

A Gimp plugin: drawing a parametric curve as an approximate Bézier curve

Version 2.1

July 2021

Introduction

These pages contain some instructions for a Gimp plugin *Parametric curve*. The plugin draws a parametric curve approximately as a composite Bézier curve (Gimp's path). When installed, the plugin is found in Gimp's menu at

<Image>/Filters/Render/Parametric curves/...

In fact, there are three plugins:

- *Parametric curve (cartesian)*,
- *Parametric curve (polar)*,
- *Parametric curve (read function from file)*.

I shall talk a little about each, though the first two are easy to use even without any instructions if one just knows what parametric curves are.

The central goal when designing the plugins was to construct a reasonably accurate approximate Bézier curve with only a sparse set of control points. To this end, the plugin first tries to find on the curve some special points, such as cusps and inflection points and some others, and then it creates an initial subdivision of the curve according to those points. In this task the user can help as will be explained below.

Creating the initial subdivision is the first of the two big phases in the work of the plugin. The second is the actual approximation part: each subarc in the subdivision is approximated with a composite cubic Bézier curve. The two phases pose quite different problems. I don't discuss here how the problems are solved in the plugin; instead, I just try to make using the plugin easier.

Contents

1	<i>Parametric curve (cartesian)</i>	2
2	<i>Parametric curve (polar)</i>	5
3	<i>Parametric curve (read function from file)</i>	6
3.1	Extra parameters	8

1 *Parametric curve (cartesian)*

We take first the basic one of the three plugins. It draws parametric curves given in the form

$$\begin{cases} x = x(t) \\ y = y(t) \end{cases} \quad (a \leq t \leq b). \quad (1)$$

This is a parametric curve (parametric arc) defined by means of two functions $x(t)$ and $y(t)$ and an interval $[a, b]$.

The GUI contains the following fields (though not literally quite as we show them here).

curve name	name of the curve
x(t)	the function $x(t)$
y(t)	the function $y(t)$
start t	the starting value of the parameter t
end t	the ending value of the parameter t
closed?	is the curve supposed to be closed?
fit in window?	should the size of the plot be adapted to the window?
padding	if Yes above, how much padding around the plot
x of the origo	the x coordinate of the origo in Gimp's window
y of the origo	the y coordinate of the origo in Gimp's window
scale	scaling factor to Gimp's coordinates
draw the axes?	should the coordinate axes be drawn?
custom values of t	user-chosen parameter values to force anchors
messages	should info be displayed of the running of the plugin?

Before explaining this all in detail, we take an example.

Example 1.1 As an example, a three quarters of a circle can be drawn as a Bézier curve with center at (500,500) and radius 100 (in Gimp's coordinates) by inserting the following inputs.

curve name	3/4 circle
x(t)	cos(t)
y(t)	sin(t)
start t	0.
end t	3*pi/2
closed?	No
fit in window?	No
padding	0
x of the origo	500
y of the origo	500
scale	100
draw the axes?	No
custom values of t	<empty>
messages	No

Note that some of the inputs are supposed to follow Python's syntax. Available are all names in Python's standard mathematical library, such as `pi` (meaning π) and functions like `cos` and `exp`.

We describe now the meanings of the fields.

curve name

This is the name that will appear in Gimp as the name of the path.

x(t) and y(t)

The parametric curve we are drawing (approximately) is supposed to be given in the form

$$f(t) = (x(t), y(t)), \quad t_0 \leq t \leq t_1, \quad (2)$$

meaning a function $f : [t_0, t_1] \rightarrow \mathbb{R}^2$ where $[t_0, t_1]$ is a real number interval. The $x(t)$ and $y(t)$ are functions $\mathbb{R} \rightarrow \mathbb{R}$. Thus, the circle above is thought to be given as the function

$$f(t) = (\cos t, \sin t), \quad 0 \leq t \leq 3\pi/2, \quad (3)$$

hence $x(t) = \cos t$ and $y(t) = \sin t$, and $t_0 = 0$ and $t_1 = 3\pi/2$.

start and end values of t

The start value and end value of t are the t_0 and t_1 above, so that $[t_0, t_1]$ is the interval where the parametric function is defined. In other words, they determine which section of the (usually infinite) curve is drawn. In the circle example above we had $[t_0, t_1] = [0, 3\pi/2]$, causing the required circle arc to be drawn.

closed

This field will be **Yes** or **No** in the GUI. The purpose is to tell Gimp if the curve should be closed or not. For example, if drawing a full circle we set **closed** to **No**, the starting and ending points of the resulting curve will be distinct though quite similarly located. The point will be doubled, and the curve is not properly closed. But with **closed=Yes** Gimp will know to close the curve.

(A bizarre fact is that even if the curve is announced to be closed in the GUI and the starting and ending points of the curve are equal when calculated mathematically, Gimp may still make a double point with a very tiny edge between. In some cases, on the other hand, Gimp just closes the curve neatly. Why this is so, is a mystery to the author. Probably it is about computing precision.)

fit in window

If this field is **Yes**, the plot will be scaled and positioned to fill the window in Gimp. In this case the inputs for the origo and scale are ignored.

padding

If you choose to have the plot fitted to fill the window, you may wish that it does not fill quite the whole window. You can then then add some padding (in pixels) and the plot will be a little smaller.

x and y of the origo for the plot and scale for the plot

If you want to specify explicitly where and how large the plot should be, set the entry **fit in window** to **No**. Then you can determine the exact placement and scaling of the plot with these three inputs.

When you decide about the parametric curve to be drawn, such as for example the full circle, you define the function

$$f(t) = (\cos t, \sin t), \quad 0 \leq t \leq 2\pi, \quad (4)$$

in some coordinate system, and of course you choose one that is most convenient for that purpose. The plugin takes care of transforming things to the coordinate system of Gimp's window, and this is what the **origo** and **scale** are for. Their meaning should be obvious. In order to these inputs to have any effect, the entry **fit in window** must be **No**; otherwise they are just ignored.

draw the axes

If **Yes**, the coordinate axes are created as another path. They are always of full width and height of the window.

custom values of t

When the plugin creates the path it has to decide where to set the anchors of the path. Since on the other hand the plugin strives to keep the number of anchors low, placement of those few anchors is crucial to the accuracy of the approximation. The plugin has its internal algorithms for this and generally they work very well, therefore usually you can just ignore the field in the GUI about custom values.

Occasionally, however, you may wish to have anchors set at some specific spots on the curve. Or it may happen that the plugin fails to find cusps or inflection points, say, sufficiently accurately. In such cases you can input a list of custom parameter values. This causes the plugin to put anchors at those spots.

Note that the anchors will be exactly on the correct curve, while everything between the anchors is approximation and necessarily slightly inaccurate. So, if you want to have some points on the curve to be precise, it is achieved by means of the custom parameter values.

Such a list of custom parameter values might look like

$$\text{pi}/4, \text{pi}/2, 3*\text{pi}/4 \quad (5)$$

where the values should obey Python syntax. Expressions like $2*\text{pi}-\text{pi}/10$ are allowed. The values can be input like this, listed with commas as separators. But they can also be given as one true Python list, such as

$$[\text{pi}/3 + k*\text{pi}/5 \text{ for } k \text{ in range}(10)] \quad (6)$$

(this is Python's list comprehension; note the enclosing brackets).

If the list is empty, it is ignored by the plugin. Some too close values may be rejected. Also, values outside of the interval are ignored, hence it does not matter if you input some extra values.

messages

If `messages` is set to `Yes`, the plugin will display some info in the error console. At least there will be an accuracy measure of the result.

2 *Parametric curve (polar)*

Suppose you have to draw a curve given in the polar form, such as, for example, $r = \exp(t)$ (logarithmic spiral). You *can* draw it with the plugin described above since the curve can be written as $(x, y) = (\exp(t) \cos(t), \exp(t) \sin(t))$. But the plugin *Parametric curve (polar)* makes it easier: there you input only one function $r(t)$, in this case $\exp(t)$.

Otherwise this plugin is similar to that explained above.

3 *Parametric curve (read function from file)*

This plugin is similar to the first plugin *Parametric curve (cartesian)* except that the functions $x(t)$ and $y(t)$ are not input in the GUI. Rather, the defining function of the curve is read from a file where it is written as Python code. This method has clear advantages: First, it is much more versatile since it allows real programming of the function. Second, the function is not restricted to the simple form $(x(t), y(t))$. Third, since the function is in a file it is automatically saved.

There is the little complication that the user must create the input file before using the plugin. But that inconvenience is negligible when compared with the advantages.

Example 3.1 Let us take an example of what such file might look like. It must be valid Python code. This example shows all inputs that the file can include (with one omission, see Section 3.1).

```
def function(t):                                # Lemniscate
    s = sin(t)
    x = cos(t) / (1 + s**2)
    y = x * s
    return complex(x,y)

curve_name = 'lemniscate'
interval = [0, 2*pi]
closed = True
custom_params = [pi/2, 3*pi/2]
```

Of these inputs the first one (`function`) is *mandatory*. The other four are *optional*. The names `function`, `curve_name`, `interval`, and `custom_params` must be literally in this form.

These inputs override the values in the GUI; on the other hand, if any of the optional names are missing in the file, the corresponding values in the GUI will be used. There is one exception to this rule: if a list of custom parameters appears both in the file and in the GUI, the plugin merges the two lists into one and uses that.

I explain now each more closely.

function

This is the defining function of the curve, written as Python code. Note that the function does not return `[x,y]` but `complex(x,y)`. Internally the plugin treats the plane as the complex number plane, and the same is used in this input file.

curve_name

This is a Python string which, if it appears in the input file, will be used as the name of the created path.

interval

This is a Python list of two floats. If it appears in the input file it will replace the values for `start t` and `end t` in the GUI.

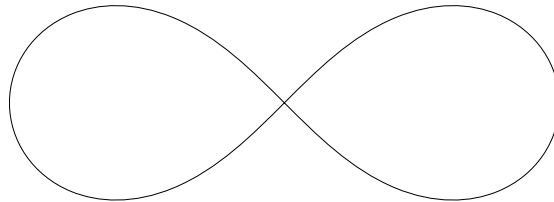
closed

This is a Boolean value (`True` or `False`). If it appears in the input file it will override the corresponding value in the GUI.

custom_params

This is a Python list of floats. There may be such a list both in the file and in the GUI. Neither overrides the other; rather, the plugin merges the lists. In Example 3.1, in the statement `custom_params = [pi/2, 3*pi/2]`, the values `pi/2`, `3*pi/2` give the two inflection points. We can think that the user (the writer of the above example file) wanted the inflection points to be exact and did not quite trust that the plugin can find them accurately enough on its own (which is just realistic) and input the exact values just to make it sure.

Finally, let us see what kind of a curve the plugin draws with the input file of Example 3.1: it is the *lemniscate* (of *Bernoulli*).



Example 3.2 The lemniscate would be easily made with the *Parametric curve (cartesian)* plugin too by inputting in the GUI functions

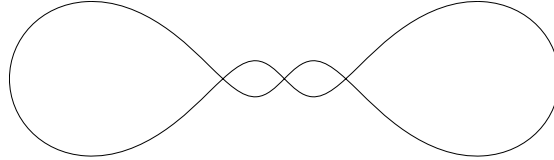
```
x(t) = cos(t)/(1+sin(t)**2),  
y(t) = sin(t)*cos(t)/(1+sin(t)**2).
```

But let us make this a little more complicated. Let us make an extra twist. Let us try the following input file:

```
def function(t):  
    s = sin(t)  
    x = cos(t) / (1 + s**2)  
    y = x * s  
    y *= 1 - 2 / (1 + 20 * x**2) # This line makes the twist  
    return complex(x,y)
```

```
curve_name = 'lemniscate with a twist'
interval = [0, 2*pi]
closed = True
```

This produces the following figure:



Now try to write *this* function as functions $x(t)$, $y(t)$! It is possible but I wouldn't do it. But that is what you would have to do if you, instead of using the plugin *Parametric curve (read function from file)* and an input file, tried to do this with the plugin *Parametric curve (cartesian)* and its GUI. Then imagine that you were doing some experimenting (like I was when finding that twisting formula—but I was wise enough not to do it this way!). You would have to write the functions in the GUI, without any typos. Then you would write the functions in the GUI a second time but with some little changes as an experiment. Then you would do it a third time, and so on.

Sorry about pounding on the same point, but the purpose of the previous example was of course to demonstrate the advantage of using an input file: experimenting requires only simple editing in the file. And another advantage: You have at your disposal the whole machinery of coding in Python, including *if*-clauses, loops, subroutines and so on and so on; and you can even easily create functions which cannot be put in the simple form $(x(t), y(t))$. And yet one: Your formulas don't disappear when you close Gimp since they are safely in the file.

3.1 Extra parameters

In the input file you can also write

```
boost_accuracy = 5
```

where the number is any float from 0 to 10; default is 0. This is currently the only parameter offered to the user to tweak the inner working of the plugin. A higher number causes the plugin to strive for better accuracy with the cost of more numerous control points. Whether there will be any actual changes in the resulting path, depends on the particular case.

It is feasible that in the future there may be other such extra parameters. Namely, there are a number of such tweaking parameters, now hard-coded, that can be changed only by editing the code. But on the other hand, it may well be that those will never be offered as parameters in an input file since they would require some understanding of how the plugin works internally.