# Flywheel Builder Internals

This document explains how the flywheel builder webpage works in case someone wants to enhance it or simply fix a bug. There’s a bit of math here, but mainly geometry. This document will try to explain the algorithms including a few references to the source code for context.

## Overall Design

There are two design points to the application that aren’t absolutely required, but explain why it is written the way it is. First, the intention was to produce an application that has few, if any, installation requirements. The web page provides this both by being accessible directly on the internet so it can be visited without any installation whatsoever, as well as being standalone in the sense that the HTML file can be downloaded (Save As…) and used on a PC, tablet, or phone, without further installation effort.

The second design point is that the application doubles as a document that not only shows dimensioning, but also includes customized instructions for building the particular flywheel constructed from the knobs provided to the user. This second design point is a bit experimental, but seems useful. The alternative is to clearly label all the dimensions and refer to them in an external instruction guide.

Note that a guide is needed either way. But for now the how-to guide is primarily about how to work the knobs to design a flywheel. This app will produce the instructions for building that flywheel, so no build discussion is needed in the how-to guide.

## The HTML

The document part of the application begins at the front of the file as a web browser would expect. To meet the first design point listed above, the file does not reference any other file in order for it to operate with no installation requirements. Some canvas HTML elements are used for the figures, with large ones near the beginning of the document, and some small figures dispersed throughout the instructions that are generated. A form exists under the main figures and includes a table that organizes most of the input fields. All the input fields are given ID=”value” so they can be referenced by the code.

### Conditional HTML

HTML doesn’t directly have conditional text, but there are many parts of the instructions that are dependent on whether the flywheel has 3 vs. 4 hole cutouts, or whether the spokes are straight or curved. Thus, a mechanism for conditional HTML was invented for this purpose. Currently there are two conditions implemented, and these were just cited as examples. HTML tags can have class names, and this is leveraged to create the conditional text. These class names can be seen in the inline style sheet at the beginning of the file. Classes of “showForStraightSpoke,” “showForCurvedSpoke,” “showFor3HoleSpoke,” and “showFor4HoleSpoke” are defined. The stylesheet declarations aren’t really needed, but are there as a reminder.

The conditional HTML is implemented by the Javascript function **chooseConditionalHTML(HTMLclass1, HTMLclass2, pickfirst)**. The if-then-else concept was implemented by declaring classes in pairs, and it can be seen that this function is called twice, once to pick straight vs. curved, and once to pick between 3 vs. 4 hole. The function selects between a pair of class names, one of which is to be displayed, and one of which is to be hidden. Under the covers it enumerates through all the document tags with class name to be displayed and sets the “display” attribute to block, inline, list-item as appropriate. In other words, the document text will be displayed normally if it is enabled. For the class name to be hidden, the “display” attribute is set to “none” and this appears to be correctly rendered by all mainstream browsers.

The net is that when writing the HTML, one of these magic four class names can be given to a tag and all the content of the tag, including nested tags, will appear or disappear as needed. This helps create instructions that are tailored precisely to the flywheel being constructed. The reader need not be concerned with curved-spoke instructions when the plan shows a straight-spoke design, for example.

More pairs of conditional classes can be trivially defined by calling the chooseConditionalHTML() function with the pair of class names and the condition that selects which to enable. They don’t need to be declared in the stylesheet, except as documentation to the programmer.

### HTML Substitutions

The generated HTML instructions need to refer to specifics of the flywheel design to make the procedure more understandable. Conditional HTML was discussed in the previous section. Here we discuss how specific values are substituted into the text.

Text substitutions are made by the **doHTMLSubstitutions(HTMLclass, obj)** function. This function will iterate through all the tags with the given HTML class *HTMLclass* and substitute values from the given Javascript object *obj*. Currently this is called only once, declaring that any tag with class=”flydata” should be substituted with data values from the **fly** object. The fly object is discussed below, and it contains all of the calculated values for the flywheel design.

A substitution will replace the body (text) of a tag with a formatted value from a field of the object. The most common tag used for these substitutions is the bold <b> tag, although other tags are suitable, including <span> which is a generic tag that doesn’t change properties of the text.

The selector of what to substitute is encoded in the tag’s id attribute. The best way to describe this is through an example:

<b class="flydata" id="innerSpokeThickness:length">0</b>

In this example the bold tag is given the “flydata” class name. The doHTMLSubstitutions() Javascript function will iterate all tags with the “flydata” and will parse the id attribute to lookup a value. The id attribute is specified as an object field name, in this case *innerSpokeThickness*, and a formatting keyword, in this case *length*. The flydata substitutions use the fly object, so fly.innerSpokeThickness will be substituted. The length formatting tells it to format this as a number with 2 or 3 digits after the decimal point, depending on whether it is mm or inches (Note: the code could add “mm” or “in” to the substitution, but currently doesn’t). Look at the implementation of this function for other formatting options.

There are a couple of things to note about this substitution. First, the example shows the number 0 as the content of the bold tag. Any text, including nothing, can go here as it will be completely replaced by the substitution operation. Second, the id value may not be unique in the document. That is, an id=”innerSpokeThickness:length” could appear many times in the document if it wants to refer to the inner spoke thickness number. The id attribute is defined in HTML to be unique, but current mainstream browsers do not seem to care. If this becomes a problem, a prefix or suffix of “unique” characters could be added. For example, it could be id=”foo:innerSpokeThickness:length” and the code could ignore “foo:.” For now this is NOT done, but if future browsers get picky, it may need to be fixed.

Currently only basic string substitutions are performed. The table substitutions are coded manually in Javascript near the end of buildFlywheels() just after the calls to set conditional HTML and perform the above general substitutions. In the future it might be possible to have “spokeTable:table” with a special “table” formatter that can produce a table. But for now they are coded manually. The same is true for figures. Javascript code looks up canvas references by id and draws into a figure.

## The Javascript

The code is standard JavaScript which initializes as the document is loaded. Other than some minor initialization (variable assignments), the real action begins with the onLoad=”buildFlywheels()” that is part of the document <body> tag. Note that any change to the form also calls buildFlywheels(). That is, anytime anything changes, the entire flywheel is recomputed and all figures are redrawn. The browser is plenty fast for that even on a smartphone or tablet.

## The URL

In order to “save” a flywheel, the application automatically updates the URL to include settings from the user-controlled knobs. The entire list of settings is tracked in the *urlstatemap* array in the code. This not only lists the form field names that need to be saved, but it also provides shortened names that are more suitable for a URL. While a person can fairly easily reverse-engineer the URL and tweak it manually, the intention is that the app does this automatically. The genhash() function is called to produce the URL blob after the # (hash) sign, and the last thing buildFlywheels() does is update location.hash with this value. This tells the browser to update the URL.

The first thing buildFlywheel() does is call parsehash() to fetch this part of the URL unless a hidden field of the form already has this saved. As a side-effect, parsehash() fills in the form field with the values. The hidden field keeps this from happening every time a value is changed. Remember that buildFlywheel() is called every time a field is updated. There may be better ways to code this.

A side-effect of all of this is that presets now can be implemented as the part of the URL past the hash. Note that the value for each option in the preset input form <select> object is the part of the URL after the hash.

The main intention of updating the URL like this is so that the browser can bookmark a specific flywheel design. Because no file is updated, this is the only way to “save” a design. Therefore, even if the HTML file is downloaded and used without the internet, bookmarks are required to save designs. Note that these URLs can also be e-mailed, posted on forums, etc, provided the URL points to a web site that is accessible to the receiver of the URL. A clever user can also hand-edit the URL in the browser to replace the text after the # (hash) to refer to a different copy of the FlywheelBuilder.html file.

## Adding to the Preset List

The preset list is a **<select>** object in the HTML with a description for each preset and a value that represents the URL parameters. If this object is changed by the user, the onchange attribute calls the selectedPreset() function. This trivial function assigns the current hash and resets the hidden urlhash input field. As a side-effect of changing the form, the buildFlywheel() function will also be called and this will load parameters from the newly updated URL. That fills in the form table with the values from the preset flywheel.

To add a new flywheel to the preset list, simply design the new flywheel and copy/paste the URL after the # (hash) and use it to add a new option to the **<select>** list. It’s that simple.

## The High-Level Code

As mentioned, buildFlywheels() is where the action is. It performs these steps every time anything changes:

* Reload the form from the URL unless this has already been done (i.e., urlhash is already set)
* Do conversion if the units have changed between imperial/metric. This updates the form fields so the user can see all the converted values immediately. It also tracks *lendigits* which is the number of digits for rounding of values. Unfortunately, FlywheelBuilder isn’t really smart enough to show significant digits properly, so it shows 3 decimal digits for inch measurements and 2 decimal digits for mm measurements.
* Fetch all the values from the input form fields
* Do a specific check if the number of spokes has been changed. If the inner spoke angle happens to be perfectly between spokes we previously had a single inner spoke hole. If this is the case with the old number of spokes, the angle is adjusted for the new number of spokes so there remains a single inner spoke hole. If the angle is not centered, it will be left as-is.
* Call calculateFlywheel() to produce a fly object that contains all the measurements (and more) that we need for drawing and producing instructions. **The algorithm is described in detail below.**
* Fetch the canvas objects and draw multiple versions of the flywheel to show what it looks like, and to show various measurements on how it will be constructed. The drawFlywheelDemo() function draws the first figure which contains few, if any, measurements. The intention of this figure is to show the look of the flywheel since that is very important for model engines! The drawFlywheelDims() function draws a second figure and adds all sorts of dimension markings.
* The doConditionalHTML() and doHTMLSubstitution() functions are called to display conditional HTML with substitutions. This was discussed earlier.
* Some tables are produced. These can be seen in the instructions that are generated. The number of rows depends on the number of spokes for each table.
* Finally, figures for steps of construction are drawn. Functions such as drawSpokeStep(n, …) are called to draw a series of figures to help visualize how to operate the rotary table correctly to align the flywheel for cuts on the mill. Figures are generated based on curved vs. straight spokes, under the assumption that straight spoke figures are hidden when curved spoke instructions are displayed (and vice-versa).

## Utility Math Functions

Javascript is a nice language that allows objects to be defined. In this application this is leveraged to some degree to, hopefully, make the math a little more understandable. A *point* object represents an obvious x,y pair. A point can represent either an actual point, or it can be interpreted as a *difference* (+x, +y) from a point. There are a few functions to manipulate points:

* toPoint(x,y) creates a point object. For example: mypoint = toPoint(29,53). A point can also be interpreted as a difference, for example (-5,5) might represent going back 5 in X and forward 5 in Y.
* ptStr(p) produces text from a point for debug. For example: “(29,53)”
* addPoints(p1,p2) will literally add the x and y values of the two points. In this case, one of the points (doesn’t matter which) must be a difference otherwise it doesn’t make much sense. It returns a new point rather than modify one of the input points.
* subtractPoints(p1,p2) is the same as addPoints, except the first point is subtracted (this is odd and likely to be fixed).
* scalePoint(p1, multiplier) will scale both x and y by the multiplier. It returns a new point.
* distance(p1, p2) calculates the distance between two points. This is simple math – just google it.
* midpoint(p1, p2) calculates a point exactly between two points. Again, simple math. It’s really the average x,y of both points.
* angle(p1, p2) calculates the “angle” between two points. In the drawing an angle of zero is “up”. So if p2 is directly above p1 (in Y) the angle is zero. If p2 is directly below the angle is 180 degrees. If p2 is to the right of p1 the angle would be 90 degrees. Calculating the angle is a matter of first calculating the difference of X and Y. These form the base and height of a triangle, where the hypotenuse is the line connecting the two points. We can easily use trig to calculate the angle of the hypotenuse (answer: it’s the arc-tangent).
* rotate(p, angle) is a key function. Think of a situation of a rotary table centered on the mill and we consider that position 0,0. Now suppose we have a point x,y. Now rotate the RT by angle degrees. Where would x,y move? That’s the point that is returned. If you draw up the triangles of how it moves you’ll understand the math. But that’s what it does.

## Drawing Functions

The HTML canvas object has some nice drawing abilities, including the ability to scale and rotate. However, there is a catch. If the code leverages these facilities, text will also be scaled and rotated, and for this project that isn’t desired.

To handle this, the code has a function makeDrawingContext() that creates an object that includes the canvas itself, the context (you need to read about canvas to understand this), the offset to the center of the canvas (width/2 and height/2), and a scale factor. The scale factor represents the flywheel radius. Actually, the flywheel radius is drawn as 90% of the canvas size, so it is scaled to that value. So if we have a 3.5 inch flywheel, a radius of 1 inch will be scaled out to produce a 2 inch circle, but scaled down to 90%. A 1.75 inch radius will be 90%. This allows the actual sizes to be drawn so they come out proportionally correct. It doesn’t matter if the units are inches or mm as everything is scaled as a proportion.

In common to most of the drawing functions are color, fill and pattern. The color is obvious the color we are going to draw. The fill is another color that fills inside the thing being drawn, and this can be *fill\_none* which means we don’t fill anything (transparent). Pattern is for drawing dotted lines for objects. A few patterns are defined including pattern\_solid (normal solid line), pattern\_dot, and patter\_dash\_dot\_dot. Others may be defined, too.

* drawCircle() draws a circle at the given center point with the given radius.
* drawRadialLine() draws a line from a center point to a distance with an angle. It’s used primarily for dimensions. The center is usually 0,0 but that isn’t required.
* drawline() draws a line from one point to another
* drawArc() draws an arc given a center point, radius and start/end angles. Zero is up for the angle.
* drawRadialDim() will draw a dimension marking that shows a line from a center point out for a distance and will place a given label at a position along that line.
* drawAngDim() will draw a dimension marking that shows two lines from a center point, an arc between, and an angle label that shows the angle. The angle is given as a text string and can be wrong ☺.

The reason the code has all these functions is that it helps prove the math is right. If we can draw the flywheel correctly, the dimensions must be ok. That’s the concept.

## The Calculations

The calculateFlywheel() function does all the “thinking” in the app. It produces the data needed to draw and machine the flywheel. This is returned as a single *fly* object. Here we describe how it works.

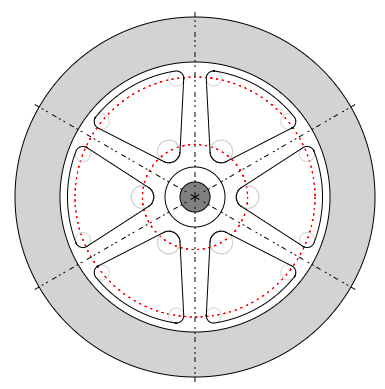
The calculateFlywheel() function mainly calculates all the values needed for a single spoke of the flywheel. Once that is done, it is assumed these values can be tweaked specifically for each of the N spokes. That is done through rotation of the values. The spoke that is calculated is the one that would be considered the “first” spoke and is up and perhaps to the right on the flywheel. The other N-1 spokes are identical, except that they are rotated. With the rotate(p, angle) function, creating actual dimensions for these other spokes is trivial.

ToDo: There should be a function/method on the fly object that can hide these per-spoke calculations. For example: spkinfo = fly.getSpoke(n) could calculate and return an object that has all the dimensions for spoke n rotated properly. The code that draws the flywheel will be simplified slightly by this.

The first thing calculateFlywheel() does is construct the fly object with many of the input parameters as fields. This keeps everything together. Then it calculates a spoke.

### Spoke Hole Calculations

A spoke is defined by the four holes surrounding it.



leftOuterCenter

leftInnerCenter

rightOuterCenter

rightInnerCenter

Figure 1: four holes define a spoke

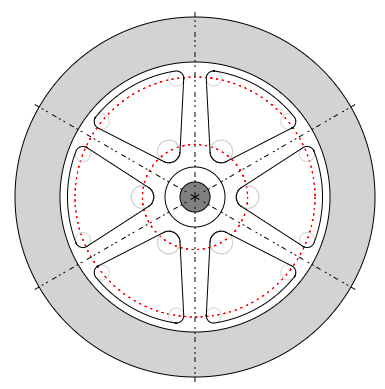
There are two pairs of holes, an inner pair near the hub and an outer pair near the rim. The orientation of the template spoke is such that the spoke is vertical. That is, the inner holes are horizontally positioned such that the spoke will be “pointing up.” For a straight spoke, the outer holes are also horizontally oriented and are situated above the inner holes. For a curved spoke the outer holes will be rotated clockwise (more on this later).

Calculation of the center points of these holes (leftOuterCenter, leftInnerCenter, etc) is straightforward. Consider what is done to position these holes on an RT. The software works with X and Y axes just as is done with a mill. To calculate the inner hole centers the first thing that is done is to create a point at X=innerHoleDistance and Y=0. The innerHoleDistance is the distance from the hub center to the hole center and is represented by the inner red dotted line in the figure. This point is located directly up from the hub where the axis line of the spoke intersects with the red dotted circle. To correctly calculate the point, it is “rotated” as if turning an RT until it is either on the left (negative rotation) or on the right (positive rotation). The angle is called innerHoleAngle and the rotation is done with the rotate() function mentioned earlier. That’s it. The outer hole centers are calculated similarly. Note that these inner and outer distances and angles are inputs provided by the user in the form table.

### Straight Spoke Cutline Calculations

Next are calculations when the spokes are straight. This is the math that produces settings that are useful to the machinist as they are non-trivial to calculate manually. But they aren’t as difficult as one might think. To understand these, think in terms of a mill’s XY table and an RT that can rotate.

What are needed are calculations for two cut lines as shown in the following figure.



leftCutEdgeEnd

leftCutEdgeStart

rightCutEdgeStart

rightCutEdgeEnd

The calculated lines are defined by their endpoints right/leftCutEdgeStart/End. A very close approximation for these endpoints could be used by taking the center points of each hole and shifting it right or left (in X) by the radius of each hole. All of those values are known. This would be close, but not accurate because these lines are at angles. The figure shows these as very nearly vertical lines, which is common, but if the line slopes heavily it will intersect the holes at significant angles. The following figure exaggerates this concept to show the orientation and diameters of the holes impact the cutline endpoints. If the cutline endpoints were placed directly to the left on a horizontal line from the center of the hole it would be probably be close enough for machining spokes. But this can be calculated exactly.

cutline

The calculation is done in three steps and the angle() function described earlier is used in the steps. The simplified figure above gives a hint how this might be done. If a line is calculated between the centers of the holes (both centers are known), and perpendicular lines are created within each hole that extend for the radius of each hole (both radii are known), then the endpoints of the cutlines are at the ends of these two perpendicular lines, as shown in the figure.

The first step uses the center points of the inner/outer holes. This calculates the rightHoleCenterAngle which is only the first step and not the answer. The distance is also calculated between these center points and recorded as rightHoleDistance.

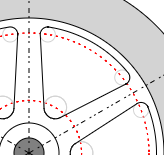
Now we do something sneaky for the second step. To simplify the math, we “pretend” the flywheel is rotated so this line between the centers of the holes is vertical. Here is the simplified figure of what this looks like.

original axis

cutline

rotated axis

And the following figure is a reminder of what this looks like on the flywheel.



rightStart

rightEnd

rightHoleDistance

In this rotated view of the world the centerline between the hole centers is always precisely vertical. Further, we setup the axis so that 0,0 is the center of the inner hole. In this rotated world the end points of the cut line are trivial to calculate. The rightStart for the inner hole end of the line is at (-innerHoleRadius, 0) and the rightEnd for the outer hole end of the line is at (-outerHoleRadius, rightHoleDistance). But this line lives in a rotated world. All we need to do is rotate it back. We rotate it back by the angle of the centerline between the two hole centers that we calculated earlier (rightHoleCenterAngle).

Finally the third step must account for the fact that 0,0 was shifted to the center of the inner hole. To produce the final rightCutEdgeStart and rightCutEdgeEnd, the centerpoint of the inner hole is added. Said another way, because 0,0 was shifted to the center hole, the calculations of the line in the previous step was relative to the center of the inner hole. Simply add the hole’s coordinates to get the actual coordinate.

Now that the two endpoints for the right cutline are known, the cut angle is calculated with the angle() function given these two points. All these calculations are repeated for the left spoke, of course. One final thing that is done is to adjust the angle to a rounded number of digits. This is done once so any rotation calculations made on the spoke doesn’t produce odd roundings later on.

## Curved Spokes

There are a few key ideas that need to be understood for curved spokes. First, the practical matter of explaining it to the user leverages the idea of grabbing the flywheel with a mighty hand and twisting the metal clockwise while the hub is held firm. The angle twisted is called the *curve angle*.

Second, is the fact that the resulting flywheel needs to be reasonably machinable. Add to that fact that there are enough knobs already in this application, so the mechanism for twisting the curve has many constraints. One who designs a flywheel by hand can create many more variations than are possible with this application (consider drawing with a French curve, for example).

A serious consideration is that the goal is to not require CNC. Thus, any given curve should be a uniform radius that can be cut using a rotary table. This rules out the French curve idea. Also, there are two sides to each spoke and it would be nice if the radius of the cut was the same for both sides to simplify setup. This is added as an additional constraint.

### Algorithm to Construct the Curve

As the user “twists” the rim of the flywheel by adjusting the curve angle, the software must calculate a curve that fits the inner/outer spoke holes. There are many ways to make a curve that fits, including using straight lines, which would not be acceptable. Here is what the flywheel builder currently does.

It is possible to compute the radius of a circle given 3 points that intersect the circle. The algorithm leverages this fact. First a simplified algorithm will be described, but a slight variation was added to make better curves and this will be described after the basics are understood.

Three points are needed. The basic algorithm chooses these three points that will form the centerline of the arc that forms a spoke:

1. The midpoint between the left/right centers of the outer holes on either side of the spoke. Again, this is easily calculated as the centers of these holes are known.
2. The midpoint between left/right centers of the inner holes on either side of the spoke. This is easily calculated as the centers of these holes are known.
3. The center of the hub. This is known.

A

B

arc center

C

This algorithm produces a fairly nice spoke curve. However, as the rim is twisted, the tightness of the circle becomes acute as the arc must reach ever further clockwise around to meet the outer hole. The constraint of the first point, the center of the hub, adds greatly to this stress. A variation of the algorithm was needed that would adjust the first point to the left to ease this tension. The algorithm eases this tension by considering that a twist halfway between what would otherwise be straight spokes to be a 100% twist, and the first point is shifted to the left by half the distance to the inner hole radius at this point. This is done in a proportional manner. The twist is not constrained so the shift can indeed to beyond half the inner hole distance. There is a point where the twist is absurd and produces a thin, perhaps non-existent, spoke. This occurs even if this variation of the algorithm is not applied. This adjustment won’t be shown in figures in this document as it is slight, but it can be seen when adjusting the curve angle in the application.

The first step for this algorithm is to calculate the circle defined by points A, B, and C. This is done by solving a system of three circle equations. The code adapted the solution from <http://2000clicks.com/mathhelp/GeometryConicSectionCircleEquationGivenThreePoints.aspx> so that anyone who really wants to review their advanced high school math can do so. It’s icky, but results in something that is very easy for a web browser to calculate even though it’s a pile of operations. This produces the center of the circle at (h,k), and most important of all it produces the radius r.

So we have an algorithm to produce a circle that traces the centerline of a spoke. Next we need to define the outside edges of the spoke. These will be the cutlines. The radius of the centerline will be used to define these edges.

### Curved Spoke Cutline Calculations

This is the tricky part. We have computed the centerline of a spoke and therefore have a radius. The centerline isn’t terribly useful, other than perhaps to show it in a drawing, but it won’t help with the mill. We need the arcs that form the edges of the spokes.

A

B

C

Consider the right side of the spoke. The left is calculated similarly. We want an arc that intersects the outer edge of the inside hole and extends to intersect the outer edge of the outside hole. There are many arcs that can do this, but we will choose to fit an arc that has the same radius as that calculated for the centerline. This radius, r, is already calculated in the code.

The solution for calculation of this arc is similar to the solution for calculating a straight spoke. We can’t simply use the center point (h,k) of the circle already calculated. It is close, but cannot simply be shifted into place. So we do some math to find a circle that fits. Remember we already decided the radius will be r, so that’s a partial step to the solution.

As with the straight spoke, the calculation is much simpler to comprehend when viewing the flywheel from a rotated angle. In this case, the rotation will show the inner and outer hole centers rotated such that they are perfectly side-by-side on the X-axis. We will use the radius r calculated earlier simply because it is known and we assume it will produce a nice looking spoke. We also have the calculation of the distance d between the center points of the holes. The distance() function calculates this.

Here is a figure that shows this rotated world. The arc center is not known yet, but r and d are known. Note carefully that the arc radius goes all the way through the center of each hole to the opposite side. This is the right side of the spoke. For the left side we’d aim the arc radius for the center of each hole, but it would end when it touches the hole. Because the holes can be different sizes, the triangle is tipped one way or the other. It’s tipped in the direction of the bigger hole in this case of the right side of the spoke.

arc center (unknown point)

r

r

d

inner hole

outer hole

Note the triangle formed by the center points of the two holes and the arc center. We make the observation that we know the radius of each of the two holes. This allows calculation of all three sides of this triangle. We already know one side, d. Call the other two sides r1 and r2.

arc center (unknown point)

r2 = r – hole radius

r1 = r – hole radius

d

inner hole

outer hole

Of course each hole radius is different, but they are known so know we can calculate r1 and r2 and with d we know all sides of this triangle. We need to calculate the arc center and to do this right triangles will come in handy. We’ll calculate the arc center relative to the center of the outer hole. Once we have calculated this arc center, we can rotate it properly and offset it so we know where the true arc center is relative to the flywheel hub. Going one step at a time, drop a line straight down from the arc center. Call the outer hole center (0,0) for now. Call the arc center (x,y) which will be calculated.

(x,y))

r2

r1

h

b

a

(d,0)

(0,0)

inner hole

outer hole

Note that there are two triangles and they are right triangles so we can apply the Pythagorean Theorem. So…

r12 = a2 + h2 and r22 = b2 + h2  But we can rearrange these as r12 – a2 = h2 and r22 – b2 = h2 .

Since both equal the same thing, we know r12 – a2 = r22 – b2 . We also know d = a + b, which means that b = d – a. We can substitute d – a in for b in the previous equation to get this mess:

r12 – a2 = r22 – (d – a)2

But this isn’t as messy as it seems, because the only unknown in that equation is a. We can solve for it. I’ll spare the drudgery and get to the punchline. Feel free to do the algebra on your own. It’s high school algebra and now you know why you need it ☺.

Now note that this is really the x coordinate of the center of the radius! We now go after h because it will be the y coordinate, and we can apply the Pythagorean Theorem again to find it:

r12 = a2 + h2 which means h2 = r12 - a2. And therefore,

As we’ve observed, the (x,y) of our center is really (a,h). All we need to do to finish this off is to rotate by the angle between the centers of the inner and outer holes, plus another 90 degrees because we laid out this center line on the X axis instead of Y. Finally we add the point relative to the center of the outer hole because that was 0,0 when we did the calculations. This gives us the true (x,y) coordinate of the center of the arc the forms the cut line on the right side of the spoke. Calculation of the start and end angles for this arc is trivial. We use the angle from the arc center to each of the inner and outer hole centers.

This completes the work for the right side of the spoke. The left side is very similar. We are calculating the center of an arc that touches the “near” side of the spoke holes for the left side. For the right side which we just finished we went to the “far” side of the spoke holes. So, for example r = r1 + outerHoleRadius. This gave us r1 = r – outerHoleRadius. For the left side we have r = r1 – outerHoleRadius because we are stopping at the near side. So we need to use r1 = r + outerHoleRadius. It’s a minor, but very important, adjustment to the calculation. Otherwise the rest is the same.

### Misc Calculations

Finally a few miscellaneous calculations are made to assist the drawing functions later on. These aren’t needed for running the mill.

First, the edge inner and outer arcs are calculated. These are simply the cutlines between the outer holes and, optionally between the inner holes if they are separated (a single inner hole is typical and therefore the arc is empty). This isn’t exciting for a machinist. The arc radius is known as it was specified as an input in the form. The angles between the two holes is also known.

Finally, four little drawing arcs are calculated. These represent the edges of the holes that remain after the web is machined out. These four arcs, along with the edge inner and outer arcs, will draw the outline of the web cutout. Like everything else with the spokes, they need to be rotated for each spoke position. Note that the web that is defined here is clockwise from the main spoke. That is, it is to the right of the spoke.