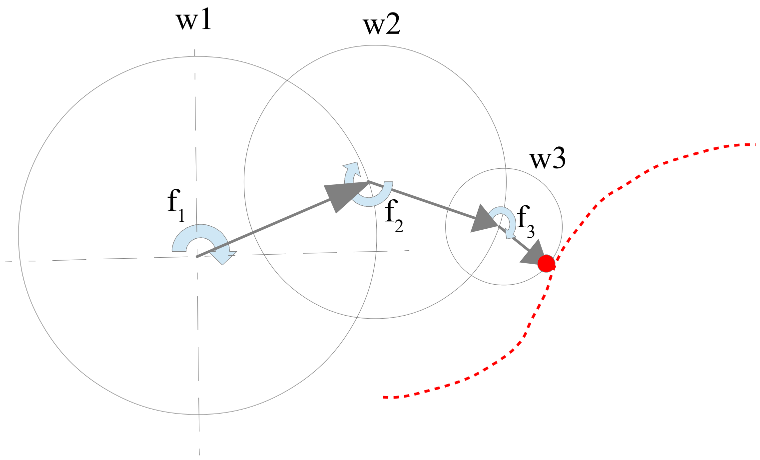
Gwheels a g-code pattern generator

Introduction

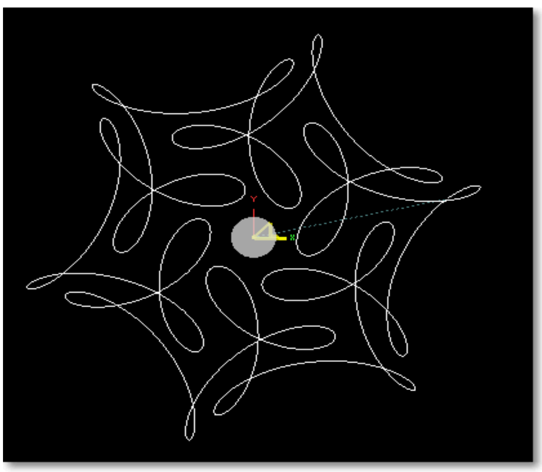
Imagine a sequence of wheels where the first wheel rotates around a fixed axis and centre of the second wheel rotates around the circumference of the first, and the centre of the third wheel rotates around the circumference of the second and so on. The first wheel rotates with a speed value of f_1, the second with $-f_2$ and the third with f_3 . All frequencies are multiples of a common base rotation in the clockwise direction with a speed of 1 rpm. So if f_1 is 6 then wheel w1 is rotating clockwise at 6 rpm. The speed of w2 is negative because it is travelling in the opposite (anticlockwise) direction.

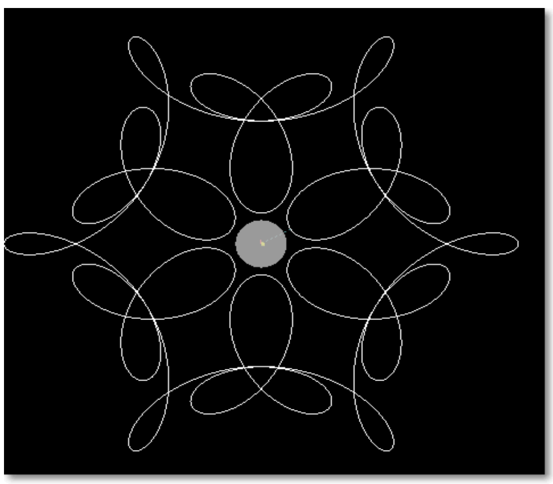


Choose any point on a wheel to specify the initial position of the centre of the wheel that is rotating around its circumference. On the last wheel choose any point on its circumference and this point will move along a path in the plane as the wheels rotate here denoted by the red dotted line.

This path through space was shown by F. A. Farris "Wheels on Wheels on Wheels—Surprising Symmetry," *Mathematics Magazine* **69**(3), 1996 pp. 185–189 , to have rotational symmetry.

The particular path traced out is dependent upon three values for each wheel. These are ; the radius of the wheel, the speed of the wheel and the starting position which can be defined as the angle made by the radius with the x axis. This gives rise to an infinite number of paths see below.





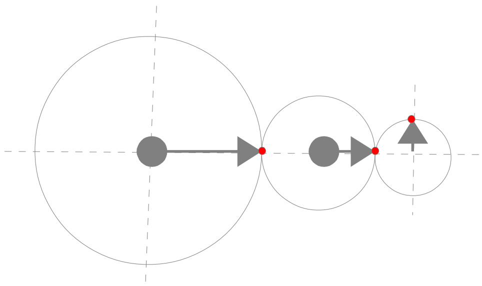
The program

My program allows you to create any number of wheels using the ***Create_Wheel*** function. You must supply the three values defined above namely radius, phase angle (for starting position) and speed. The first wheel defined is the base wheel and then each subsequent wheel will rotate around the circumference of the prior wheel. The variable **#<_Count>** is initially set to zero and then automatically incremented by the ***Create_Wheel*** function.

```
(Create Wheels collection                 )
(param #1 is the radius of the wheel      )
(param #2 is the phase of the wheel       )
(param #3 is the speed  of wheel rotation )

#<_Count> = 0
o<Create_Wheel> call [10] [0] [1]
o<Create_Wheel> call [5] [0] [7]
o<Create_Wheel> call [3.33] [90] [-17]
```

The code above creates three wheels as in the diagram below



Once created we can create the pattern by calling the ***Generate_Path*** function which requires us to specify 6 parameters.

1. The starting angle in degrees
2. The angle to step between points, also in degrees
3. The angle to end the pattern. This may be more than 360 degrees and you can either specify this yourself or alternatively use the ***Nturns*** function which calculates the number of full rotations required to complete the pattern and returns this value in the global variable ***#<_result>***. This value must of course be multiplied by 360.
4. The feed rate to use when moving along the path. In the program fragment below this value is stored in a variable called ***#<Feedrate>*** but may also be written as a number directly.
5. The depth of cut.
6. A phase angle which may be used to rotate the pattern around the centre.

A typical program fragment is shown below. Notice that the flag #<_UseRotary> is tested to see if the inverse time mode is to be used. That is we expect that this flag is set to one if a rotary table is being used and to zero if the standard XY movement is used instead.

```
(****************** Now calculate path *******************************)
o<main1> if [#<_UseRotary> EQ 1]
      g93  (Inverse time mode )
o<main1> endif

o<NTurns> call
#<nturns> = [#<_result>]
g61
G10 L2 P1 X0 Y0 Z0 (ensure that G54 is set to machine zero)
g90
o<MoveToSafeHeight> call

(param #1 is start angle                 )
(param #2 in angle increment             )
(param #3 is end angle                   )
(param #4 is feedrate                    )
(param #5 is depth of cut                )
(param #6 is phase                       )
o<Generate_Path> call [0] [0.1] [#<nturns> * 360 ] [#<Feedrate>] [2] [0]

M2
```

You can of course include any g-code in this main fragment for example a loop to cut to a specified depth in a number of steps as illustrated below.

```
#<final_depth> = 5
#<depth_inc> = 1
#<depth> = [ #<depth_inc> ]
#<end_angle> = [#<nturns> * 360]
o<depthloop> while [ #<depth> LE #<final_depth>]
   o<Generate_Path> call [0] [0.1] [#<end_angle>] [#<Feedrate>] [#<depth>] [0]
   o<MoveToSafeHeight> call
   #<depth> = [ #<depth> + #<depth_inc> ]
o<depthloop> endwhile
```

There are a number of values and flags you may set to **before** calling the ***Generate_Path*** function which are useful in controlling the program.

```
#<_XScale>              = 4.9 ( x scale factor                      )
#<_YScale>              = 4.9 ( y scale factor                      )
#<_ZScale>              = 1   ( z scale factor , I never use        )
#<_Safeheight>          = -2  (Safe height                          )
#<Feedrate>             = 120 (  Feed rate                          )
#<nturns>               = 1   ( number of spindle turns required )
#<_UseProfile>          = 0   (not using Z profile                  )

#<_UseRotary>           = 0   (we are not using a rotary machine )
#<_Use_Z_for_Depth>     = 1   (using w axis for depth               )
#<_Use_Rad_Phase>       = 1   ( specifying radius and phase         )
```

The variables #<_Xscale> , #<_Yscale> and #<_Zscale> may be used to enlarge or reduce the size of the resultant figure in order to fit it within a required area without having to adjust the wheel sizes. So if the pattern is 40mm wide and you want it to fit 80mm then set the #<_Xscale> value to 2.00. Setting the value to 0.5 would result in a pattern 20mm wide. The Y and Z scales act in a similar way.

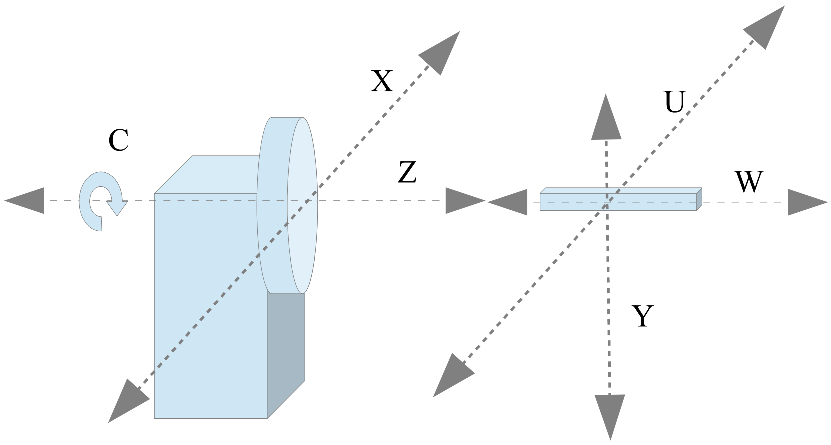
The #<_Safeheight> and #<Feedrate> variables allow you to set your machine parameters globally.

The #<_UseProfile> flag is set to one if you want to cut the pattern over a non flat profile. As this functionality has not yet been implemented leave this value at zero.

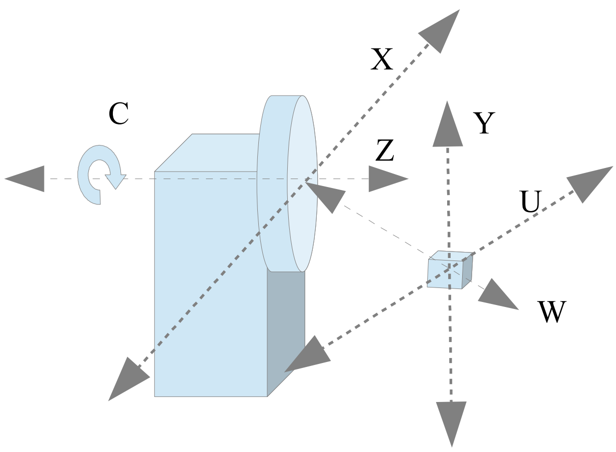
The #<_UseRotary> and #<_Use_Z_for_Depth> flags are used to help define the machine you are using.

Description of my machine in its current version

My machine is a rose engine lathe / mill hybrid. It has a headstock and a tool post and each can move in three directions independently from the other.



The headstock can move linearly along the X and Z axes and rotate around the C axis. The tool can move linearly in the U (parallel to X) , W (parallel to Z) and Y (vertical) axes. I can also rotate the tool post so that the U and W axes are not parallel to X and Z.



This allows me a great deal of flexibility. Looking at the upper figure I can use (X or U) and Y as a vertically mounted table and either Z or W for depth as a mill with a vertical work table (here #<_UseRotary> flag is set to 0).

Or alternatively I can use C as a rotary table (X or U) for radial movement and (Z or W) for depth (here #<_UseRotary> is set to 1 or any number greater than 0).

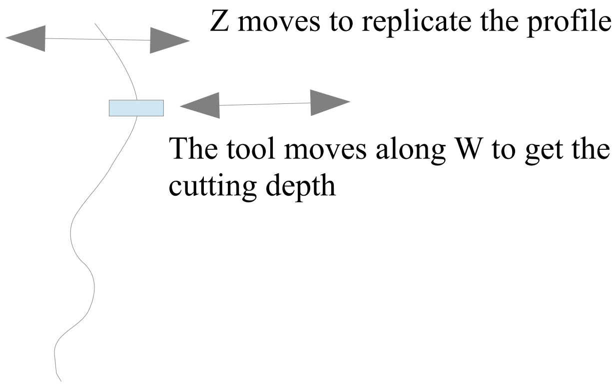
Used as a CNC based rose engine I use X for the standard rosette movement, Z for the pumping movement and C for rotary movement, W for tool depth movement. To represent this the

#<_UseRotary> flag is again set to 1

So to repeat for a standard mill us #<_UseRotary> flag set to zero and for a rotary table use #<_UseRotary> flag set to one. The difference is essentially between using standard Cartesian coordinates (X,Y) as on a mill or Polar coordinates (R,Θ) as with a rotary table.

I guess the question you are thinking is why have Y at all? The reason is to enable me to cut several patterns around the rim of a platter simply using the C axis as an indexing mechanism so that having cut a pattern using X and Y, I can rotate the work around to the next position. Again this can be done on a standard mill with a rotary table placed on the mill bed. Remember that this machine is a result of a development path starting with a rose engine lathe. If I where to start again I would seriously consider using a cnc frame like the ones that hold routers, but with a rotary base.

Regarding the Z and W axes my intentions are to make movements along a profile separate from tool depth movements so when doing faceplate work I will dedicate Z to profile movement and W to tool depth movement.



When using this arrangement I need to specify both Z and W axes in a linear move. The #<_Use_Z_for_Depth> flag when set to 1 indicates this case.

When #<_Use_Z_for_Depth> is set to zero the depth and profile movements are combined into one Z movement.

On my machine the positive direction of the W axis on the tool post is moves the tool away from the work so that for me W depth cuts are negative values (my W axis acts like a standard mill Z axis so maybe I should swap the Z and W axes around)

However the positive Z axis is towards the tool post. So Z for tool depth values are positive numbers .

When simulating profiles positive Z values will result in a concave profile when moving against a stationary tool, whilst negative Z values will result in a convex profile.