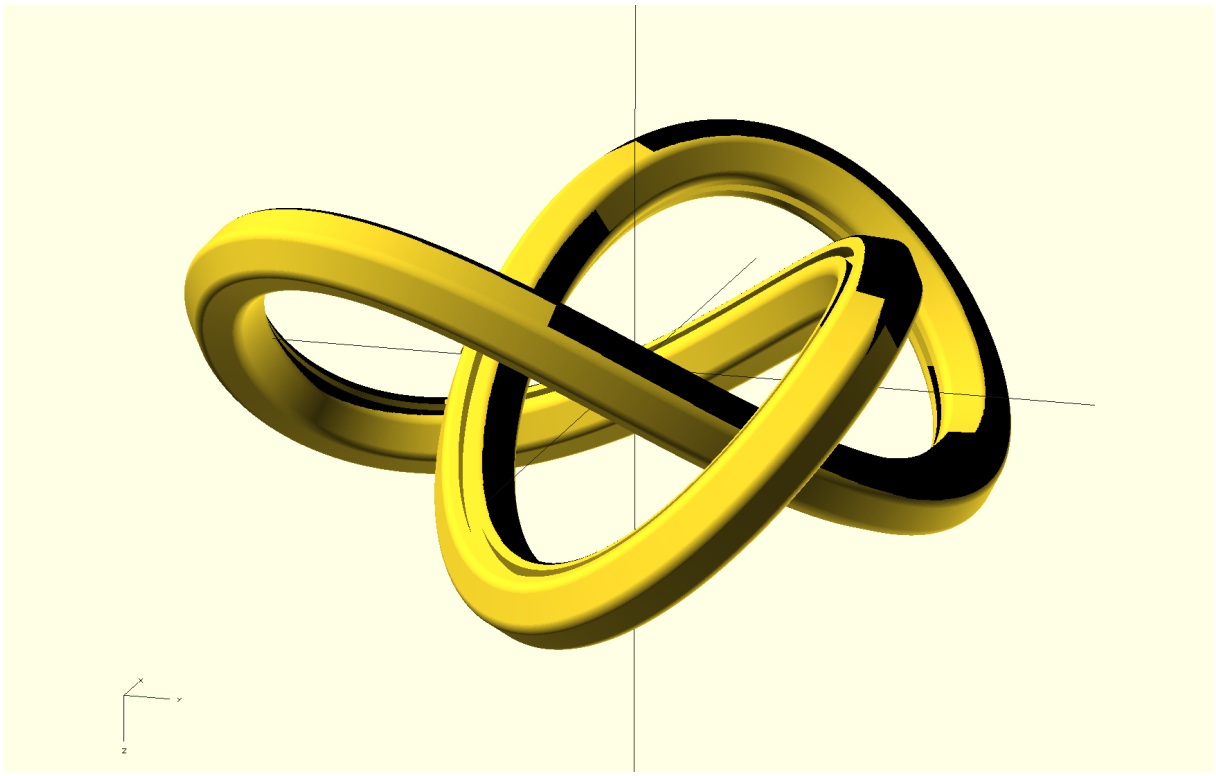
Extension of openSCAD with loop() and loop_extrude() and some other hacks.

Ruud Vlaming

October 19, 2013

Abstract



Version $^{1}/_{\sqrt[7]{2}}$

Proofreading of version $\sqrt[-3]{2}$ by William Adams.

1 Motivation

Sometime ago i first encountered openSCAD after having played a while with FreeCAD. Although objects can be manipulated in python there, it was not as intuitive as I expected. OpenSCAD is much easier to use, certainly when your experience with CAD programs is not that extensive. However, openSCAD is also lacking some basic features. This extension fills in the features that were lacking for my project. Hopefully it is of use to you as well.

2 The source code

The source code can be downloaded from my fork on github, branch 'loop_extension':

`https://github.com/devlaam/openscad.git`

You have to compile it yourself, following the instructions from the openSCAD website. The source is released under the same license as openSCAD.

3 Syntax

We first present the syntax of both loop commands. It is not meant to be rigorously correct (in a production language), just to given an idea how the stuff should be used.

3.1 Loop and alike

```
loop(
   points^(req),def = [ "cartesian"^def, "polar"^opt, [x,y]^#3+, ] ,
   rect^(req),opt   = [ "clock"^opt, [width, height^opt]^req ] ,
   poly^(req),opt   = [ n^>2,req, "flat"^opt, "clock"^opt, "Q1"^opt, {"side"^def|"rin"|"rout"}^req, [size^#1+,]^req ] ,
   vertices^opt     = [ n^#0+, [ {"def"^def|"lin"|"arc"|"bez"}^req, {"iden"^def|"cen"|"csx"|"csy"|"csxy"}^req, [x,y]^#0+ ] ] ,
   edges^opt        = [ n^#0+, [ {"def"^def|"lin"|"wav"|"bez"}^req, {"iden"^def|"cen"|"csx"|"csy"|"csxy"}^req , [x,y]^#0+ ] ] );

loop_extrude(
   points^(req),def = [ "cartesian"^def, "cylinder"^opt, "sphere"^opt, [x,y,z]^#3+, ] ,
   rect^(req),opt   = [ "clock"^opt, [width, height^opt]^req ] ,
   poly^(req),opt   = [ n^>2,req, "flat"^opt, "clock"^opt, {"side"^def|"rin"|"rout"}^req, [size^#1+,]^req] ,
   vertices^opt     = [ n^#0+, [ {"def"^def|"lin"|"arc"|"bez"}^req, {"iden"^def|"cen"|"csx"|"csy"|"csxy"}^req, [x,y,z]^#0+ ] ] ,
   edges^opt        = [ n^#0+, [ {"def"^def|"lin"|"bez"}^req, {"iden"^def|"cen"|"csx"|"csy"|"csxy"}^req , [x,y,z]^#0+ ] ] ,
   segments^opt     = { [ n^#0+, [ {"show"^def|"hide"}^req, {"in"^def|"out"}^req, {"rel"^def|"abs"}^req , val^opt] ] |
                        [ n^#0+, [ {"def"^def|"lin"|"wav"|"bez"}^req, {"iden"^def|"cen"|"csl"}^req, [x,y,z]^#0+ ] ] }^opt );
```

The (curly) brackets '{', '}' and '|' indicate that only one of the keywords should be specified, and are not part of the syntax. Some keywords are optional, indicated by superscript `opt`, or required `req` or may be default `def` in case of absence of any keyword; if a keyword is required, but a default is specified, it is effectively optional. The superscript `#3+` indicates the minimal number of times an element must be present (three in this case), `>2` shows the minimum value for `n`, which represent a vertex, edge or segment number. Below we give a short description, but the use of the keywords will become clear in the examples.

points Indicates the points (vertices) the loop passes through. Vertices are numbered from 1 to N, in the order they appear. Edges connect the vertices, and are numbered as well from 1 to N, where edge 1 connects vertex 1 and 2.

poly Constructs a regular polygon, vertex numbering is counter clock wise and the first vertex is located on the X-axis.

vertices Allows for possible modification, such as rounding, of each vertex. Without further designation, each supplied dressing acts on the corresponding vertex number and all that come after that.

edges Allows for possible modification, such as deformation, of each edge. Without further designation, each supplied dressing acts on the corresponding edge number and all that come after that.

segments Allows for possible modification, such as deformation, of each segment. A segment is a vertex or an edge, numbered from 1 to 2N. Segment 1 corresponds to vertex 1, segment 2 to edge 1, segment 3 to vertex 2 etc. Without further designation, each supplied dressing acts on the corresponding segment number and all that come after that.

Note that, in the latest update you can define sequences of points with repeated coordinates using empty strings. This is especially handy when you have long parameter names in modules. Thus

```
module long(length, width, height)
{ loop_extrude(
   points=[
    [0,0,0],[length,0,0],[length,width,0],[length,width,height],[0,width,height],[0,0,height]])
  circle(1, $fn=20); }
```

can be written down as:

```
module short(length, width, height)
{ loop_extrude(
   points=[
    [0,0,0],[length,0,0],["",width,0],["","",height],[0,"",""],[0,0,""]])
  circle(1, $fn=20); }
```

which can make a significant difference when you follow long trajectories of points. Besides that, the notation makes immediately clear what the changes are.

3.2 Boxing and Positioning

Below find the syntax of the box() and position() commands. The first enables you to draw a bounding box around the object, that may be modified in size, the second to position an object independent of its own position, but dependent to an other objects position or the origin.

```
box(
   add^{opt,def} = d_all^#1,
   xadd^opt      = d_x^#1,
   yadd^opt      = d_y^#1,
   zadd^opt      = d_z^#1,
   act^opt       = {true^def|false}^req );

position(
   {xmin|xmid|xmax}^opt = Δx^#1,
   {ymin|ymid|ymax}^opt = Δy^#1,
   {zmin|zmid|zmax}^opt = Δz^#1,
   keep^opt             = {true|false^def}^req] );
```

Note that the box() command operates on one argument. If it is applied on a list of arguments, they are implicitly unified. The position may operate on one or two arguments. If only one argument is supplied, positioning will be relative to the coordinate origin, if two (or more) arguments are supplied, positioning of the first object will be relative the last object. All other objects are ignored.

3.3 Centering

The keyword 'center' can be used at some of the primitives in the original version of openSCAD. However, this has been done in the rather counter intuitive fashion to my opinion. Cubes and squares are per default not centered (i.e. they are located in the first quadrant) but can be centered by setting the center keyword to 'true'. Spheres and circles are centered per default, but cannot be brought to the first quadrant by setting the center keyword to 'false'. The cylinder is kind a centered half-way, per default, and stays that way when center is set to 'false', but becomes fully centered when center is set to 'true'.

In my version, the sphere and circle also accept the keyword center and move to the first quadrant when this is set to false, the default remains 'true'. We cannot change the behavior of the cylinder without breaking code, so we leave this as is.

In extension to this, all primitives accepting 'center' with a boolean will also accept center with a vector $[\delta x, \delta y, \delta z]$ as argument where these variables can be -1, 0 or $+1$. A 0 will lead to centering along the axis, whereas -1 will shift it to the negative side, and $+1$ to the positive side. Thus

```
sphere(4, center=[0,1,-1]);
```

will display the sphere $x^3 + (y-4)^3 + (z+4)^3 = 64$, thus centered along the x-axis, with positive y and negative z coordinates. This brings at least some uniformity in the use of that keyword. Furthermore, it can be quite practical to partially center an object. No examples are given in this document for this extension.

3.4 Vector addition and generation

Sometimes you want to add an element to a vector. In this version this is possible through the following syntax:

```
echo( [4,5,6] + 7 );                // generates one array [4,5,6,7]
echo( [1,0,0] + [[0,1,0],[0,0,1]] ); // generates one array [[1,0,0],[0,1,0],[0,0,1]]
echo( "jan" + ["piet","klaas"] );    // generates one array ["jan","piet","klaas"]
echo( [] + [[true],[false]] );      // generates [[true], [false]], so no change!
echo( [1,0,0] + [0,1,0] );          // generates [1,1,0] !!
```

Extension of the array only works when the arrays cannot be added in the standard way, so this is backwards compatible and should not break existing code. Therefore you cannot add to 'equal' arrays, see the last example above. Also, adding 'empty' arrays does not have an effect.

If you need the generate a large set of points, make use of the for(<range>,[vector]) function, for example:

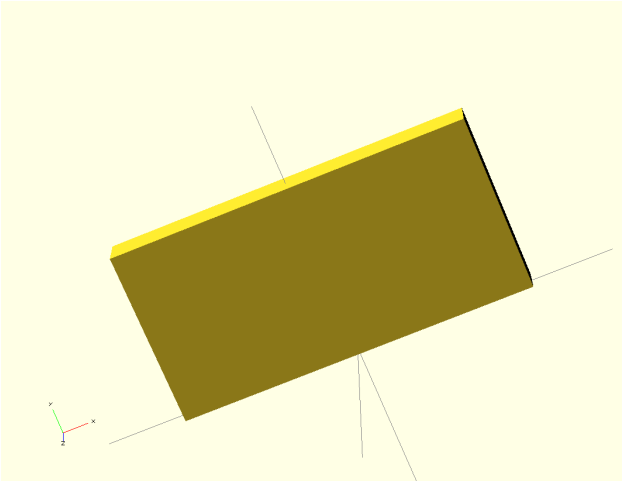
```
function p(t) = for(i=[0:t],[i]);
function q(t) = for(i=[0:t],[i,2*i,t]);
echo( p(5) ); //generates [[0], [1], [2], [3], [4], [5]]
echo( q(5) ); //generates [[0, 0, 5], [1, 2, 5], [2, 4, 5], [3, 6, 5], [4, 8, 5], [5, 10, 5]]
```

ranges can be defined in the same way as for the object related for operator, but only one variable can be used, and the vector index is not allowed. Of course, the second argument can be a function itself, as long as that returns a vector.

4 Examples, for the 2D loop() primitive

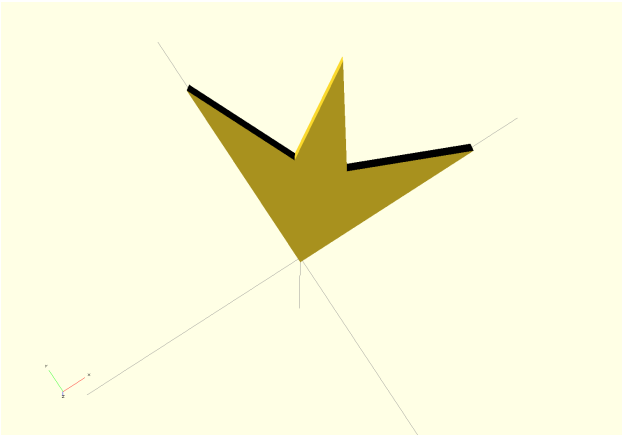
Big fat **warning** to start out with: Just "copy-past" the examples from this pdf into openSCAD may *not work at all* because invisible characters are copied along that are not displayed but invalidate the instructions. OpenSCAD will display nothing (no top level object found), and source highlighting may also be absent.

```
loop(points=[[-5,5],[5,5],[5,0],[-5,0]]);
loop([[-5,5],[5,5],[5,0],[-5,0]]);
```



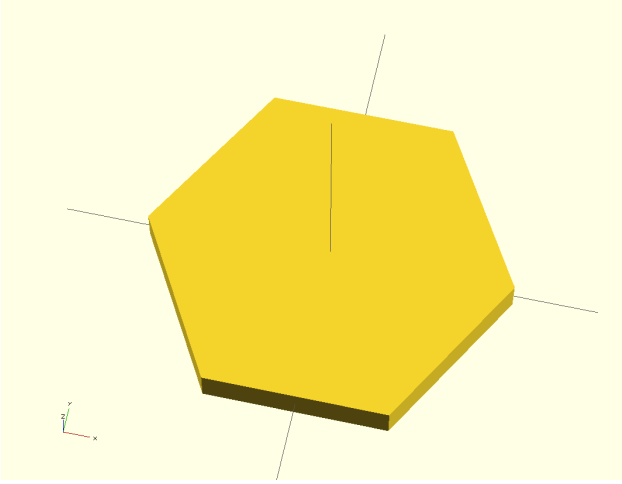
With points it is possible to specify a number of 2D points that are drawn as a 2D primitive figure. The last point is automatically connected with the first. The points are numbered 1,2 ... starting with the left point first. The designation 'points' can be omitted, the first argument is understood as being points. From now on the points themselves will be called vertices, and the connecting lines, edges. The edges are numbered themselves, with edge 1 connecting vertex 1 and 2.

```
loop(points=["polar",[0,0],[4,0],[2,30],[4,45],[2,60],[4,90]]);
```



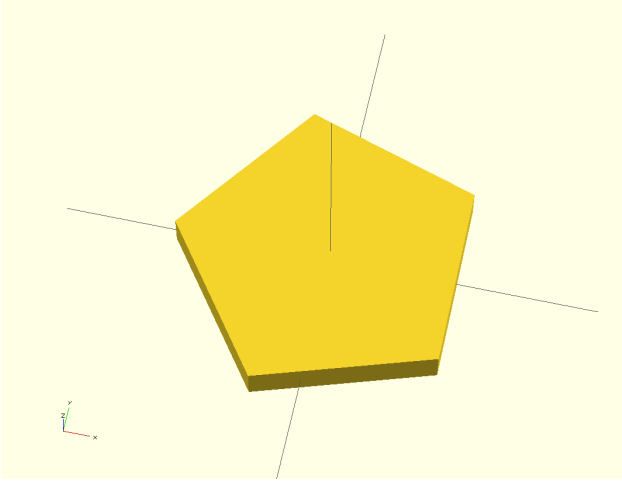
It is possible to define the coordinates of the points in polar form, thus $[r, \phi]$. Angles are, like usual in openSCAD, in degrees.

```
loop(poly=[6,[5]]);
loop(poly=[6,"side",[5]]);
loop(poly=[6,"rin",[5]]);
loop(poly=[6,"rout",[5]]);
```



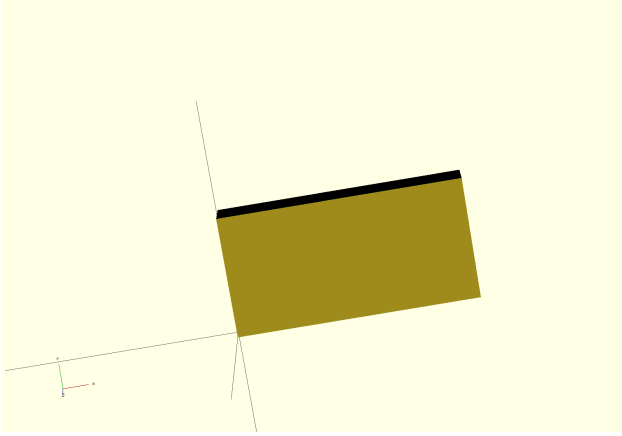
With ‘poly’ it is possible to draw many regular polygons. The first (integer) parameter specifies the number of vertices, the second, which must be enclosed in square brackets, the size. The minimum number of vertices is three. Polygons are always centered. The vertices are numbered from 1 to N, starting with the first vertex on the X-axis and increasing the numbering counter-clockwise. This is conform the mathematical definition of the positive angle. Per default the given size equals the size of the side. It is possible to specify other sizes. These are ‘rin’ and ‘rout’. In these cases the size defines the radius of the inner or outer circle.

```
loop(poly=[5,"flat",[5]]);
loop(poly=[5,"clock",[5]]);
loop(poly=[5,"flat","clock",[5]]);
```



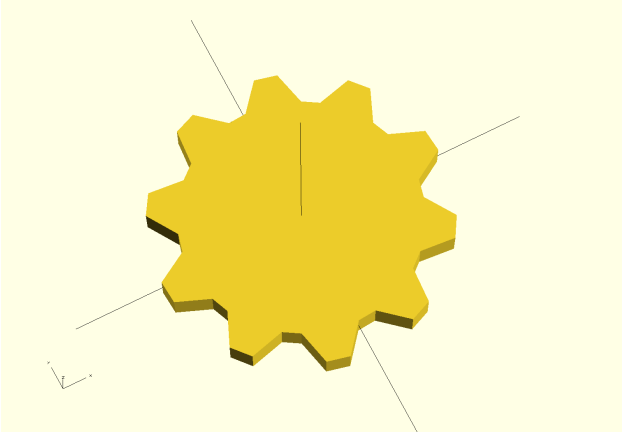
If a flat size is needed on the X-axis, use the option ‘flat’. The first edge is now perpendicular to the X-axis, meaning that the first vertex is below the X-axis. Sometimes it is more convenient to have a vertex numbering that is clockwise and starts at the top. To that end use the option ‘clock’. The first vertex now lies on the positive Y- axis and vertices are numbered towards the X-axis. Note that this option can be combined with ‘flat’. In that case the first edge is on top.

```
loop(poly=[4,"Q1","flat",[3,6]]);
loop(rect=[[3,6]]);
```



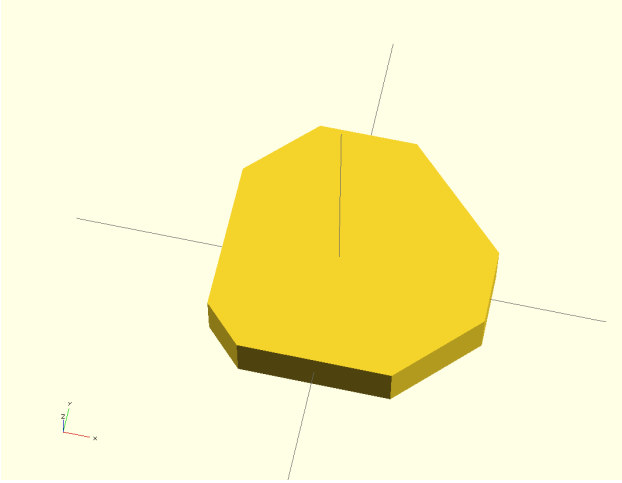
In fact, the polygon generator is more versatile. If used with the ‘default’ option ’side’, you may specify more side sizes which will be used sequentially to generate a cyclic polygon. Note, there are some restrictions, for instance the origin must be inside the polygon, and it must be possible at all to construct the polygon. With the option ‘Q1’, you can enforce the result to be situated in the first quadrant. This is useful when performing a rotation afterwards. Since the rectangle in the first quadrant is used so often, we defined a special keyword for that. Numbering of points however start at the origin.

```
loop(poly=[40,"rout",[5,5,4,4]]);
```



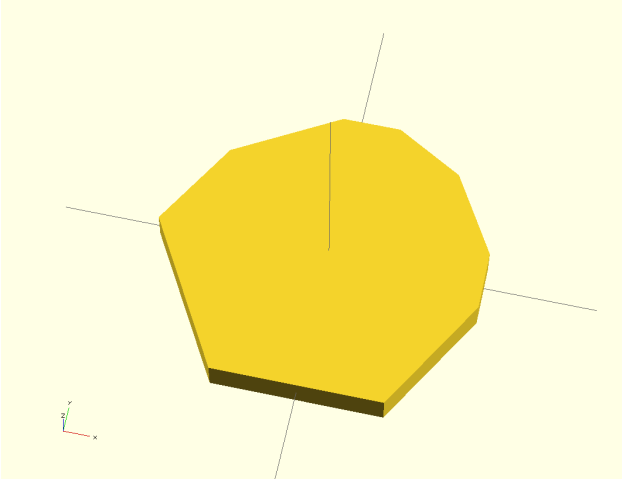
Likewise, you can use ‘rout’ and ‘rin’ with more size parameters to generate equiangular polygons with predefined inner and outer radii. In this case you may easily specify values from which no polygon can be constructed (especially with ‘rin’).

```
loop(poly=[4,[5]], vertices=[["lin",1]]);
loop(poly=[4,[5]], vertices=[["lin",1],["lin",1.3],["lin",2]]);
loop(poly=[6,[5]], vertices=[["lin",1],["lin",1.3],["lin",2],["def"]]);
```



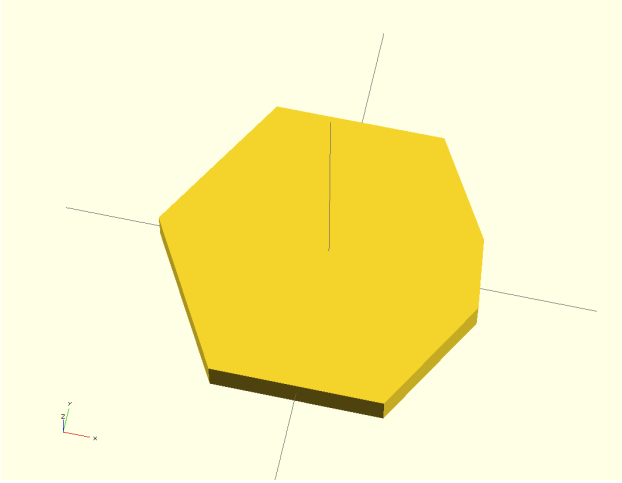
Vertices can be modified with a couple of different modifiers. First, it is possible to define a single cut-off on each vertex. This is done using the keyword ‘lin’. If only one parameter is given this is interpreted as a length to be cut off on both sides, after which both loose ends are connected using a straight line. Note that these new points are not counted as extra vertices, they both stay member of the same vertex. It is possible to specify modification of each vertex by putting them into succession. Note that the modification is applied to all remaining vertices. If you need to stop that process, define the remaining vertices using ‘def’ without parameters.

```
loop(poly=[6,[5]], vertices=[1,3,["lin",1]]);
```



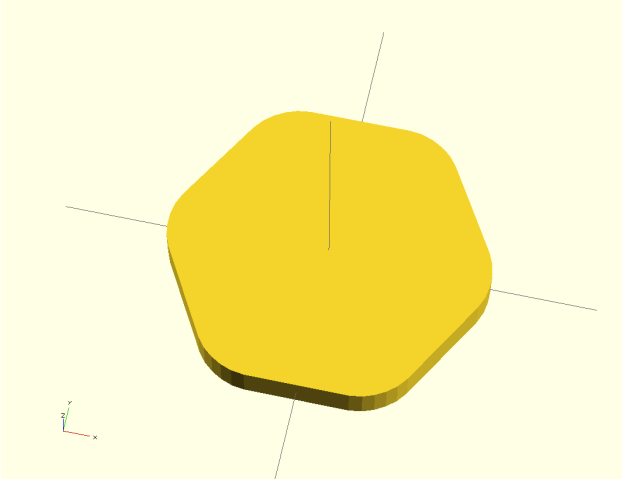
If only special vertices need to be dressed, you can indicated this by putting the vertex numbers in front of the dressing.

```
loop(poly=[6,[5]], vertices=[1,["lin",1,1.5]]);
```



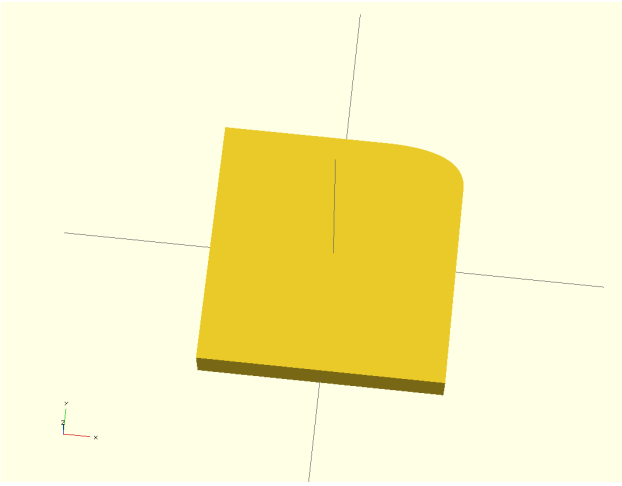
It is not obligatory to cut off the portion of the vertex in a symmetrical manner. You may supply two parameters, which are then used to shorten the edges around the specified vertex with the different values.

```
loop(poly=[6,[5]], vertices=[["arc",1]]);
```



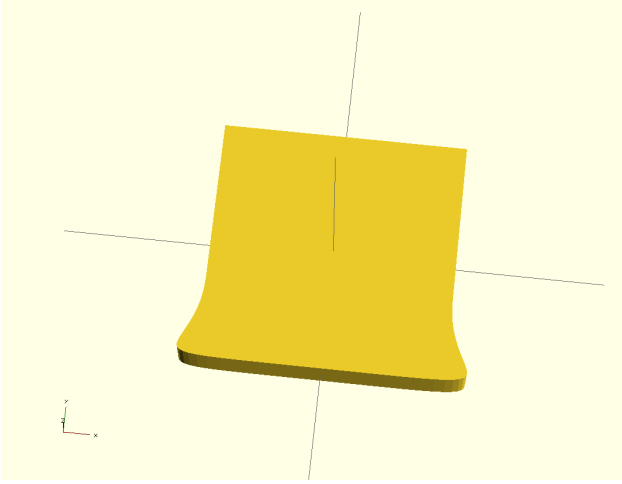
Besides the linear cut-off, it is also possible to fill the space with a curved structure. If one parameter is given, after the cut-off the gap is filled with an arc with constant curvature that is tangent to both sides. This gives a symmetrical rounding off effect.

```
loop(poly=[4,"flat",[6]], vertices=[2,["arc",1,2]], $fn=20);
loop(poly=[4,"flat",[6]], vertices=[2,["arc",1,2,10]], $fn=20);
```



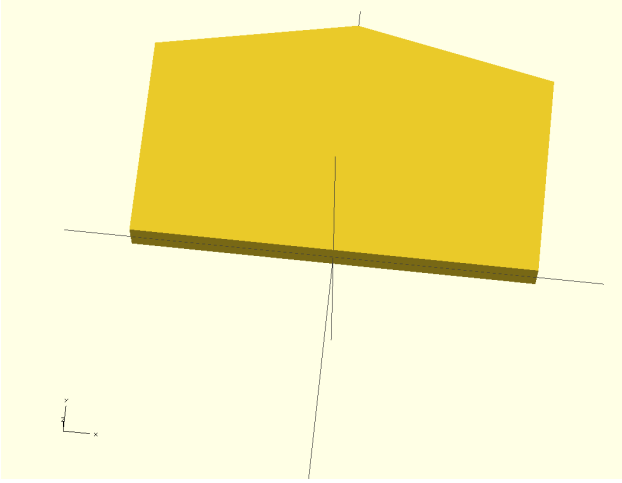
If an elliptic rounding off effect is required, you can specify two different cut off values. An ellipse that is tangent to both edges will be fitted into the space. Since this ellipse is not uniquely defined by both edge ends and their tangents, you may specify an extra parameter, that determines the size of the fitted ellipse. This parameter may be any double, where negative values make it flatter and positive values make it more convex. Note that, if both cut-off values are equal, a zero for the third parameter represents the unique fitting circle.

```
loop(poly=[4,"flat",[6]], vertices=[1,["bez",2,2,2,1],4,["bez",2,2,1,2]], $fn=20);
```



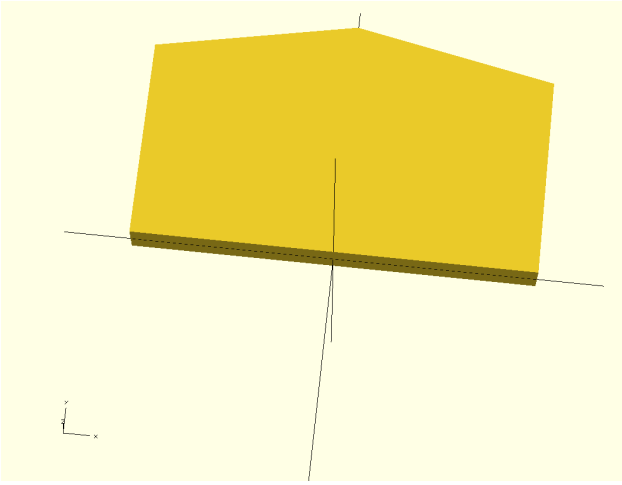
If more than a rounding is needed, it is also possible to define a Bézier curving within the open space after cut off. The fitted curve is a predefined cubic Bézier with the outer points fixed on the ends of the edges and both other points in the extension of the edges. Only the distance from the edge end may be defined (third and fourth parameter), where a 0 puts it on the cut off point and a 1 puts it on the original vertex location. By putting the point even further the vertex may be pulled outside the original figure. Negative values cause inside bending.

```
loop([[-5,5],[5,5],[5,0],[-5,0]], edges=[1,["lin",[0,6]]]);
```



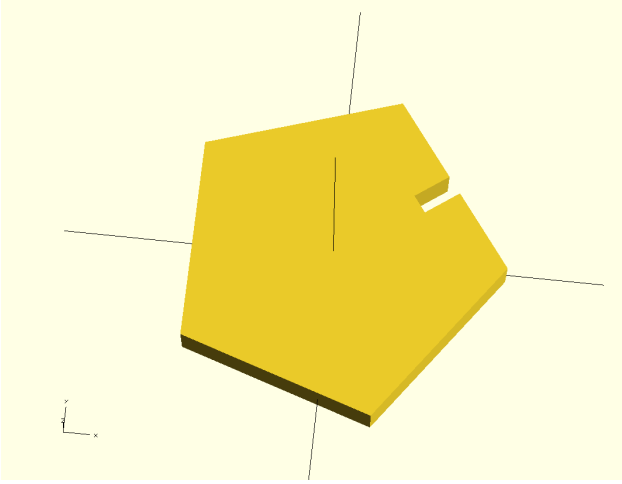
Besides the vertices, also the edges can be modified. The parameters are defined in the same manner, so selection of which edges to modify is done by their numbers before the modifier, and if they are absent, all edges are modified in succession. The default edge is defined by 'def'. By using the 'lin' keyword, extra points can be added to the edge. In the example above, one extra point is added. The edge does not necessarily lie between the points specified with 'points', but are rather the points as they arise after the cut-off of the vertices has been applied. This even holds true if the 'edges' are specified before the the vertices.

```
loop([[-5,5],[5,5],[5,0],[-5,0]], edges=[1,["lin","cen",[0,1]]]);
```



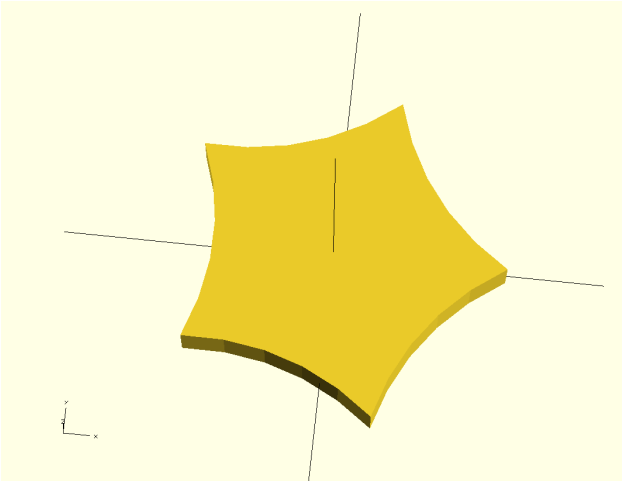
In the former example, the coordinates system is simple, since our edge is horizontal with the center on the Y-axis. So if we want to add a point in the middle one unit elevated, we know we have to pick [0,6]. However, if the edge is not as situated as ideal it is much harder to lift the middle one unit outwards. To that end we may add the keyword 'cen'. In that case the edge is thought to be lying on the X-axis, with its middle in the origin (centralized). Now we can simply define the point as [0,1]. In real space this implies that it will land on [0,6]. So this command and the former give the same result. It is very practical for edges that are not parallel to the X-axis for they are rotated as well.

```
loop(poly=[5,[5]], edges=[1,["lin","csx",[-0.1,0],[-0.1,1],[0.1,1],[0.1,0]]]);
```



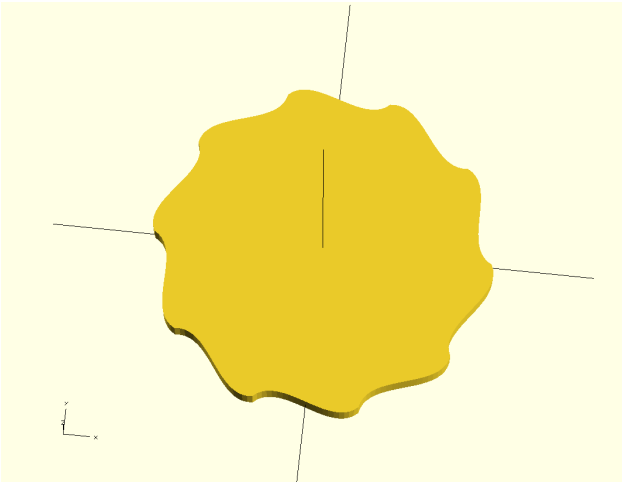
Suppose you need an extra “needle” sticking out distance 1 with a width of 10% of the side. How to do this? For that we have the transformation option ‘csx’. It does the same as ‘cen’ does, so a translation to the center of the edge and a rotation to become parallel to X-axis, but it also scales the edge to lay between -1 and $+1$. Now you can specify relative coordinates from -0.1 to 0.1 and that will always be 10% of the edge length. Note btw that the “needle” is pointing inwards. This is due to the counter clock orientation of the polygon vertices. Add a ”clock” option to poly to correct this. The last scaling option we have is ‘csxy’ which also scales lengths perpendicular to the edge according to the side length.

```
loop(poly=[5,[5]], edges=[["bez","csxy",[0,0.4]]]);
```



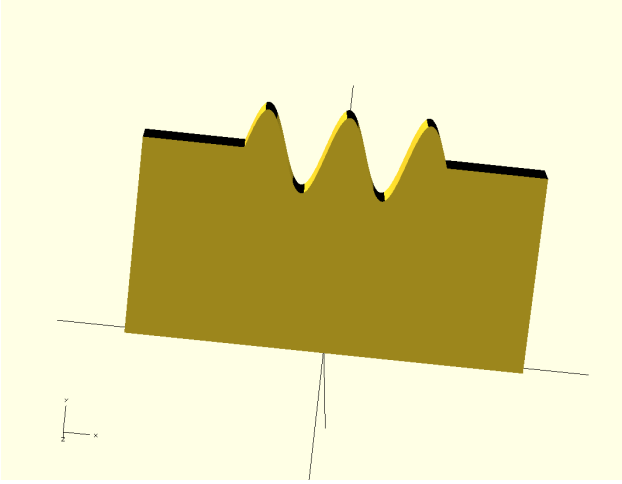
It gets more exiting, when we use the Bézier function on the edges. This is implemented a little differently here. Bézier points themselves are specified, not parameters. The same transformation options as with ‘lin’ are possible. The specified points are interpreted as intermediate Bézier point, the edges ends (after vertex modification) as first and last point of the Bézier curve. Thus, this Bézier curve is not limited to cubic, but is fully of N^{th}-order.

```
loop(poly=[10,[5]] , edges=[["bez","csxy",[-0.5,0.5],[0.5,-0.5]]], $fn=20);
loop([[-5,5],[5,5],[5,0],[-5,0]] , edges=[["bez",[-2.5,7.5],[2.5,2.5]]], $fn=20);
```



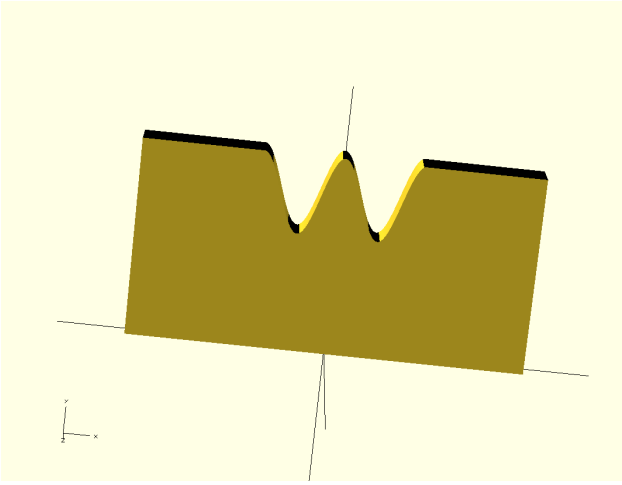
With the Bézier functionality it is possible to make beautiful and bizarre figures. Please note that the resulting figures may not be printable (try the latter yourself) or not even be drawable.

```
loop([[-5,5],[5,5],[5,0],[-5,0]], edges=[1,["wav",5,1,5]], $fn=200);
```



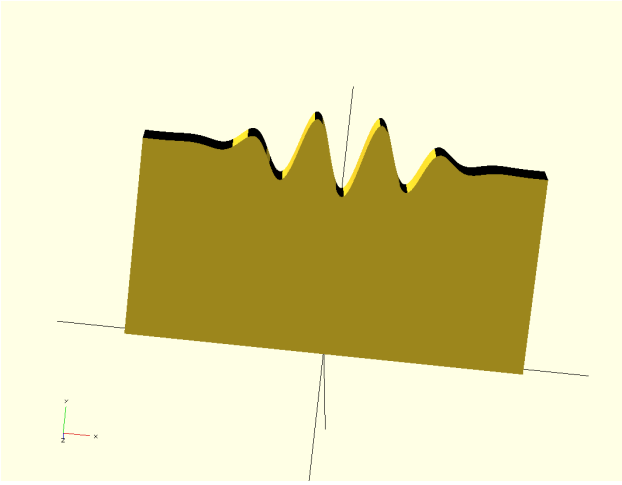
If you need to define a wobbly edge, or have a lot of bumps, using Bézier is possible, but not practical. To that end we can use the keyword 'wav'. It has many parameters of which two are obligatory. Also, all transformations discussed before are usable, for the parameters that are lengths. The first parameter designates the length of the wavy pattern, the second its height. If no other parameters are given, one single bump is drawn, i.e. a centered cosine function of half a period. Note that it may be needed to increase the number of faces quite a bit, otherwise the whole wave may disappear.

```
loop([[-5,5],[5,5],[5,0],[-5,0]], edges=[1,["wav",4,1,4,0,0,-1]], $fn=200);
```



It is possible to shift the height in order to make it nicely fit the edges. In this example we have chosen for 4 extrema (third parameter, note that each side counts as half, due to the shift) and we lowered the whole wave by 1 (-1 in the sixth parameter).

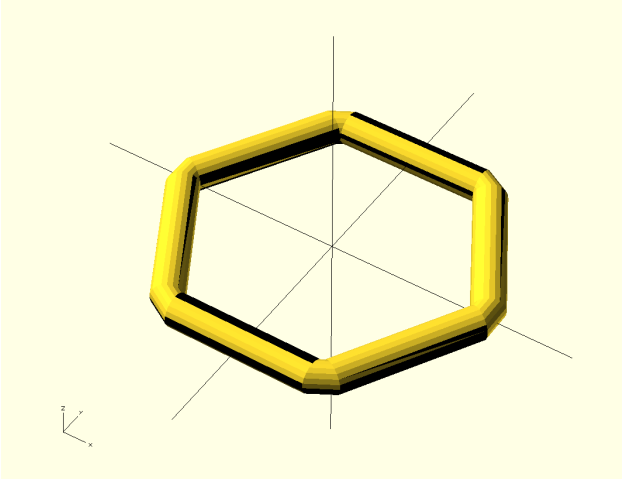
```
loop([[-5,5],[5,5],[5,0],[-5,0]], edges=[1,["wav",4,1,5,2,0.5]], $fn=200);
```



The fourth and fifth parameter denote the envelope and phase shift respectively. If an envelope is used, the width is not rigid any more, but defines the border of a smooth transition between the wave and the edge. The larger the value the steeper the transition. The phase shift of 0.5 corresponds to an full inversion, with other values, you may shift the wave to the left or right.

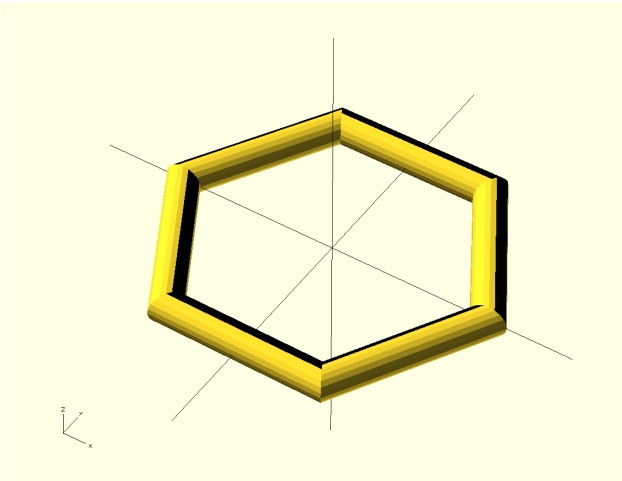
5 Examples, for the 3D loop_extrude() function

```
loop_extrude(poly=[6,[10]]) circle(1, $fn=20);
```



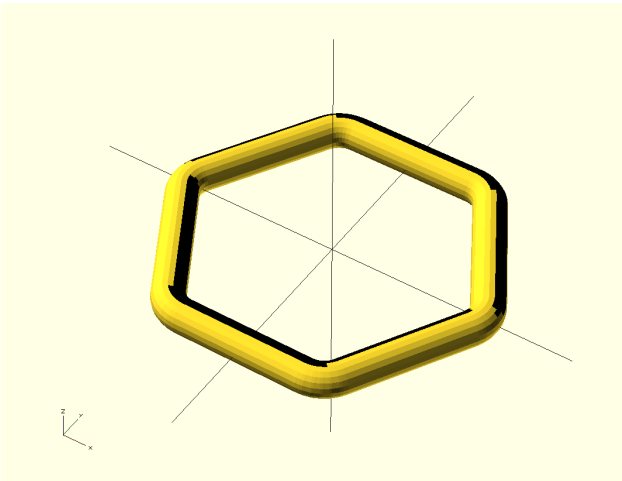
The loop extrusion command extrudes its child along the loop defined with its parameters. The child itself can be any 2D object, including a loop generated by loop(). The parameters of loop_extrude() resemble those of loop() but there are a few differences. These will be discussed below. Just as with the standard extrusion commands, the child is rotated from the XY plane to the ZX plane. As you can see in the picture the extruder must make a choice how to handle the extrusion around the vertices, for the size of the extruded object makes the sharp turn impossible. Therefore the largest dimension is used to shorten the edge size of the extrusion loop, and both open ends are connected.

```
loop_extrude(poly=[6,[10]], segments=[["out"]]) circle(1, $fn=20);
```



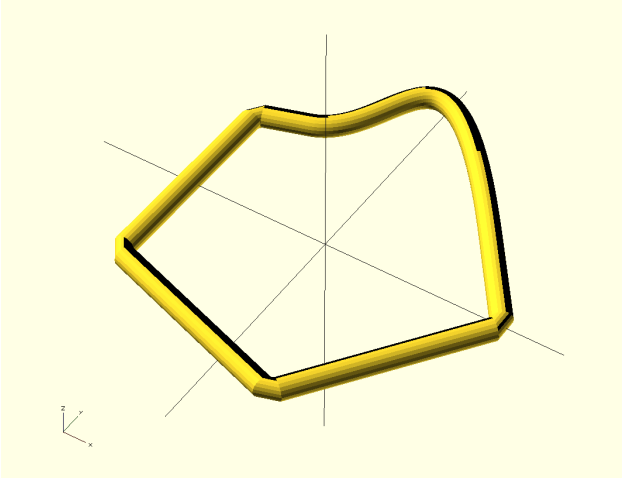
The former example shows the vertices being bent inwards as a result of the size of the extruded object. There are two alternatives to this solution. First, you may define any vertex of the type 'out' so that both edges are extended as far as possible. Note that 3D dressing of the any component of the extrusion loop (vertex or edge) is called a segment, numbered from 1 ... 2N, where 1 corresponds to vertex 1, 2 to edge 1, 3 to vertex 2 etc. So all odd segment numbers represent vertices all even numbers represent edges.

```
loop_extrude(poly=[6,[10]], vertices=[["arc",1]]) circle(1, $fn=20);
```



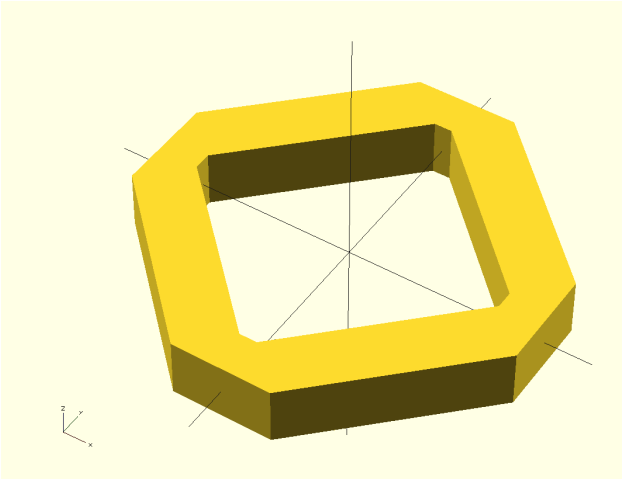
Another possibility to make better corners is to smooth the extrusion curve itself. As the loop_extrude command takes most parameters of the loop command (those which are not supported will be discussed later on) we can define an arc on the vertices. The keyword 'out' on the segments is of no use in this case, and is typically only used to extend sharp edges.

```
loop_extrude(poly=[5,[20]], vertices=[2,["bez",10,15,3,0.5]],$fn=60) circle(1, $fn=20);
```



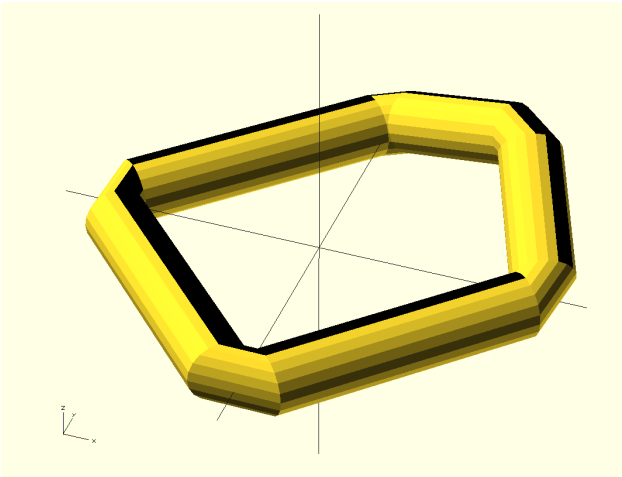
Bézier deformation of the vertices is also possible for loop_extrude() but usually you need a lot of space to guarantee a smooth extrusion. If insufficient space is provided, this may result in a sharp cut off of the corners or the total inability to render the object. Also the default 'lin' keyword is allowed; no example is provided.

```
loop_extrude(poly=[4,[10]], vertices=[],$fn=60) loop(poly=[4,"flat",[2]]);
loop_extrude(poly=[4,[10]], segments=[1,["rel",0.707]]) loop(poly=[4,"flat",[2]]);
```



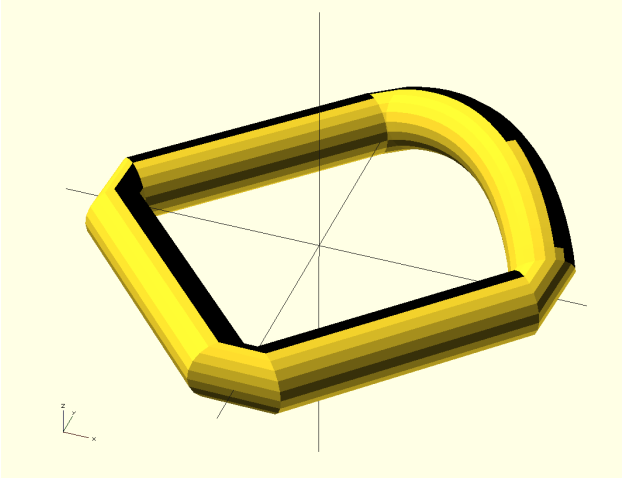
If you extrude a shape with irregular form, the extruder may have difficulty to determine a lower bound on the cut off needed at the corners. It may therefore be too wide (usually it is not to small). This can be corrected by adding a parameter to the keyword 'rel' (default, or to 'in','out' or 'show') or 'abs', in which case the size will be interpreted as absolute minimal allowed curvature. In this example we need to reduce it by a factor $\sqrt{2}$ to make it fit exactly. Try the latter instruction yourself.

```
loop_extrude(poly=[4,[10]], edges=[1,["lin",[5,5,2]]]) circle(1, $fn=20);
```



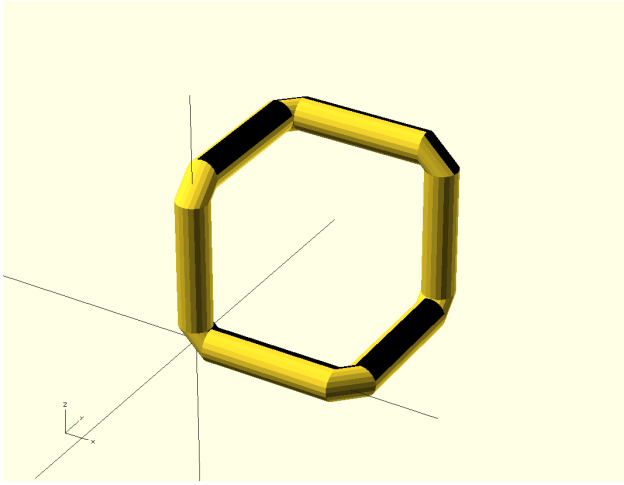
Again, besides the vertices, it is possible to modify the edges. Parameters are like those that are used in the loop primitive, but now there is no need to be limited to two dimensions. So, if we specify an extra point using 'lin', we may also specify a third coordinate. Without any transformation modifiers, these are points in absolute space. As with the vertices, the extrusion is performed as long as physically possible (to not cause internal overlap of the exterior walls) and then subsequently connected by the shortest route.

```
loop_extrude(poly=[4,[10]], edges=[1,["bez",[5,5,3]]], $fn=20) circle(1, $fn=20);
```



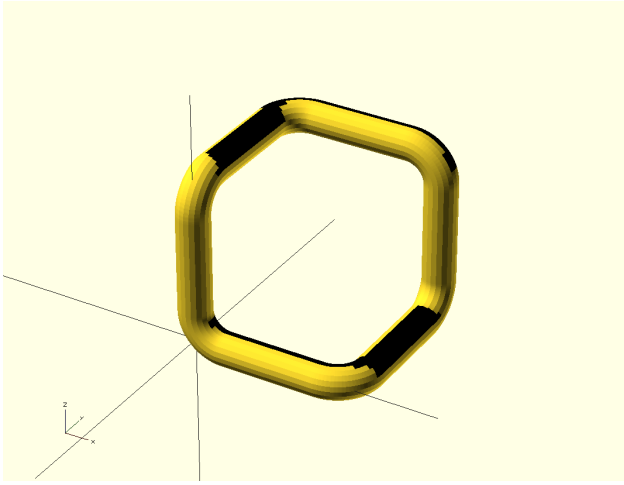
The 'lin' keyword can be replaced by a Bézier 'bez' keyword. In that case the points are interpreted as Bézier points (in 3D space). Adding a few more faces may be needed to get a sufficiently smooth curve. Note that the keyword 'wav' is not allowed in edges, since we could not define a suitable generalization of a wave in 3D without adding a lot of new parameters. That would make working with the option very confusing.

```
loop_extrude(points=[[0,0,0],[10,0,0],[10,10,0],[10,10,10],[0,10,10],[0,0,10]]) circle(1, $fn=20);
```



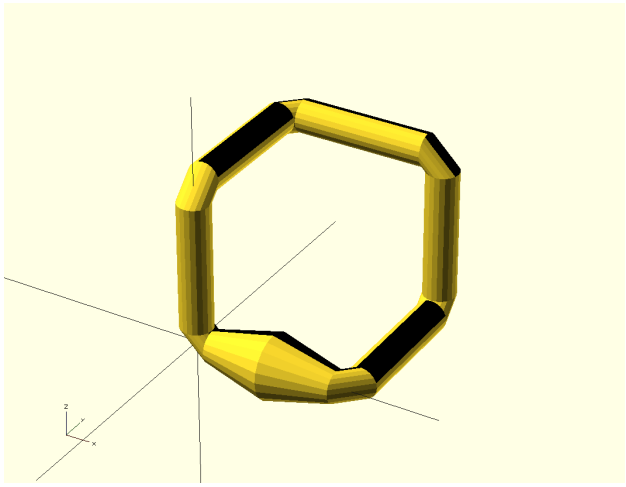
Just as it is possible to use 3D points in the edge you can directly specify which points are visited in the first place. These are in absolute coordinates, per default interpreted as cartesian, but you may use 'cylinder' $[r, \phi, z]$ or 'sphere' $[r, \phi, \theta]$ as well. Note that in the latter case θ is measured from the $x - y$ plane, and not from the z-axis. This is because we want $\theta = 0$ to coincide with the default $z = 0$. Corners are cut as usual, but the 'out' keyword can be used to prohibit that.

```
loop_extrude(
   points=[[0,0,0],[10,0,0],[10,10,0],[10,10,10],[0,10,10],[0,0,10]],
   vertices=[["arc",2]],
   $fn=20)
 circle(1, $fn=20);
```



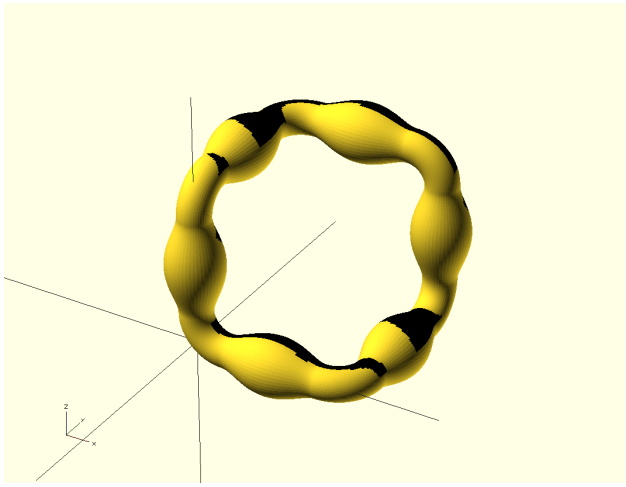
In this case it is also possible to dress the vertices and edges. In order to dress the segments, a moving frame is introduced along the curve. This is a so called Bishop frame, reducing the torsion as much a possible. Usually this makes smooth connection between the beginning and end possible. However, this cannot be guaranteed in all circumstances.

```
loop_extrude(
   points=[[0,0,0],[10,0,0],[10,10,0],[10,10,10],[0,10,10],[0,0,10]],
   segments=[2,["lin","csl",[0,2,0]]],
   $fn=20)
 circle(1, $fn=20);
```



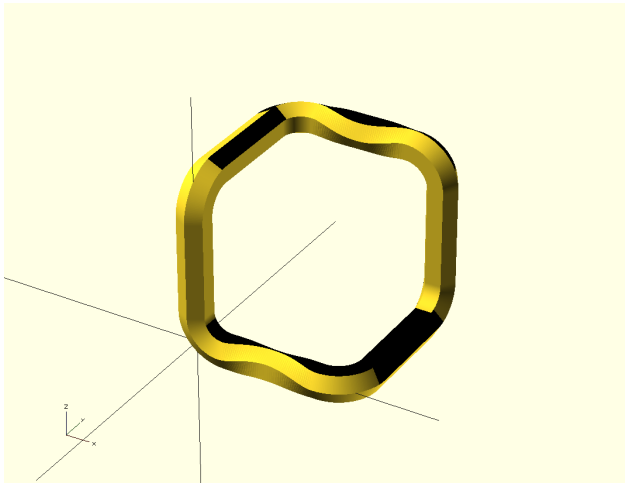
Segments can be separately dressed. In this case you can influence the scale (along the radius) and the angle. Since it is almost impossible to formulate this in absolute coordinates, the segment is translated and rotated (centralized) per default. You can additionally request for 'csl', which then subsequently scales the length of the segment to -1..1. The middle is then easily addressed by a 0. The coordinates for segment modulation are in a moving cylinder frame: $[l, r, \phi]$, with respect to the local curve (which can be curved by itself).

```
loop_extrude(
   points=[[0,0,0],[10,0,0],[10,10,0],[10,10,10],[0,10,10],[0,0,10]],
   vertices=[["arc",2]],
   segments=[2,4,6,8,10,12,["bez","csl",[-0.6,1,0],[0,3,0],[0.6,1,0]]],
   $fn=50)
 circle(1, $fn=50);
```



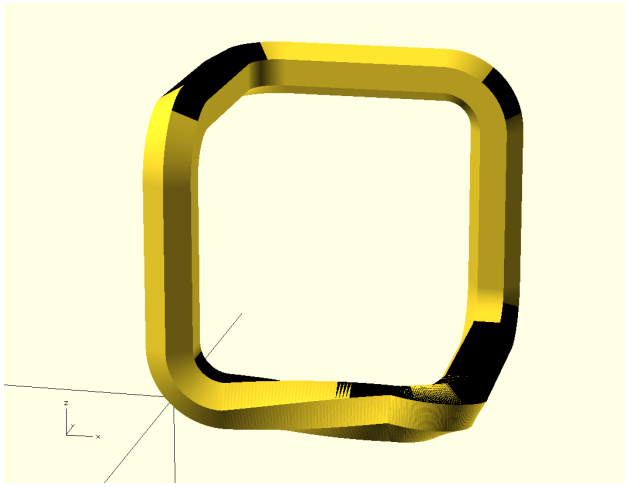
Apart from 'lin' it is also possible to use 'bez'. In this particular example we modulated the scale in such a way the a smooth transition to the rounded corners arises. Note that, if you want to dress all edges, you must specify all even numbers.

```
loop_extrude(
   points=[[0,0,0],[10,0,0],[10,10,0],[10,10,10],[0,10,10],[0,0,10]],
   vertices=[["arc",2]],
   segments=[2,8,["bez","csl",[-0.6,1,0],[0,1,90],[0.6,1,0]]],
   $fn=50)
 loop(poly=[6,[1]]);
```



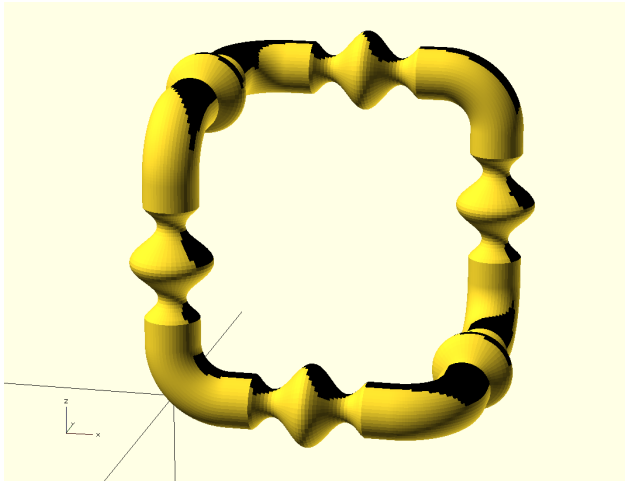
In order to observe the effect of angular displacement (torsion) you cannot use a circle as extruded object, so in this example we made use of a hexagon. Note that, in this example there is no residual torsion between the segments.

```
loop_extrude(
   points=[[0,0,0],[10,0,0],[10,10,0],[10,10,10],[0,10,10],[0,0,10]],
   vertices=[["arc",2]],
   segments=[2,["lin","csl",[-1,1,0],[0,1,30],[1,1,60]],3,["lin","csl",[-1,1,60],[0,1,30],[1,1,0]]],
   $fn=50)
 loop(poly=[6,[1]]);
```



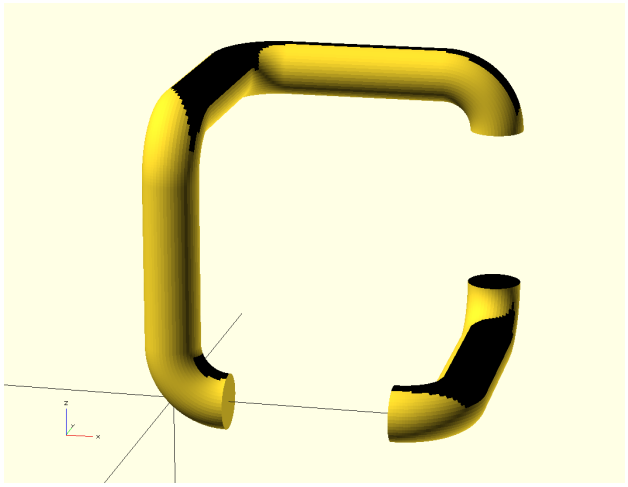
If you need an ongoing bending or scaling, just apply this torsion to every segment, and making sure the borders (located at $l = -1$ and $l = 1$) of the segments are explicitly defined. If they are not defined they are implicitly set to [-1,1,0] and [1,1,0] to ensure continuity. Depending on the form of the curve, at one joint continuity problems may arise due to built up torsion.

```
loop_extrude(
   points=[[0,0,0],[10,0,0],[10,10,0], [10,10,10],[0,10,10],[0,0,10]],
   vertices=[["arc",2]],
   segments=[2,4,6,8,10,12,["wav",4,0.5,3]],
   $fn=50)
 circle(1, $fn=50);
```



The last modifier that can be applied to the segments is the wave. It only affects the scale (so the angle cannot be modulated), but it enables one to modulate the width with an ongoing sine wave. Parameters are as with the two dimensional loop, but now the default value of the height equals 1 instead of zero (scaling 1 results in the identity transformation).

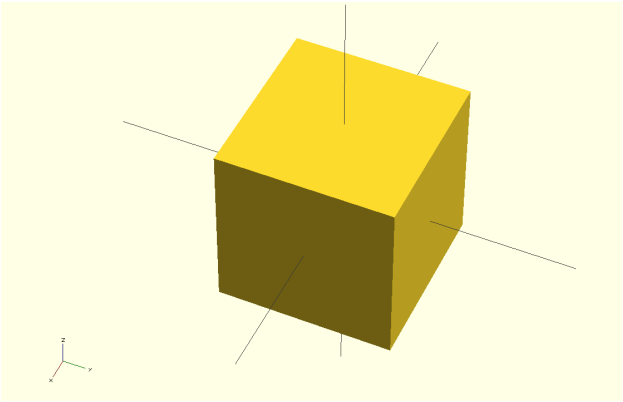
```
loop_extrude(
   points=[[0,0,0],[10,0,0],[10,10,0],[10,10,10],[0,10,10],[0,0,10]],
   vertices=[["arc",2]],
   segments=[2,6,["hide"]],
   $fn=50)
 circle(1, $fn=50 );
```



If you do not want or need a complete closed loop, you may cut out any segment you do not want by using the 'hide' keyword. Open ends are automatically closed by a plane.

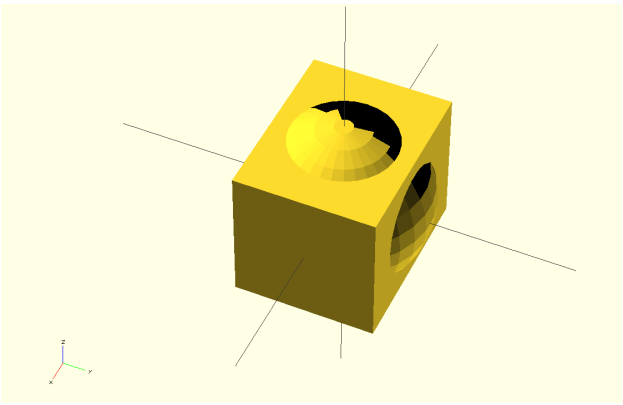
6 Examples, for the box() function

```
box() sphere(r=10);
```



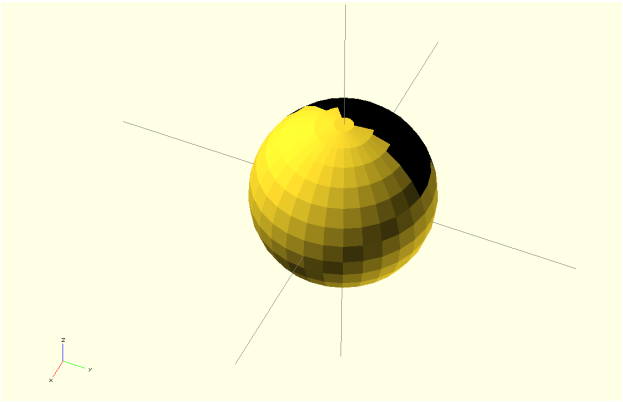
Basic use of the box() function is to replace the object with the its rectangular bounding box. So in this example we thai the long route to define an cube that is not even precisely 10 units long (because of the render accuracy. The main purpose for this function is to use it for complex objects of course. For example to be able to cut a region out of an other object. It is much faster than hull(); Note that the underlying object itself is removed.

```
box(add=-2, xadd=2) sphere(r=10);
sphere(r=10);
```



It is possible to shrink or enlarge the boxing box is all directions ('add') or in one specific direction. Values are added per direction. To make this clear we redraw the original object. Other directions are 'yadd' and 'zadd', which all may be used simultaneously.

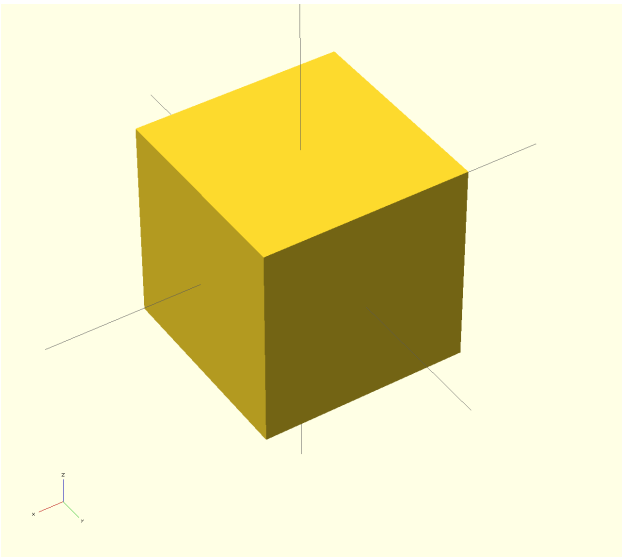
```
box(act=false) sphere(r=10);
```



The last option, which may seem rather pointless is 'act'. If it is set to true, box() works as usual and if set to false, it just passes the child, or a union of the children, on. The idea is that with this option it is possible to define a module for an object once, and choose between the object itself or its outer boundaries,for example for a cut-out. See a more sophisticated example below.

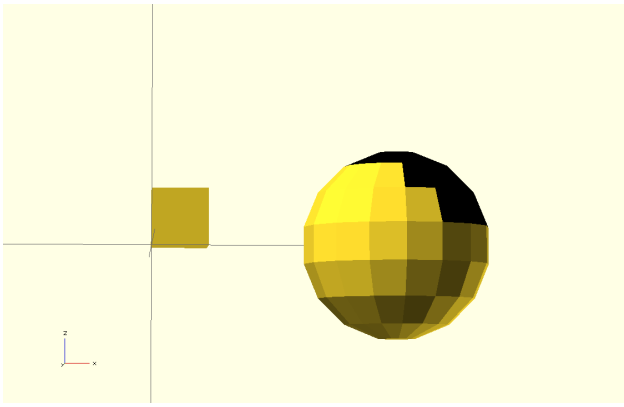
7 Examples, for the position() function

```
position(xmid=0, ymid=0, zmid=0) cube(10);
```



With the position() function you can quickly place an object somewhere without knowing its location. Normally you would use translation() for placing object, but this requires knowledge of the objects own location. Per coordinate you can choose between 'min', mid', max'. So if you quickly want to place an object in the center use the construction above, or choose other values to place it at that point. If you use 'xmin', for example the leftmost side will be placed to that coordinate. So with position(xmin=0, ymin=0, zmin=0) the object will placed in the first quadrant.

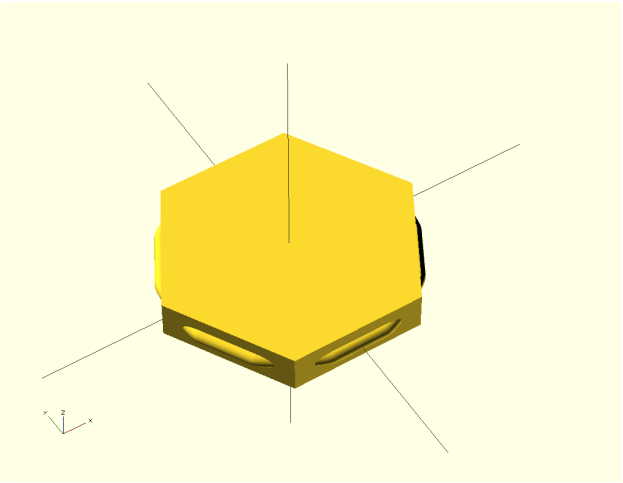
```
position(xmin=5, keep=true)
{ sphere(5);
  cube(3); }
```



The position() command can operate on two objects. The last object will be used as reference for the placement of the first. In this example the sphere is placed with it leftmost side to right of the cube with a distance of 5. Note that normally the object of reference will be removed from scene unless we use the 'keep' option. If 'xmax' was used the sphere was placed to the left of the cube, but due to the shift of 5, would result being covering the cube. In that case you need a negative value.

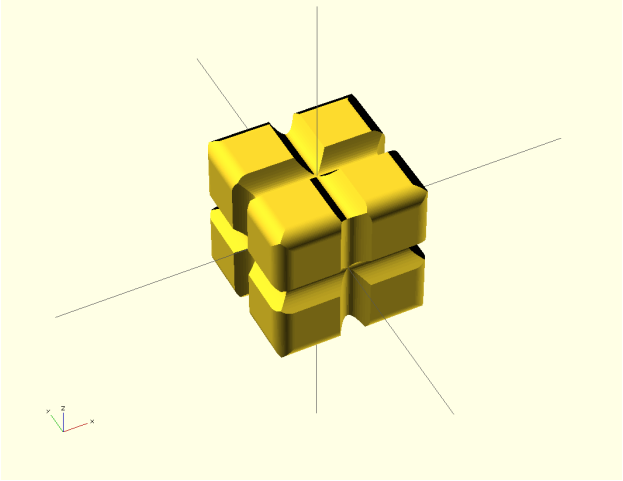
8 Other examples.

```
union()
{ loop_extrude(poly=[6,[10]], vertices=[["arc",3,3,-10]]) circle(0.5);
  linear_extrude(height=3, center=true) loop(poly=[6,[10]]); }
```



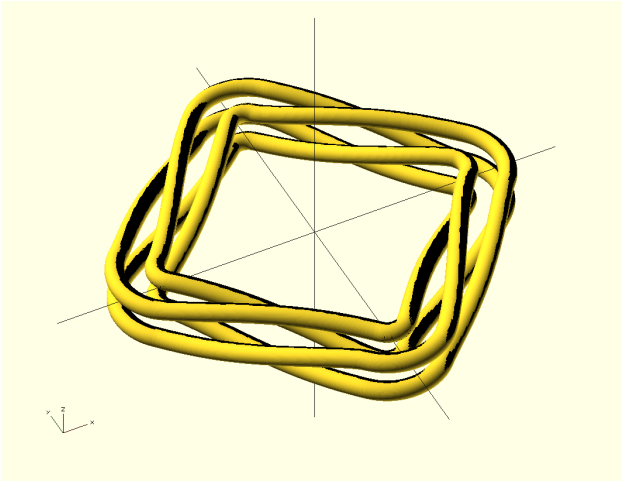
Hexagon with tabs.

```
$fn=20;
module base(axis)
{ rotate(90,axis)
  linear_extrude(height=10, center=true, convexity=3)
  loop(poly=[4,"flat",[10]], vertices=[["arc",1]], edges=[["wav",2,1]]); }

intersection()
{ base([1,0,0]);
  base([0,1,0]);
  base([0,0,1]); }
```



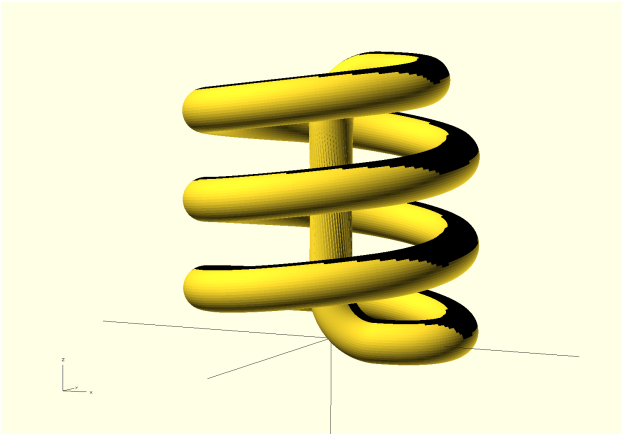
Cube with groove and partially rounded edges.

```
$fn=50;

loop_extrude(
 poly=[4,[30]],
 vertices=[["arc",4]],
 segments=[
  2,["bez","csl",[-1,1,  0],[-0.8,1, 0],[0,1, 45],[0.8,1, 90],[1,1, 90]],
  3,["bez","csl",[-1,1, 90],[1,1, 90]],
  4,["bez","csl",[-1,1, 90],[-0.8,1, 90],[0,1,135],[0.8,1,180],[1,1,180]],
  5,["bez","csl",[-1,1,180],[1,1,180]],
  6,["bez","csl",[-1,1,180],[-0.8,1,180],[0,1,225],[0.8,1,270],[1,1,270]],
  7,["bez","csl",[-1,1,270],[1,1,270]],
  8,["bez","csl",[-1,1,270],[-0.8,1,270],[0,1,315],[0.8,1,360],[1,1,360]]] )
 union()
 { translate([ 2, 2]) circle(1);
   translate([-2, 2]) circle(1);
   translate([ 2,-2]) circle(1);
   translate([-2,-2]) circle(1); }
```



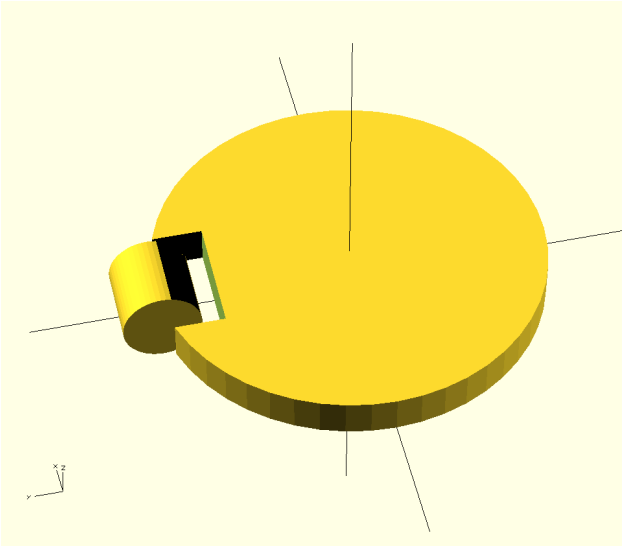
My garden hose, somehow i can never find the beginning.

```
$fn=50;

loop_extrude(
 points=["cylinder",[0,0,0],
   [8,0,0],[8,90,1],[8,180,2],[8,270,3],
   [8,0,4],[8,90,5],[8,180,6],[8,270,7],
   [8,0,8],[8,90,9],[8,180,10],[8,270,11],
   [8,0,12], [0,0,12]],
 vertices=[["arc",5.65],1,15,["arc",2.3]])
 circle(1);
```



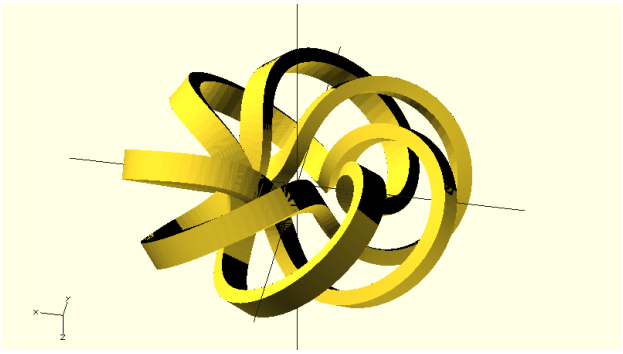
Use of cylinder coordinates to draw a helix.

```
$fn=50;

module rotor(marge)
{  box(add=marge, act=(marge>0))
   position(xmid=0,ymin=-1,zmid=0)
   { child(0);
     child(1); } }

module emplace(marge)
{  union()
   { rotor(0)
     { child(0);
       child(1); }
     difference()
     { child(1);
       rotor(marge)
       { child(0);
         child(1); } } } }

emplace(0.5)
{ rotate([0,90,0]) cylinder(h=2, r=1);
  cylinder(r=5, h=1); }
```



Use the box() and position() functions to place a rotor in a plate at unknown location.

```
$fn=50;
r=163/150;

function p(t) = [ 3*(2+cos(7*(3*t)/3)) , (3*t) , 3*sin(7*(3*t)/3) ];
function q(t) = (t%2==0) ?
  ["lin" , "csl" , [-1,1,t/2*r] , [1,1,t/2*r]] :
  ["lin" , "csl" , [-1,1,(t-1)/2*r] , [1,1,((t-1)/2+1)*r]];

function pnts(t) = for(i=[0:t-1],p(i));
function segs(t) = for(i=[0:t-1],q(i));

loop_extrude(points="cylinder"+pnts(360) , segments=segs(720)) square([0.8,1.6], center=true);
```



Use vector generation and addition to make space curves