Fractions, Formulas + Geometric Shapes*. Updated edition, Spring House Press, 2018.

[RS274] Thomas R. Kramer, Frederick M. Proctor, Elena R. Messina.
https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=823374
https://www.nist.gov/publications/nist-rs274ngc-interpreter-version-3

[SoftwareDesign] Ousterhout, John K. *A Philosophy of Software Design*. First Edition., Yaknyam Press, Palo Alto, Ca., 2018

Command Glossary

. 25

setupstock setupstock(200, 100, 8.35, "Top", "Lower-left", 8.35). 23

Index

Routines

Variables