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

Version 1.09

April 2020

These pages contain instructions for a Gimp plugin *Parametric curves*, designed to draw a parametric curve, given in Cartesian coordinates, approximately as a Bézier curve (Gimp's path). When installed, it is found in Gimp's menu at

<Image>/Filters/Render/Parametric curves.

The central idea in the plugin is to find an approximate Bézier curve with only a sparse set of control points.

The plugin is applied through its GUI in Gimp. There are two ways to use the plugin: the simple way just by filling the fields in the GUI, and the advanced way by feeding inputs in a user-written file. The former may be quite sufficient for a quick task, but anything more complicated requires usage of an input file. Furthermore, the latter way brings several advantages: First, it offers more control on the plot and the running of the plugin; second, it enables one to draw several parametric curves simultaneously; third, the work becomes saved automatically since it is written before-hand in a file (and this is the only way to save the input curve and the settings!); fourth, to define the curves, available is the whole machinery of Python programming.

The main disadvantage of the advanced method is its complexity and that it requires some effort to master. The user may find it wisest to apply instead another plugin, *Simple parametric curve*, or its counterpart *Simple polar curve*, found at the same location in Gimp's menu if installed.

Those simple plugins have their own documentation elsewhere. We start now studying the plugin *Parametric curves*. The text is in two parts: One covers the simple way (inputs from the GUI), and the other the advanced way (inputs from a file).

Contents

1 Simple usage: inputs from the GUI

1.1 The GUI of the plugin

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

curve name	name of the curve
x(t)	the function $x(t)$
y(t)	the function $y(t)$
starting t	the starting value of the parameter t
ending t	the ending value of the parameter t
custom values of t	user-chosen parameter values to force anchors
closed?	is the curve supposed to be closed?
fit in window?	should the size of the plot be adapted to the window?
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 marked as guides?
read file?	should inputs be read from a file?
» file	if `Yes` above, choose the file
level	a number `1..100`, controlling running of the plugin
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 half 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)`
starting t	`0.`
ending t	3*pi/4
custom values of t	<empty>
closed?	`No`
fit in window?	`No`
x of the origo	`500`
y of the origo	`500`
scale	`100`
draw the axes?	`No`
read file?	`No`
» file	<empty>
level	`1`
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, like `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 constructed approximate Bézier curve.

x(t) and y(t)

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

$$f(t) = (x(t), y(t)), \qquad t_0 \le t \le t_1, \tag{1}$$

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

$$f(t) = (\cos t, \sin t), \qquad 0 \le t \le 3\pi/4, \tag{2}$$

hence $x(t) = \cos t$ and $y(t) = \sin t$, and $t_0 = 0$ and $t_1 = 3\pi/4$.

starting and ending values of t

The starting value and ending 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 where the drawn arc will start and where it will end. In the circle example above we had $[t_0, t_1] = [0, 3\pi/4]$, causing the required circle sector to be drawn.

custom values of t

In the GUI the user may input a custom list of parameter values forcing the plugin to include the corresponding points among the anchors of the Bézier curve. Such a list might look like

```
pi/4, pi/2, 3*pi/4
```
(3)

where the values should obey Python syntax. Expressions like `2*pi-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

```
[pi/3 + k*pi/5 for k in range(10)]
```
(4)

(this is Python's list comprehension; note the enclosing brackets). If the list is empty, it is ignored by the plugin. When the list is not empty, the corresponding points on the original parametric curve will appear among the anchors of the resulting Bézier curve. (Though some too close ones may be rejected, as so are all those not inside the interval.)

The purpose of this feature is that occasionally the result may be unsatisfactory. It was so in the case of the simple plugin, since that plugin makes no effort in finding any special points on the curve. In particular, cusp points are troublesome. But the current plugin works much better since it does search for suitable special points to create an initial subdivision before starting work in earnest. This means that generally the user need not bother to input any custom values.

But it should be noted that cases arise where feedíng the custom points is essential, depending on the curve and on how accurate result the user wants.

closed

This field will be `Yes` or `No` in the GUI (meaning `True` or `False`). 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. In a way the point will be double, 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.)

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 next three entries in the GUI (the coordinates of the origo and the scale) are ignored.

x and y of the origo and scale

When the user decides about the parametric curve to be drawn, such as the full circle

$$f(t) = (\cos t, \sin t), \qquad 0 \le t \le 2\pi, \tag{5}$$

it will be in some coordinate system, and the user of course chooses one that is most comfortable. The plugin has to transform it to the coordinates 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`.

draw the axes

If `Yes`, the coordinate axes are created, currently only as two guides. If either axis is outside of the window, it will not be created.

read file? and file

The inputs or some of them can be inputted in a user-written file. If `read file` is set to `Yes`, the user has also to use the next field `file` to choose the file. We look at this more in detail in Section 2.

level

By the input `level` the user can to some extent determine how much the plugin will work and how accurate the result will be. The `level` is an integer `1..100`; default is `1`, and higher level may cause a larger number of control points on the resulting Bézier curve (and higher recursion depth in the algorithm).

messages

If `messages` is set to `Yes`, the plugin will display some info in the error console. There will always appear an accuracy measure of the result; the meaning of that measure has no simple description but hopefully it will serve some purpose. In addition, there may appear messages about decisions made by the algorithm, or difficulties encountered; the messages might help the user to catch some inadvertences in the inputs, in particular when some of the inputs are taken from a file.

Note however that the plugin often displays rather scary error messages. But if it then finally informs having finished successfully with good accuracy, the user can safely ignore the error messages. Namely, the plugin works recursively by splitting arcs in two where necessary. So, if it finds some arc too difficult to

handle it may show an error message and then it simply goes to recursion, splits the arc in two, and usually ends up with a quite good result after all.

Remark 1.2 Of the entries in the GUI, `level` and `messages` work differently from the rest when inputs are read from a file (next section): when inputs are taken from a file, the plugin obeys the settings of `level` and `messages` currently appearing in the GUI (and there is no way to change them in the file!). As for the other entries, if some of them are undefined in the input file, the plugin uses default values for them, ignoring what happens to be in the GUI.

An explanation to why the author has chosen such behaviour, is that `level` and `messages` control the running of the plugin and are not concerned with the parametric curves or the plot.

And note that the simple plugins mentioned at the beginning behave differently in this respect.

2 Advanced method: Inputs from a file

2.1 Names read from the file

The GUI is simple to try but too simple for serious work. Consider, for instance, the difficulty of inserting the expressions for `x(t)` and `y(t)` and getting them right (see Example **??**). It is much easier to edit and save a file for this purpose. This choice also enables one to use much more complicated parametric functions $f(t)$: the whole machinery of writing Python functions is available.

Example 2.1 Take a decent text editor (like Notepad++) and create a file with the following contents. The text must be proper Python, with proper indentations and so on. Save the file.

```
def astroid(t):
    return [cos(t)**3, sin(t)**3]

pcurve = ParametricCurve(
    function    = astroid,
    pcurve_name = "astroid",
    start       = 0.,
    end         = 2*pi,
    closed      = True,
    custom      = CustomPoints(params = [pi/2, pi, 3*pi/2])
    )

pcurve_collage = ParametricCurveCollage(
    pcurve_list  = [pcurve],
    collage_name = 'Astroid'
    )
```

```
c2bs = Curve2BezierSettings()

c2bo = Curve2BezierOptions()

coos = CoordinateSettings(
    fit_window = True,
    )

drawo = DrawingOptions(
    axes = True,
    )
```

If in the GUI the entry `read file?` is set to `Yes` and the entry below it is used to choose the above file, and then OK is hit, the plugin draws precisely the same astroid as before. To tell the truth, it may be ever so slightly different because of the list of custom points—we shall talk about that more later.

Admittedly, the contents of the file must look strange. The meanings of the various terms in the file are discussed next. First, the plugin expects to find in the file definitions for certain names. The names the plugin reads from the file are the six in the following list.

```
pcurve_collage
c2bs
c2bo
coos
drawo
mark_polylines
```

Of these only the first is mandatory. The others can be omitted, which causes the plugin to use some default values; for example, the line

```
c2bs = Curve2BezierSettings()
```

could have been left out entirely. The names mentioned in the list, if present, must appear in the file literally as they are above but their order can be arbitrary. On the other hand, to implement the definitions one probably has to write some extra Python code in the file. In the example above, to define `pcurve_collage` we had to define `pcurve` and `astroid`.

Remark 2.2 If inputs are read from a file, the current entries in the GUI are ignored, with the exception of `level` and `messages` (cf. Remark 1.2). The settings of `level` and `messages` in the GUI are in force whether the inputs are taken from the GUI or a file (and there is no way to change them in the file!). As for other inputs, on the other hand, if a particular entry is not defined in the file, a default value is used and the value currently in the GUI is ignored—this behaviour differs from the simple plugins.

However, the effect of `level` is obtained through the input file too, by setting the value of `c2bs.max_error` (see Section 2.3 below). Namely how the entry

`level` works, is that after reading the value of `max_error` from the file (or setting it to the default value if it is not specified in the file), the plugin replaces the value with

```
max_error / level
```

before starting working, where the `level` is taken from the GUI. So for example, the effect of `max_error=0.01` in the file and `level=10` in the GUI, is equivalent to the effect of `max_error=0.001` in the file and `level=1` in the GUI.

Now we look at the input entries one by one.

2.2 pcurve_collage

The plugin is not restricted to drawing only a single parametric curve. It is designed so that one can define and draw several parametric curves in the same picture. For this reason, in Example 2.1, the object `pcurve_collage` is the container which could hold several parametric curves, though in this example there was only one, called `pcurve`.

In the Python code for the plugin there is the definition of the class called `ParametricCurveCollage`. The part which interests us now reads as follows.

```
class ParametricCurveCollage:
    """A collection of objects of class ParametricCurve together
    with a name.

    The purpose of the class 'ParametricCurveCollage' is to enable
    drawing simultaneously several parametric curves in the same
    picture using the same settings and options.

    Initialization arguments:
    - pcurve_list:  [ParametricCurve]
    - collage_name: string or None
    """
    def __init__(self,
            pcurve_list  = None,
            collage_name = None
            ):
        ..............
```

You see now that in Example 2.1 the lines

```
pcurve_collage = ParametricCurveCollage(
    pcurve_list  = [pcurve],
    collage_name = 'Astroid'
    )
```

create an object of class `ParametricCurveCollage` and initialize it by setting (1) argument `pcurve_list` to the one-item list `[pcurve]`, and (2) argument

`collage_name` to `'Astroid'`. The latter is clear: it is just a string, the name for the collage. But the first entry needs explaining. In the above extract from the code, inside the comment we read:

```
    Initialization arguments:
    - pcurve_list: [ParametricCurve]
    ............
```

This informs that the initialization argument `pcurve_list` must be a list of objects of class `ParametricCurve`. We explain that class next.

2.2.1 ParametricCurve

Class `ParametricCurve` is data structure for one parametric curve. Let us again find the relevant part in the code:

```
class ParametricCurve:
    """A parametric curve in the xy-plane.

    Initalization arguments:
    - function:    callable (function R-> R2 in Python)
    - pcurve_name: string
    - start:       float
    - end:         float
    - closed:      boolean
    - custom:      CustomPoints or None
    """
    def __init__(self,
            function    = None,
            pcurve_name = "parametric curve",
            start       = 0.,
            end         = 1.,
            closed      = False,
            custom      = None,
            ):
         ..............
```

In Example 2.1 the lines

```
pcurve = ParametricCurve(
    function    = astroid,
    pcurve_name = "astroid",
    start       = 0.,
    end         = 2*pi,
    closed      = True,
    custom      = CustomPoints(params = [pi/2, pi, 3*pi/2])
    )
```

initialize such an object and give it the name `pcurve` (this name is arbitrary and is only used when initializing the `pcurve_collage`). This initialization creates an object which implements the parametric curve *astroid*

$$f(t) = (\cos^3 t, \sin^3 t) \qquad (0 \le t \le 2\pi). \tag{6}$$

The initialization argument `function` must refer to a Python implementation of a function $\mathbb{R} \to \mathbb{R}^2$. In Example 2.1 there is first the definition

```
def astroid(t):
    return [cos(t)**3, sin(t)**3]
```

which implements the function for the astroid. Then, in the initialization of `pcurve`,

```
pcurve = ParametricCurve(
    function    = astroid,
    pcurve_name = "astroid",
    start       = 0.,
    end         = 2*pi,
    closed      = True,
    custom      = CustomPoints(params = [pi/2, pi, 3*pi/2])
    )
```

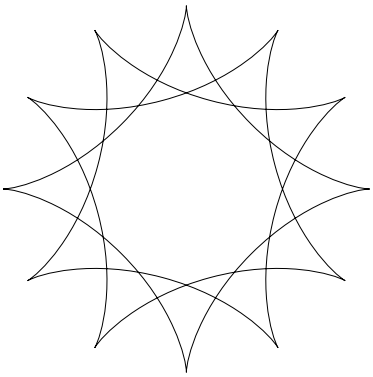
this function `astroid` is set as the value for the argument `function`. The next four entries are clear: `pcurve_name` sets the name for this parametric curve (not to be confused with the name of the whole collage); `start` and `end` set the starting and ending values for the parameter t; and `closed` tells Gimp that the curve should be closed. The final entry, `custom` is more complicated, and we defer the discussion of it to Section 2.2.2. We take first another example.

Example 2.3 Let us see what the input file looks like if we are drawing more than one curve at the same time. We take our astroid, and in the same picture we draw also the astroid rotated by 30 and by 60 degrees. So, we build a collage consisting of three parametric curves. The file might look as follows, where we have dropped everything not strictly necessary (resorting to default values):

```
def astroid(t):
    return [cos(t)**3, sin(t)**3]

def astroid30(t):
    angle = pi/6
    c,s = cos(angle), sin(angle)
    x,y = cos(t)**3, sin(t)**3
    x,y = x*c - y*s, x*s + y*c    # Rotation
    return [x,y]

def astroid60(t):
```

```
    angle = pi/3
    c,s = cos(angle), sin(angle)
    x,y = cos(t)**3, sin(t)**3
    x,y = x*c - y*s, x*s + y*c    # Rotation
    return [x,y]

pcurve = ParametricCurve(
    function    = astroid,
    pcurve_name = "astroid",
    start       = 0.,
    end         = 2*pi,
    closed      = True,
   )

pcurve30 = ParametricCurve(
    function    = astroi30,
    pcurve_name = "astroid 30",
    start       = 0.,
    end         = 2*pi,
    closed      = True,
    )

pcurve60 = ParametricCurve(
    function    = astroid60,
    pcurve_name = "astroid 60",
    start       = 0.,
    end         = 2*pi,
    closed      = True,
    )

pcurve_collage = ParametricCurveCollage(
    pcurve_list  = [pcurve, pcurve30, pcurve60],
    collage_name = 'Three astroids'
    )
```

Indeed, this file draws the following figure. The three astroids will be three separate vectors objects in Gimp. We shall later see how they can be automatically merged into one object.



The file above is meant to be easy to read. A seasoned Python coder would write it much more concisely.

2.2.2 CustomPoints

In Section 1.1 we saw, in the connection with the GUI, that the plugin can be forced to include some chosen points as anchors of the resulting Bézier curve. There those custom points where inputted as anchors of an auxiliary path. The advanced method (inputs from a file) too allows the same thing, but now the way of inputting the custom points is quite different. They are inserted as numerical values in the initialization argument `custom` of the object of class `ParametricCurve`. We saw above the following definition.

```
pcurve = ParametricCurve(
    function    = astroid,
    pcurve_name = "astroid",
    start       = 0.,
    end         = 2*pi,
    closed      = True,
    custom      = CustomPoints(params = [pi/2, pi, 3*pi/2])
    )
```

Here `[pi/2, pi, 3*pi/2]` is the list of custom points. Its effect is that the three cusps of the astroid (excluding the starting and ending point) will appear as anchors, ensuring that the Bézier curve will certainly get the cusps exactly right.

There are, in fact, three forms to input the custom points. Above we see the name `CustomPoints`, appearing in

```
CustomPoints(params = [pi/2, pi, 3*pi/2]),
```

which is, as one can guess, initialization of an object of class `CustomPoints`. To see what that class is like, we refer again to the code of the plugin:

```
class CustomPoints:
    """User-defined points on the parametric curve which the user
```

```
    wants to be forced as anchor points of the approximate Bezier
    curve, in addition to the anchor points which the program
    itself makes.

    The custom points can be given in any of the following three
    forms, or even in two or three forms simultaneously. Custom
    points is the means for the user to provide, for example,
    exact cusp points or inflection points - rather than just
    relying on what the program itself finds, which is always a
    little inexact.

    Here "raw coordinates" means the coordinate system in which
    the input parametric curve is defined, and "screen coordinates"
    means the coordinate system of the screen of the drawing
    application.

    Initialization arguments:
        params:        a list [...,t,...] of custom parameter
                       values t such that each f(t) will be a
                       custom point
                       (default=[]);
        points:        a list [...,[x,y],...] of custom points
                       [x,y] in raw coordinates
                       (default=[]);
        screen_points: a similar list [...,[x,y],...] of custom
                       points [x,y] but in screen coordinates
                       (default=[]).
    """
    def __init__(self,
            params        = [],
            points        = [],
            screen_points = []
            ):
         ............
```

That should explain it. We take one example.

Example 2.4 Suppose that we are one more time drawing the astroid and we wish to input the same three custom points as above. In addition, for some strange reason, we wish to get anchors somewhere in the vicinity of points $(\pm 1/3, 1/3)$ in the "raw coordinates" (the coordinates where the astroid function was defined) and somewhere near the point $(600, 600)$ in Gimp's window. Then the whole definition of `pcurve` would be the following:

```
pcurve = ParametricCurve(
    function    = astroid,
    pcurve_name = "astroid",
```

```
start       = 0.,
end         = 2*pi,
closed      = True,
custom      = CustomPoints(params = [pi/2, pi, 3*pi/2],
                           points = [[1/3,1/3], [-1/3,1/3]],
                           screen_points = [[600,600]]
                          )
```

The points listed in `points` or `screen_points` need not to lie on the astroid; rather, for any such point the plugin tries to find the closest point on the true mathematical astroid to be used as an anchor. If any of the arguments `params`, `points`, or `screen_points` is omitted, the plugin uses an empty list as default.

2.3 c2bs

We return to the list of the names that the plugin reads from the file:

```
pcurve_collage
c2bs
c2bo
coos
drawo
mark_polylines
```

The first one is already explained. We take now the second one, `c2bs`. It enables the user to adjust inner working of the plugin. To say it bluntly, the user probably does wisest to leave it alone. But we explain it here anyhow. If you wish to use it, you type in the input file something in the form:

```
c2bs = Curve2BezierSettings(...)
```

with a list of some arguments. This is initialization of an object of class `Curve2BezierSettings`. From the code of the plugin we learn:

```
class Curve2BezierSettings:
    """Numerical parameters to manage how a parametric curve is
    resolved approximately as a Bezier curve.

    Initialization arguments:
        angle:         the maximum allowed turn in each
                       subdivision arc (forced to between
                       30 and 180 degrees with tolerance
                       of 1e-8)
                       (default=90 degrees);
        refinement:    higher refinement forces refinement of the
                       subdivision along the whole curve, increasing
                       the number of anchor points otherwise used
                       (default=0);
```

```
    max_error:     maximum allowed relative error
                   ('max_error' is forced to between 0.0005
                   and .5 but these are arbitrary limits)
                   (default=0.01);
    max_recursion: if the error (in initial computation), for
                   some arc of the subdivision, appears to be
                   larger than the allowed maximum ('max_error'),
                   then the program goes to recursion by
                   splitting the troublesome arc in two, so
                   increasing the number of anchor points; the
                   value 'max_recursion' is the maximum allowed
                   recursion depth; setting it to 0 suppresses
                   recursio.
                   (default=3);
"""
def __init__(self,
        angle          = pi/2,
        refinement     = 0,
        max_error      = 0.01,
        max_recursion  = 3
        ):
    ............
```

Of these we describe only the `angle`. The plugin follows the idea that it splits the input curve recursively into smaller pieces when trying to find an acceptable approximation. But prior to going to do any recursion, the plugin tries to make its work easier (and presumably getting better final result) by dividing the curve first at some special points and then only after that doing the main work on each smaller arc separately. Construction of this initial subdivision on the curve is the first big phase in the algorithm. The user can, actually, determine which special points are searched for; about this in the next section. But the most important are (1) cusps, (2) points of locally maximum or minimum curvature, and (3) the points controlled by `angle`: the points where the curve has turned a certain amount in one direction. To explain the last one, consider as an example a spiral—for example, a long arc of the logarithmic spiral. To approximate it by a Bézier curve, some subdivision has to be made, but there are no cusps, no inflection points, or any other special points. The plugin makes the initial subdivision by dividing the curve into arcs which turn at most by the amount of the value of `angle`. The default value for `angle` is 90°, which works quite well. The user may set it to some smaller value, say 45°, thus getting more precise approximation with the cost of a larger number of control points. Or the user can experiment with larger values for `angle` at ones own risk.

2.4 c2bo

With the next name, `c2bo`, the user may decide which special points on the curve are searched for in the construction of the initial subdivision (see the previous section). The choice is done by writing in the input file something like:

```
c2bo = Curve2BezierOptions(...)
```

with a list of arguments. This initializes an object of class `Curve2BezierOptions`. In the code of the plugin we read:

```
class Curve2BezierOptions:
    """Boolean parameters to manage which special points are
    searched for and are included among subdivision points
    (in addition to the user-defined custom points if any).

    The default values should be safe. Other choices may cause the
    algorithm to fail but often may give interesting results.

    Initialization arguments (all Boolean):
        cusp_points:           include cusp points
                               (default=True);
        high_curvature_points: include points of locally highest
                               curvature
                               (default=True);
        low_curvature_points:  include points of locally lowest
                               curvature
                               (default=True);
        straight_end_points:   include both start and end points
                               of straight line segments
                               (default=True);
        inflection_points:     include inflection points
                               (the program forces this to be
                               - False if 'low_curvature_points'=True,
                               - True if 'low_curvature_points'=False
                                        and 'angle_turn_points'=True)
                               (default=False);
        angle_turn_points:     include subdivision points to ensure
                               that each arc turns at most by the
                               amount of 'angle' (set in the attribute
                               'angle' in an input argument of class
                               Curve2BezierSettings)
                               (default=True).
    """
    def __init__(self,
            cusp_points           = True,
            high_curvature_points = True,
```

```
            low_curvature_points  = True,
            straight_end_points   = True,
            inflection_points     = False,
            angle_turn_points     = True
            ):
        ..............
```

Perhaps there is no need to explain this any further. The user may experiment.

2.5 coos

Inserting in the input file a line

```
coos = CoordinateSettings(...)
```

with some arguments, the user initializes an object of class `CoordinateSettings` which controls working with coordinates. Recall that we have two coordinate systems: the one where the input parametric curve is defined (cf. the astroid examples above), and the coordinates of Gimp's window where the plot is drawn. The code of the plugin tells us:

```
class CoordinateSettings:
    """Manage how the final plot is positioned on the screen.

    Initialization arguments:
        fit_window: Boolean: determines if the plot is to be fit
                    in the window minus the paddings
                    (when 'fit_window' is True, the settings for
                    'origo_x', 'origo_y', and 'scale' are ignored)
                    (default=True);
        padding:    float: the padding to be left empty on the
                    screen as a fraction of the window size when
                    'fit_window' is True
                    (it is forced that 0 <= padding <= 0.45)
                    (default=0.);
        origo:      a pair of floats: where to set the origo on
                    the screen in screen coordinates
                    (ignored if 'fit_window' is True)
                    (default=[500,500]);
        scale:      scaling factor from raw coordinates to screen
                    coordinates
                    (ignored if 'fit_window' is True)
                    (default=100.).
    """
    def __init__(self,
            fit_window = True,
            padding    = 0.,
```

```
            origo      = [500.,500.],
            scale      = 100.
            ):
        .............
```

This all should be self-evident.

2.6 drawo

A line

```
drawo = DrawingOptions(...)
```

with some arguments in the input file causes initializion of an object of class `DrawingOptions` which controls what objects will be drawn on the screen. From the code:

```
class DrawingOptions:
    """Boolean parameters to manage which objects are to be
    drawn.

    Here "drawing" means creating a separate Gimp vectors object,
    inserting it in Gimp, and making visible; or, for the
    coordinate axes, it means creating guides.

    The 'pcurves' mentioned below are the elements of
    'pcurve_input.pcurve_collage.pcurve_list' where
    'pcurve_input' is an object of class ParametricCurveInput.
    Thus, 'pcurve_list' is the list of parametric curves to be
    drawn simultaneously in the same picture using the same
    settings and options.

    Initialization arguments:
        bezier_curves:  Boolean: if True, then the resulting Bezier
                        curves (approximations of the elements
                        'pcurves') are not only computed but also
                        drawn, each Bezier curve individually
                        (default=True);
        bezier_collage: Boolean: if True, then from the Bezier curves
                        mentioned above, a combined object is created
                        and drawn (in Gimp this means one vectors
                        object where each one of the 'pcurves' appears
                        as one stroke)
                        (default=False);
        axes:           Boolean: if True, then the coordinate axes are
                        marked as guides on the screen
                        (default=False);
```

```
    x_axis:         Boolean: if True, then the x axis is
                    marked as a guide on the screen
                    (default=False);
    y_axis:         Boolean: if True, then the y axis is
                    marked as a guide on the screen
                    (default=False);
    exact_pcurves:  Boolean: if True, a sparse set of points on
                    each exact input parametric curve is drawn
                    (default=False);
    handles:        Boolean: if True, the anchors and handles of
                    the computed Bezier curve are drawn
                    (default=False);
"""

def __init__(self,
        bezier_curves  = True,
        bezier_collage = False,
        axes           = False,
        x_axis         = False,
        y_axis         = False,
        exact_pcurves  = False,
        handles        = False
        ):
```

If `bezier_curves=False` and `bezier_collage=True`, the parametric curves in the collage are not drawn individually; instead they are merged into one vectors object which is drawn.

The reason behind the options `exact_pcurves` and `handles` is the following. The option `exact_pcurves` gives the user chance to check visually how well the computed Bézier curves approximate the input curves. If it is `True`, the plugin makes another vectors object (for each pcurve in the collage) which shows a sparse set of points along the exact mathematical input parametric curve. Note that the computation of the sparse point set is completely independent of the algorithms with which the program computes the approximate Bézier curve, hence it is indeed a good way to check the result.

The option `handles` enables the user to see visually what kind of control points the program has chosen. That is something the user may be curious about, and it is likely to have some relevance when trying to find the best settings for some specific task. The handles are constructed5 as a separate vectors object.

2.7 mark_polylines

The last entry, `mark_polylines`, is distinct from the main working of the plugin. It enables the user to embellish the picture with line segments (polylines). To be more precise, the polylines include single points, line segments, open polylines,

and closed polylines (polygons).

The format is

```
mark_polylines = a list of items (polylines)
```

where each `item` (a `polyline`) is of one of the following three forms:

```
[[x0,y0],[x1,y1],...,[xn,yn]]
[[x0,y0],[x1,y1],...,[xn,yn], False]
[[x0,y0],[x1,y1],...,[xn,yn], True]
```

with each `xi` and `yi` float. So, each `polyline` is just a Python list of points in the plane (the vertices of the polyline) with one optional item which is either `False` or `True`. That last item tells if the polyline is supposed to be closed. If the last item is omitted, it is taken to be `False`; so of the three forms above, the first and the second are equivalent—an open polyline.

Let us take an example. Suppose we are given six points `p0`, `p1`, `p2`, `p3`, `p4`, `p5` in the plane (that is, each `pi` is of the form `[x,y]` where `x` and `y` are float). Then the following commands in the input file,

```
poly0 = [p0]
poly1 = [p1, p2]
poly2 = [p3, p4, p5, True]
mark_polylines = [poly0, poly1, poly2]
```

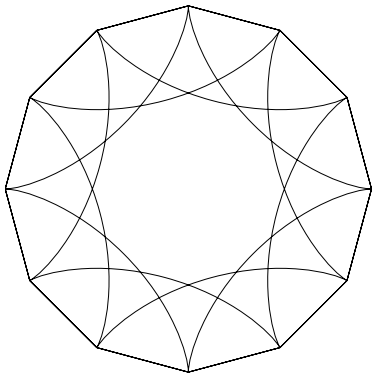
will produce a vectors object consisting of three strokes: (1) one point `p0`; (2) a line segment from `p1` to `p2`; (3) a closed triangle with vertices `p3`, `p4`, `p5`.

It should be noted that if the user sets `fit_window=True` in the entry `coos` (`CoordinateSettings`), the polylines are not taken into account when fitting the drawing in the window, so they may fall outside of it.

Example 2.5 If in the file of Example 2.3 we add lines

```
mark_polylines = [
                  [
                   astroid(0),      astroid30(0),      astroid60(0),
                   astroid(pi/2),   astroid30(pi/2),   astroid60(pi/2),
                   astroid(pi),     astroid30(pi),     astroid60(pi),
                   astroid(3*pi/2), astroid30(3*pi/2), astroid60(3*pi/2),
                   True
                  ]
                 ]
```

we get the following picture.



One closed polyline was drawn. If we had had several polylines to add, it would have been done in the same `mark_polylines` clause.

2.8 Summary

We draw a scheme about all the inputs that the plugin reads from a file. Recall that `pcurve_collage` is the only mandatory.

Names

```
pcurve_collage = ParametricCurveCollage(pcurve_list = (*),
                                        collage_name
                                       )
(*) [ParametriCurve(function,
                    pcurve_name,
                    start,
                    end,
                    closed,
                    custom = CustomPoints(params,
                                          points,
                                          screen_points
                                         )
                   )
    ]

c2bs = Curve2BezierSettings(angle,
                            refinement,
                            max_error,
                            max_recursion,
                           )

c2bo = Curve2BezierOptions(cusp_points,
                           high_curvature_points,
                           low_curvature_points,
```

```
                            straight_end_points,
                            inflection_points,
                            angle_turn_points
                           )

coos = CoordinateSettings(fit_window,
                          padding,
                          origo,
                          scale
                         )

drawo = DrawingOptions(bezier_curves,
                       bezier_collage,
                       axes,
                       x_axis,
                       y_axis,
                       exact_pcurves,
                       handles
                      )

mark_polylines = [(**)]

(**) = one of:
        [[x0,y0],[x1,y1],...,[xn,yn]]
        [[x0,y0],[x1,y1],...,[xn,yn], False]
        [[x0,y0],[x1,y1],...,[xn,yn], True]
```

Types

```
ParametricCurveCollage(
    pcurve_list:  [ParametriCurve]
    collage_name: string
    )

ParametriCurve(
    function:    callable (float -> [float,float])
    pcurve_name: string
    start:       float
    end:         float
    closed:      boolean
    custom:      CustomPoints

CustomPoints(
    params:        [float]
    points:        [[float,float]]
    screen_points: [[float,float]]
```

```
    )

Curve2BezierSettings(
    angle:         float
    refinement:    integer >= 0
    max_error:     float
    max_recursion: integer >= 0
    )

Curve2BezierOptions(
    cusp_points:           boolean
    high_curvature_points: boolean
    low_curvature_points:  boolean
    straight_end_points:   boolean
    inflection_points:     boolean
    angle_turn_points:     boolean
    )

CoordinateSettings(
    fit_window: boolean
    padding:    float
    origo:      [float,float]
    scale:      float
    )

DrawingOptions(
    bezier_curves:  boolean
    bezier_collage: boolean
    axes:           boolean
    x_axis:         boolean
    y_axis:         boolean
    exact_pcurves:  boolean
    handles:        boolean
    )

mark_polylines = [one of:
                  [[x0,y0],[x1,y1],...,[xn,yn]]
                  [[x0,y0],[x1,y1],...,[xn,yn], False]
                  [[x0,y0],[x1,y1],...,[xn,yn], True]
                 ]
```