Curved Crease Origami in Semi-Rigid Materials

Landon Carter
lcarter@mit.edu

ABSTRACT

I have explored alternative material choice and manufacturing methods for creating curved crease origami, and have developed a software package that is useful not just for curved crease origami, but origami in general. I have used this software package to produce a work of art from aluminum, and have published this software with a helpful reference guide. In the process of producing this piece, I have also optimized the cutting parameters (though only for 1/32” aluminum intended to be bent by hand).

ACM Reference format:
Landon Carter. 2017. Curved Crease Origami in Semi-Rigid Materials. In *Proceedings of Geometric Folding Algorithms, Cambridge, Massachusetts USA, May 2017 (6.849’17),* 11 pages.
DOI: xxxxxxxxxxxxxxxxxxxxxxx

1 INTRODUCTION

Curved crease origami is a unique type of origami, where creases are not straight, but curved. This gives curved crease origami a unique aesthetic within origami, but also means that relatively little work has gone into curved crease origami. In particular, to the author’s knowledge, there are no software packages which currently support curved crease origami.

Additionally, though there are a number of software packages designed to manipulate origami, none are optimized for producing files which can be CNC machined by a waterjet, laser cutter, vinyl cutter, or similar. Many software packages only give a visual crease pattern as output, which is unsuitable for CNC manufacture. It would be nice if there were a software suite which could be used to convert these crease patterns into files useful for CNC manufacture. Unfortunately, there are not yet CNC creasing machines available to hobbyists or even most universities - the only current use of CNC bending machines is in specialized sheet metal shops, or mass manufacturing of sheet metal components. Therefore, a software package targeted to CNC manufacture of origami parts should be capable of working with vector-based through-cutting machines, such as the aforementioned waterjet, laser cutter, and vinyl cutter.

The unique aesthetics of curved crease origami tend to highlight the flowing, smooth nature of paper. Therefore, it would be artistically interesting to break this connection, producing curved crease origami out of something that is tough and strong - sheet metal. Metal is also ductile, which will allow it to bend without snapping, a critical feature for origami. Wood and almost all materials which can be lasercut are not ductile, and would require special, more advanced patterns to produce origami. Therefore, I decided to focus my efforts on sheet metal.

2 CURVED CREASES

Curved creases have not had much research done relative to the rest of computational origami. Some highlights are Huffman’s original work, Demaine’s work, and Koschitz’s work [1, 2]. Koschitz’s thesis in particular provides a fairly extensive review of previous work on curved crease origami [3].

Some motivating examples of curved crease origami for me are the work of Erik and Martin Demaine, especially the following few models, which are all based on the original Bauhaus design of 1920’s. Some examples are shown in Figure 1.

(a) Computational Origami (2008)

(b) Tsunami (2009)

(c) Waterfall (2014)

Figure 1: Curved crease origami by Erik and Martin Demaine, the inspiration for this project.

3 SOFTWARE PACKAGE

The primary piece of software is a `Python` module for applying cut patterns to SVG input files. The parameters for these cut patterns can be easily adjusted, and custom cut patterns can be used.

3.1 Parameter Input

There are just a few important parameters required to define the software behavior. The difference between offset cuts and custom cuts is made to be as small as possible. Most important are the `tab_length` and `cut_length` parameters, which control the discretization. Ranges are required for these values so that the software can find an appropriate integer number of cuts that can be evenly spaced to partition each crease.

`tab_length`:	A range for the length of solid tabs, measured in pixels. The algorithm will try to make tabs in the middle of this range.
`cut_length`:	A range for the length of cuts, measured in pixels. The algorithm will try to make cuts in the middle of this range.
`cut_width`:	The width of the cut, in pixels. This is measured normal to the crease direction. If a cut file is provided, this parameter is ignored.
`cut_file`:	The filename for specifying a custom cut. If a simple offset cut is desired, this should be false.

Table 1: JSON specification for the parameter input file.

Variable Name	Offset Cut Sample	Cut File Sample
`tab_length`	[5,10]	[5, 10]
`cut_length`	[50, 60]	[50, 60]
`cut_width`	5	5 (ignored)
`cut_file`	`false`	`cut.svg`

Table 2: Sample JSON inputs for both offset cut mode and custom cut file mode.

3.2 Software Package Operation

The full code for the software is provided in Appendix A, but I'll highlight the functionality of a few key functions

3.2.1 `cut`

Inputs: `(input_svg, params, cut_file=None)`

This is the main function within the module, and controls the overall operation. The function operates as follows

(1) Step through each path in the input SVG.
(2) For each path in the input SVG, determine the length, and determine a number of cuts which will be made.
(3) For each cut, determine the start and end positions.
(4) Apply the cuts. That involves either calling `partial_offset_curve` or `scale_cut`. In `partial_offset_curve`, a positive offset curve and negative offset curve are created, and joined together to make a continuous cut.

3.2.2 `partial_offset_curve`

Inputs: `(path, start_t, end_t, offset_distance, steps=10)`

Partial offset curve is used for offset cuts, and follows the curve of the path.

(1) The path's normal is calculated by the svgpathtools library, which calculates the normal by taking the normal of the derivative of the appropriate curve. Because SVG paths are defined by Bezier curves, arcs, or lines, the derivative can be easily computed symbolically.
(2) The curve is broken into `step` segments, and each point is offset by the appropriate normal. These points are then reconnected by lines, for a close approximation. The number of steps can be increased to improve the fidelity of the approximation at the cost of slower running time and larger output files. In practice, 10 steps was found to be sufficient for inspection by eye. In the simple circle example, there are 20 cuts, each with 10 steps, and a gap between, so the 10-step approximation is roughly equivalent to a 230-gon.

3.2.3 `scale_cut`

Inputs: `(start_pos, end_pos, cut_file)`

Scale cut is used for custom cuts, and scales the input cut file before repositioning it appropriately.

(1) The start and end points are used to determine the offset, rotation, and length of the cut.

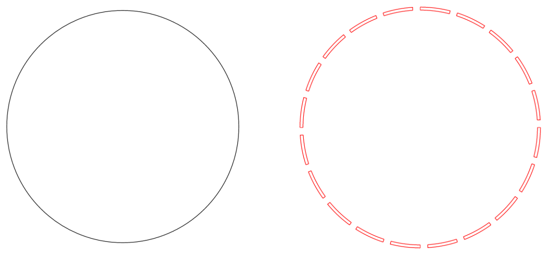


Figure 2: Input and output for a basic circle example with offset curve.

(2) The input cut file is scaled to the correct length. This process is memoized to improve performance - within a single path, the length of each cut should be the same, and multiple paths with the same total length will also share the same cut length.
(3) The rescaled cut is translated and rotated into the correct position.

3.3 Sample In/Out

As you can see in Figures 6 and 7, the custom cut solution still has some serious drawbacks. Namely, it is unable to follow the curves, and only approximates them via interpolation. Therefore, it does not deal well with sharp angles or tight curves. This is also true for the offset cut, but to a lesser example. The offset cut can deal well with tight curves, but does not take into account corners, and therefore may have cuts which span these corners.

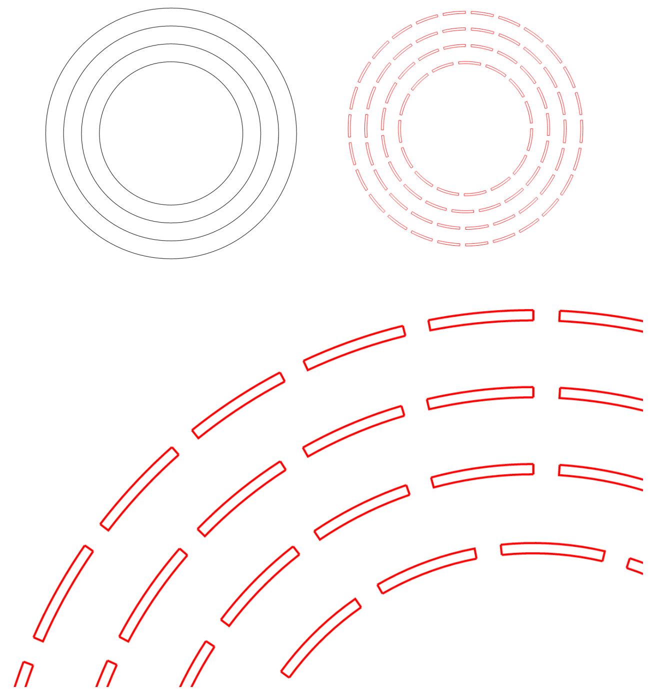
4 ART

4.1 Experimentation

Hand foldability vs strength in 1/32 aluminum was investigated. Further experimentation in steel was planned as well, but waterjets were unavailable when steel stock came in. The experimental results for aluminum are compiled in Table 3.

4.2 Aluminum

The primary goal of this project was to produce an actual piece of art, a metal curved-crease origami structure. I decided to focus on the Bauhaus design which Erik and Martin Demaine also explored, as in Figure 1. I thought I was reasonably successful in this endeavor,



(a) Detail of the cut pattern.

Figure 3: Input and output for a multi-circle example with offset curve. This is similar to the pattern waterjetted in aluminum.

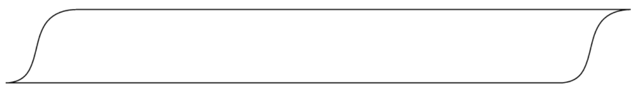


Figure 4: Cut file used in custom cut examples. This is an example of a cut pattern where the tab is designed to warp and twist in a specific way. In theory, this should increase the apparent ductility of the material. In practice, this design was unsuccessful.

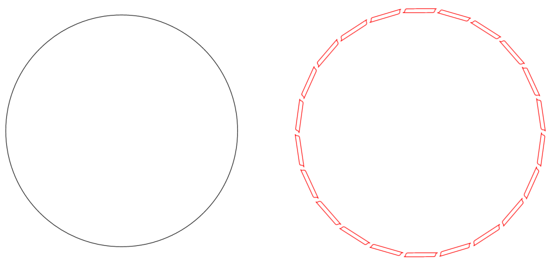
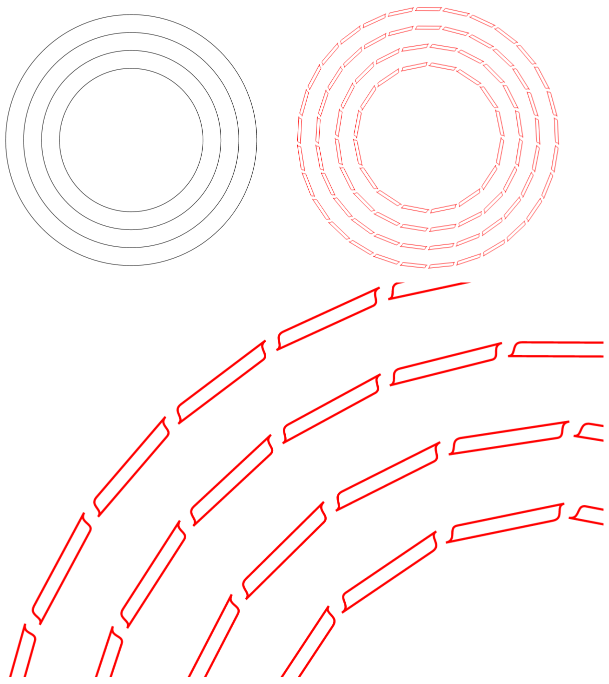


Figure 5: Input and output for a basic circle example with custom cut.



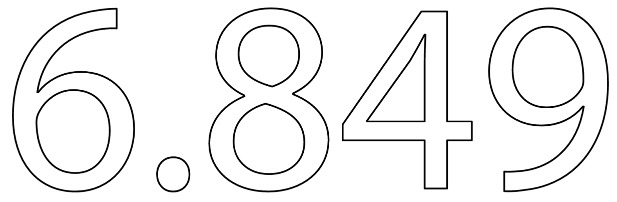
(a) Detail of the cut pattern.

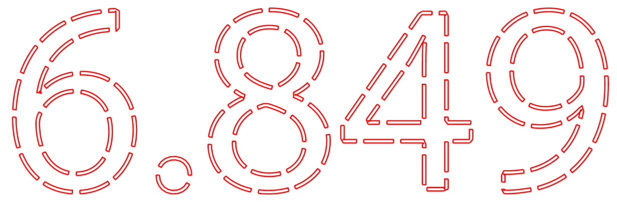
Figure 6: Input and output for a multi-circle example with custom cut.

tab_l	cut_l	Results
0.020	1.50	Too weak.
0.020	0.50	Too weak. Too many cuts (time)
0.250	1.50	Too stiff, slightly brittle
0.150	1.00	Too stiff, not far off
0.100	1.00	Good mix. Slightly too brittle.

Table 3: Experimentatal results for various cut length parameters in 1/32 aluminum. Sample cut files are shown in Figure 8. All files were scaled such that 1pt=1in, so measurements can be thought of as inch measurements. Unfortunately, svg files make this scaling somewhat nonapparent.

though producing a curved crease origami met with some challenges that were not exposed during the linear testing period for perforations - namely, it was now much harder to fold everything. I found it quite a bit more difficult to fold the curved creases than the equivalent straight creases, and also more difficult to handle folding multiple creases at once. I eventually found success by starting from the middle crease out - that


(a) Input for the classic 6.849 example.


(b) Output with offset cut applied to the 6.849 example.

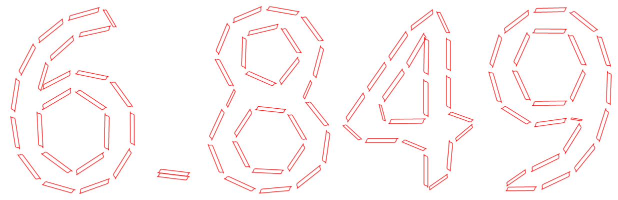

(c) Output with custom cut applied to the 6.849 example.

Figure 7: 6.849 example with offset cut and custom cut.

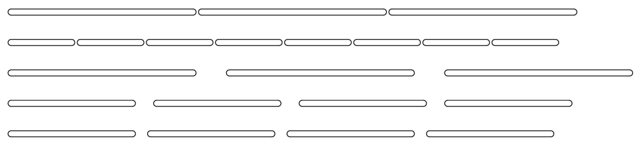


Figure 8: Sample input cut patterns that were used to test various cut parameters. From top to bottom, they correspond to the entries in Table 3.

way, I would always have a flat surface to hold with a pair of pliers when folding.

Unfortunately, during folding, the innermost ring tore off. I was afraid of this, since I was using aluminum, which is not the most ductile material, and because my testing had shown a possibility for cracking during even the course of a dozen or so fold/unfold motions. Another factor likely contributing to the failure of the inner ring was the fact that it had a smaller curvature diameter than any of the other rings - this put extra stress on the tabs, especially in the direction perpendicular to the normal direction - they were flexed along the tab width, instead of the tab length.

Despite this, I wound up with a very pretty piece of art, and observed it following the same folding behavior as paper - the 360 degrees contracted as each ring formed a cone, exactly as expected. I was able to create a simple "rollercoaster" loop that looks quite nice as a desk ornament.

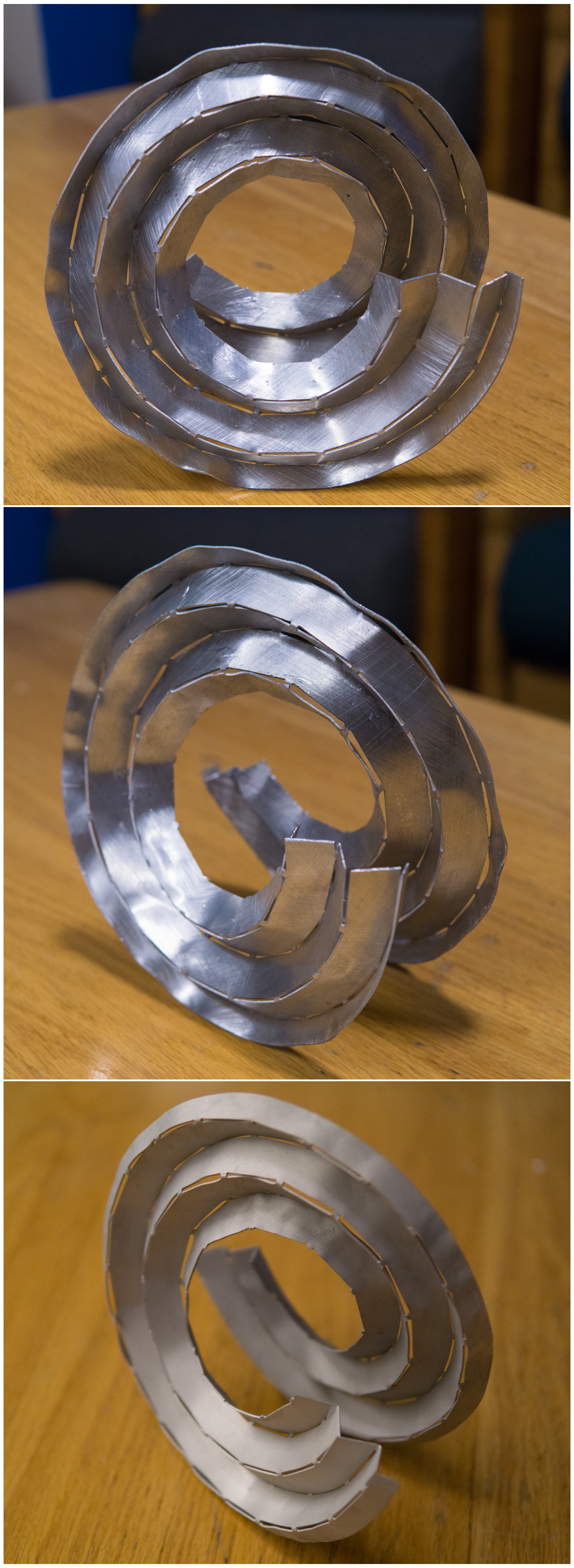
4.3 Steel

After finishing the example in aluminum, I wanted to create another piece in steel, including welding multiple of them together to achieve a ¿360 degree total curvature, as Erik and Martin Demaine did in many of their curved crease origami pieces. Unfortunately, by the time my steel arrived, both of the waterjets that I have access to on campus were down for maintenance - likely from other students also trying to finish their final projects in the last week of school. I was incredibly disappointed by this, because I was honestly looking forward to seeing how ¿360 degrees of curvature would look, and applying all of the lessons I learned when folding the first aluminum example. I still have the aluminum, so I'll email you once one of the waterjets comes back online and I manage to actually cut out a pattern. I realize that won't be able to count for this project, but it'll be beautiful and I already have the material.

I was excited for steel due in particular to the better ductility, which would help prevent the cracking I saw in the aluminum example, and for the easier chance to weld multiple pieces together. Any cracks that did appear could also be welded back together. Though aluminum can be welded, it is much more difficult, and nearly impossible for thin aluminum. Even my best welder friends didn't think they were capable of welding aluminum less than 1/8" thick.

5 CONCLUSIONS

Overall, this project took quite a bit longer than I expected. The software was more difficult to write than I had anticipated, especially because the documentation of the `Python` library I used to interface with SVG's was poor - in particular, I had to read through the source code to discover a method to set the numerical tolerances so that my code would run in a reasonable time.


(a) The aesthetics were significantly improved with a bit of extra care bending back wrinkled sections and sandblasting the piece.

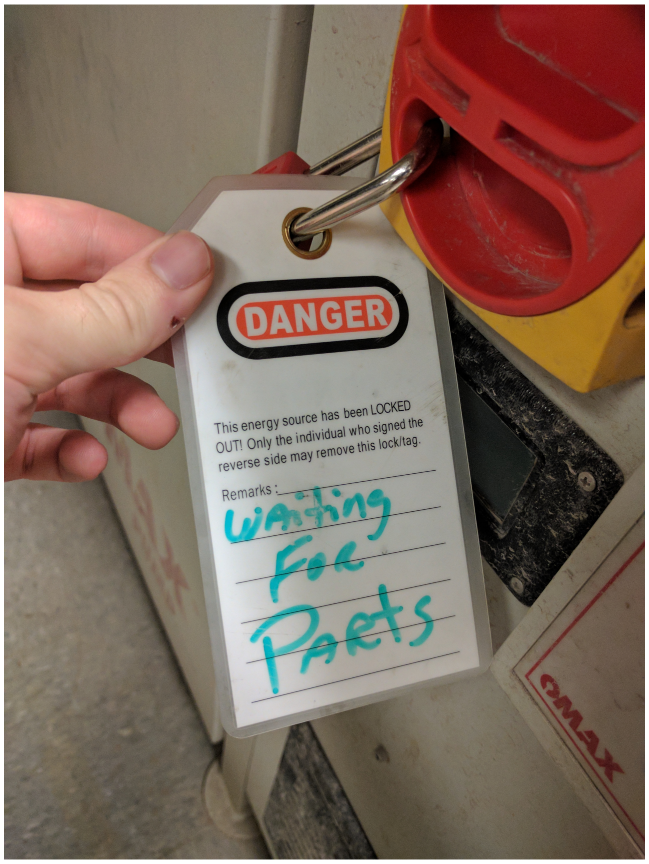

DANGER
This energy source has been LOCKED OUT! Only the individual who signed the reverse side may remove this lock/tag.
Remarks:
Waiting For Parts
OMAX

Figure 10: The N51 waterjet went down for maintenance right as my steel arrived, so I was unable to cut a pattern out of steel. The Makerworks waterjet has been down for the past 3 or 4 weeks, so I wasn't able to use that either. I was very sad to discover this.

Before setting the tolerance to 1e-3, it was set by default to 1e-12. Before the change, my offset code on the multi-circle input would take about 5 minutes to run, and after the change, my offset code took about 3-5 seconds. Ultimately, I was fairly happy with how general the software can be - the software suite can be used with any svg input file, whereas I had originally planned to just iterate in Solidworks and focus more on the physical implementation. By focusing on the software, I was able to create a tool that can be used to produce cut patterns for any origami. This will be useful for future work in metal origami, especially by hobbyists that don't have access to CNC bending machines. I have written a comprehensive readme, and the code is all located at my Github repo. The `Python` code has also been attached as Appendix A.

6 FUTURE WORK

One of the pieces of software I wanted to implement, but did not have time to implement, was a converter from FOLD ↔ SVG. This would increase compatibility with other origami programs, though only slightly, since an unfolded representation would still be necessary. I also spent an undue amount of time looking into unfolding algorithms, and found O'Rourke's review article [4] very informative (namely, there are no known good algorithms, and it is not even known if all convex polyhedra are unfoldable). I considered implementing a slow algorithm to test all possible unfoldings for a specific input (equivalent to testing every spanning tree on the polyhedral edge graph, which is not actually that large - the Matrix-Tree theorem allows you to calculate the number of spanning trees for a graph). Testing each unfolding would take $O(n \log(n))$ time by sorting the resultant edges before comparing them. Then, the locality of comparison can be preserved, very similarly to the convex hull algorithm. I may implement this, as well as the FOLD ↔ SVG converter, in the future.

Another aspect of the project which I did not have time to explore was the extension to wood and other materials, especially laser-cuttable materials. I think that cut patterns for these materials could be created by modifying the offset-cut method to produce multiple (3-6) parallel offset cuts. The difficulty here would be ensuring that the tabs don't line up, otherwise the connection would still be connected by a single brittle member. A custom cut file could also potentially work, though the same difficulty would occur, in addition to the difficulties of approximating curves with straight lines.

ACKNOWLEDGMENTS

I would like to thank the 6.849 staff for their helpful comments on the project proposal and presentations, which provided valuable direction in completing this project. I would also like to thank them for an incredibly fun and engaging class - I truly enjoyed it, and am looking forward to taking 6.851 in the fall!

REFERENCES

[1] Erik D. Demaine, Martin L. Demaine, David A. Huffman, and Duks Koschitz. 2010. Reconstructing David Huffman's legacy in curved-crease folding. *Origami: Proceedings of the 5th International Conference on Origami in Science, Mathematics, and Education* 5 (2010). http://erikdemaine.org/papers/Huffman_Origami5/paper.pdf

[2] Erik D. Demaine, Martin L. Demaine, David A. Huffman, Duks Koschitz, and Tomohiro Tachi. 2015. Characterization of Curved Creases and Rulings: Design and Analysis of Lens Tessellations. *CoRR* abs/1502.03191 (2015). http://arxiv.org/abs/1502.03191

[3] Duks Koschitz. 2014. *Computational Design with Curved Creases: David Huffman's Approach to Paperfolding.* Ph.D. Dissertation. Massachusetts Institute of Technology.

[4] Joseph O'Rourke. 2008. Unfolding Polyhedra. (2008). https://pdfs.semanticscholar.org/5395/012766afda5b9d40bbb62b335bc04de958d7.pdf

A SVG_TOOLS.PY

```python
"""SVG tools for curved crease origami."""
# disable warnings about variable names
# pylint: disable=C0103

import json
import cmath
import math
import sys
import svgpathtools as svg

def memodict(f):
    """ Memoization decorator for a function taking a single argument.
    From http://code.activestate.com/recipes/578231-probably-the-fastest-
   memoization-decorator-in-the-/
    """
    class memodict(dict):
        def __missing__(self, key):
            ret = self[key] = f(key)
            return ret
    return memodict().__getitem__

def read_json_params(param_file):
    with open(param_file, 'r') as f:
        params = json.loads(''.join(f))
    return params

def read_svg(input_svg_file):
    """Reads an svg file and returns a list of svg path objects."""
    paths, attributes = svg.svg2paths(input_svg_file)
    return (paths, attributes)

def partial_offset_curve(path, start_t, end_t, offset_distance, steps=10):
    nls = []
    diff = end_t - start_t
    for k in range(steps):
        t = start_t + diff*k/steps
        offset_vector = offset_distance * path.normal(t)
        p = path.point(t)
        # print(p)
        nls.append(p + offset_vector)
    connect_the_dots = [svg.Line(nls[k], nls[k+1]) for k in range(len(nls) -
   1)]
    offset_path = svg.Path(*connect_the_dots)

    return offset_path

@memodict
def scale_cut_file(input_tuple):
    (scale, cut_file) = input_tuple
    new_file = svg.Path()
    for path in cut_file:
        new_file.append(type(path)(*[point*scale for point in path]))
    return new_file

def scale_cut(start_pos, end_pos, cut_file):
    print(type(start_pos))
    print(start_pos)
    vec = end_pos - start_pos
    r, theta = cmath.polar(vec)
    (xmin, xmax, ymin, ymax) = cut_file.bbox()
    R = xmax - xmin
    scale = float(r)/R

    mid_pos = (start_pos + end_pos)/2.0

    new_file = scale_cut_file((scale, cut_file))

    center = (scale*(xmin+xmax)/2.0) + (scale*(ymin+ymax)/2.0)*1j

    new_file = new_file.rotated((180/math.pi)*theta, center)
    new_file = new_file.translated(mid_pos - center)

    return new_file

def cut(input_svg, params, cut_file=None):
    print('beginning cuts')
    l_range = params['cut_length']
    t_range = params['tab_length']

    if cut_file is None:
        positive = svg.Path()
        negative = svg.Path()
    else:
        cut_paths = []

    for path in input_svg:
        print('starting a path')
        # calculate number of cuts
        pathlength = float(path.length(error=1e-3))
        maxcuts = pathlength/(l_range[0]+t_range[0])
        mincuts = pathlength/(l_range[1]+t_range[1])
        ncuts = int((mincuts + maxcuts)/2)
        if ncuts == 0:
            print path
            print pathlength
        ltotal = pathlength/ncuts
        l = max(ltotal - (t_range[0] + t_range[1])/2, l_range[0])
        t = ltotal - l
        print('there are %s cuts' % ncuts)

        # accumulate segments
        for i in range(ncuts):
            print("%s/%s" % (ltotal*i, pathlength))
            cut_start = path.ilength(ltotal*i, s_tol=1e-3, error=1e-3)
            cut_end = path.ilength(ltotal*(i+1) - t, s_tol=1e-3, error=1e-3)
            if cut_file is None:
                positive.append(partial_offset_curve(path, cut_start, cut_end
   ,
                                                     params['cut_width']/2))
                negative.append(partial_offset_curve(path, cut_start, cut_end
   ,
                                                     -params['cut_width']/2))
            else:
                cut_start = path.point(cut_start)
                cut_end = path.point(cut_end)
                cut_paths.append(scale_cut(cut_start, cut_end, cut_file))
        print('cut a path')

    if cut_file is None:
        print('number of segments: %s' % len(positive))
    else:
        print('number of segments: %s' % len(cut_paths))

    if cut_file is None:
        cut_paths = []
        for i in range(len(positive)):
            pos_path = positive[i]
            neg_path = negative[i]
            pos_path.append(svg.Line(pos_path[-1].end, neg_path[-1].end))
            pos_path.extend(neg_path[::-1])
            pos_path.append(svg.Line(neg_path[0].start, pos_path[0].start))
            cut_paths.append(svg.Path(*pos_path))

    return cut_paths


if __name__ == '__main__':
    """ Usage: python svg_tools.py params.json in.svg out.svg """
    cut_param_file = sys.argv[1] if len(sys.argv) > 1 else 'params.json'
    input_svg_file = sys.argv[2] if len(sys.argv) > 2 else 'in.svg'
    output_svg_file = sys.argv[3] if len(sys.argv) > 3 else 'out.svg'


    params = read_json_params(cut_param_file)
    input_cut_file = params['cut_file']

    (input_svg, svg_attributes) = read_svg(input_svg_file)
    print input_svg

    if input_cut_file:
        (input_cut, cut_attributes) = read_svg(input_cut_file)
        print input_cut[0]
        output_paths = cut(input_svg, params, input_cut[0])
    else:
        output_paths = cut(input_svg, params)

    svg.wsvg(output_paths, 'r'*len(output_paths), filename=output_svg_file)
```