
qsketchmetric

Release 1.7.1

Franciszek Łajszczak

Mar 19, 2024

TABLE OF CONTENTS:

1	Getting started	3
1.1	Why QSketchMetric?	3
1.2	What is DXF file?	3
1.3	DXF file versions supported	3
1.4	How the documentation is organized?	3
2	Quick install guide	5
2.1	Supported Python Versions	5
2.2	Basic Installation	5
3	Tutorials	7
3.1	Tutorial - Rendering your first parametric DXF file	7
3.2	Tutorial - Manual parametrizing your first DXF file	9
3.3	Tutorial - Semi-automatic parametrization of a DXF file	13
3.4	Tutorial - Validation your first DXF file	15
3.5	Tutorial - Rendering a point	18
3.6	Tutorial - Rendering a custom line style	20
4	How-to guides	23
4.1	Manual parametrization	23
4.2	Semi-automatic parametrization	26
4.3	Validating a parametrized DXF file	28
4.4	Rendering a DXF file	30
5	Explanation	33
5.1	Debug report	33
5.2	MTEXT	35
5.3	VIRTUAL_LAYER	36
6	Reference	39
6.1	Renderer	39
6.2	Semi-automatic parametrization	40
7	Getting help	41
8	How to contribute	43
8.1	Issues / Feature requests	43
8.2	Make changes	43
8.3	Tests	44
8.4	Commit your update	44
8.5	Pull request	44

8.6 Your PR is merged!	44
Python Module Index	45
Index	47

gitp.. qsketchmetric documentation master file, created by

sphinx-quickstart on Sun Aug 20 14:22:12 2023. You can adapt this file completely to your liking, but it should at least contain the root *toctree* directive.



Python 2D parametric DXF rendering engine.

Tip:

The name QSketchMetric comes from the combination of three words:

- QCAD (reference to the CAD software)
 - Sketch (reference to the process of DXF file parametrization)
 - Parametric (reference to the parametric nature of the rendering engine)
-

GETTING STARTED

QSketchMetric is a tool designed to interpret parametric DXF files and render them. Parametrizing of a DXF file happens through **QCAD Professional's** CAD software, which is a commercial software. The parametrized DXF file is then interpreted by QSketchMetric and rendered according to the mathematical expressions.

1.1 Why QSketchMetric?

QSketchMetric was born out of a genuine need. There was an evident demand for a Python tool capable of generating **DXF 2D** drawings based on parametric descriptions, rendering them according to provided variables. This tool enables a seamless workflow during the production process, particularly beneficial for devices like **plotters**.

Typical Use Cases

Consider a box's cutting layout: with `qsketchmetric`, it can be dynamically rendered to adapt based on its desired size.

1.2 What is DXF file?

DXF (Drawing Interchange Format, or Drawing Exchange Format) is a CAD data file format developed by Autodesk for enabling data interoperability between AutoCAD and other programs.

1.3 DXF file versions supported

QSketchMetric supports DXF files newer or equal to **R13** (Introduced in 1994).

1.4 How the documentation is organized?

Documentation follows [diataxis](#) structure. It is divided into the following parts:

- *Tutorials* take you by the hand through a series of steps to start using QSketchMetric. Start here if you're new to QSketchMetric.
- *How-to guides* are recipes. They guide you through the steps involved in addressing key problems and use-cases. They are more advanced than tutorials and assume some knowledge of how QSketchMetric works.
- *Explanation* guides discuss key topics and concepts of QSketchMetric.
- *Reference guides* contain technical reference for all aspects of QSketchMetric's machinery. They describe how it works but assume that you have a basic understanding of key concepts.

QUICK INSTALL GUIDE

2.1 Supported Python Versions

QSketchMetric requires at least Python 3.9

2.2 Basic Installation

The most common case is the installation by pip:

```
pip install qsketchmetric
```


TUTORIALS

Tutorials take you by the hand through a series of steps to start using QSketchMetric. Start here if you're new to QSketchMetric.

3.1 Tutorial - Rendering your first parametric DXF file

Let's learn by example.

In this tutorial you will render a parametric DXF file using the `qsketchmetric.renderer.Renderer` module.

We'll assume you have *QSketchMetric installed* already.

First download the `tutorial.dxf` file from the [QSketchMetric repository](#). It is an example of a parametric DXF file that we will use in a tutorial.

To do so, open it and click Ctrl+S to save it to your computer. As a convention, we'll assume you saved it in a file called `tutorial.dxf`.

Now, create a new file called `render.py` and place it in the same directory as `tutorial.dxf`.

Open `render.py` in your favorite text editor and import the `qsketchmetric.renderer.Renderer` module as well as the `ezdxf.new()` function:

```
from qsketchmetric.renderer import Renderer
from ezdxf import new
from ezdxf import units
```

The first one will be used to render the parametric DXF file, the second one to create output `ezdxf.document.Drawing` and the third one to set the units of the output drawing.

Create an output `ezdxf.document.Drawing` object using `ezdxf.filemanagement.new()` module. Remember to set the units of the output drawing to millimeters as the parametric DXF file defaults to meters.:

```
output_dxf = new()
output_dxf.units = units.MM
```

Before we will render `tutorial.dxf` let's check it out in the [QCAD Professional](#) CAD software to see briefly what it looks like (*File -> Open*). This is what you should see:

Note: To see how to parametrize a drawing, see [Manual parametrization](#).

We can see it is a parametric drawing of a chalice. To render, it needs variable `h` that stands for height of the chalice. Let's set it to 50:

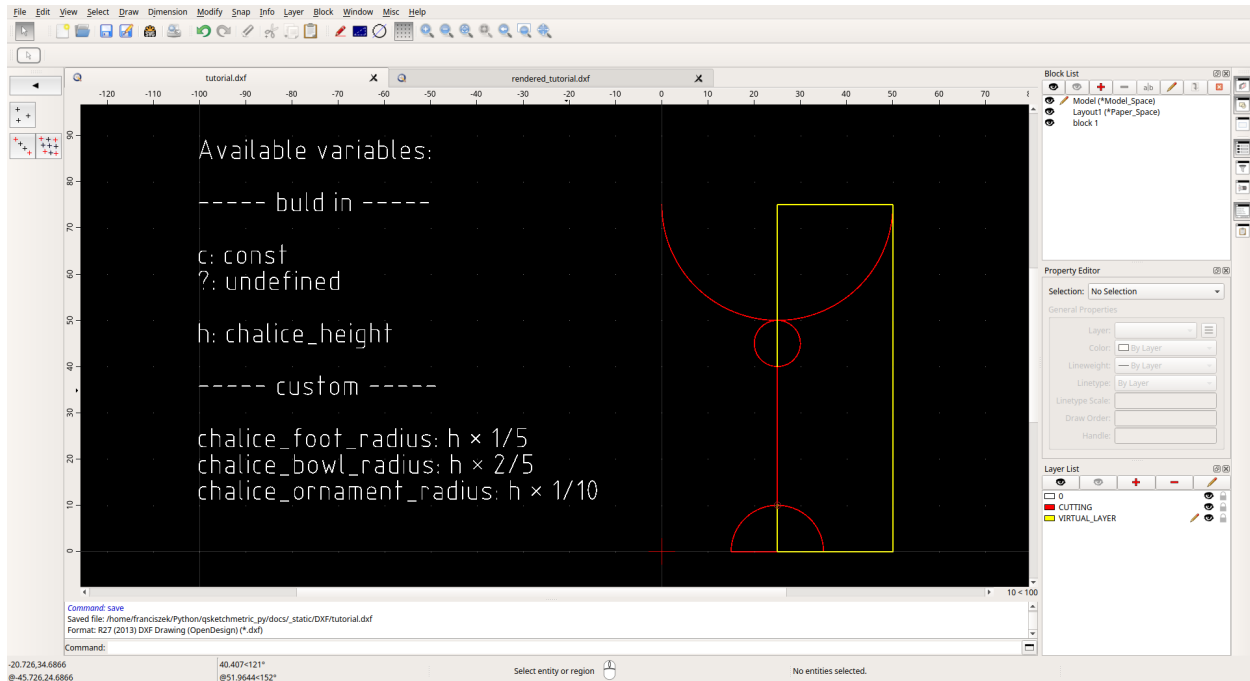


Fig. 1: tutorial.dxf opened in QCAD Professional

```
input_variables = {'h': 50}
```

Now we are ready to roll. Let's render the parametric DXF file:

```
renderer = Renderer('tutorial.dxf', output_dxf, input_variables)
renderer.render()
```

Finally, save the output drawing:

```
output_dxf.saveas('rendered_tutorial.dxf')
```

The whole code should look like this:

```
from qsketchmetric.renderer import Renderer
from ezdxfr import new

output_dxf = new()
input_variables = {'h': 50}
renderer = Renderer('tutorial.dxf', output_dxf, input_variables)
renderer.render()
output_dxf.saveas('rendered_tutorial.dxf')
```

Now run the code:

```
python render.py
```

Finally open rendered_tutorial.dxf in QCAD Professional. This is what you should see:

As you can see, the parametric DXF file was rendered successfully and the chalice height is 50.

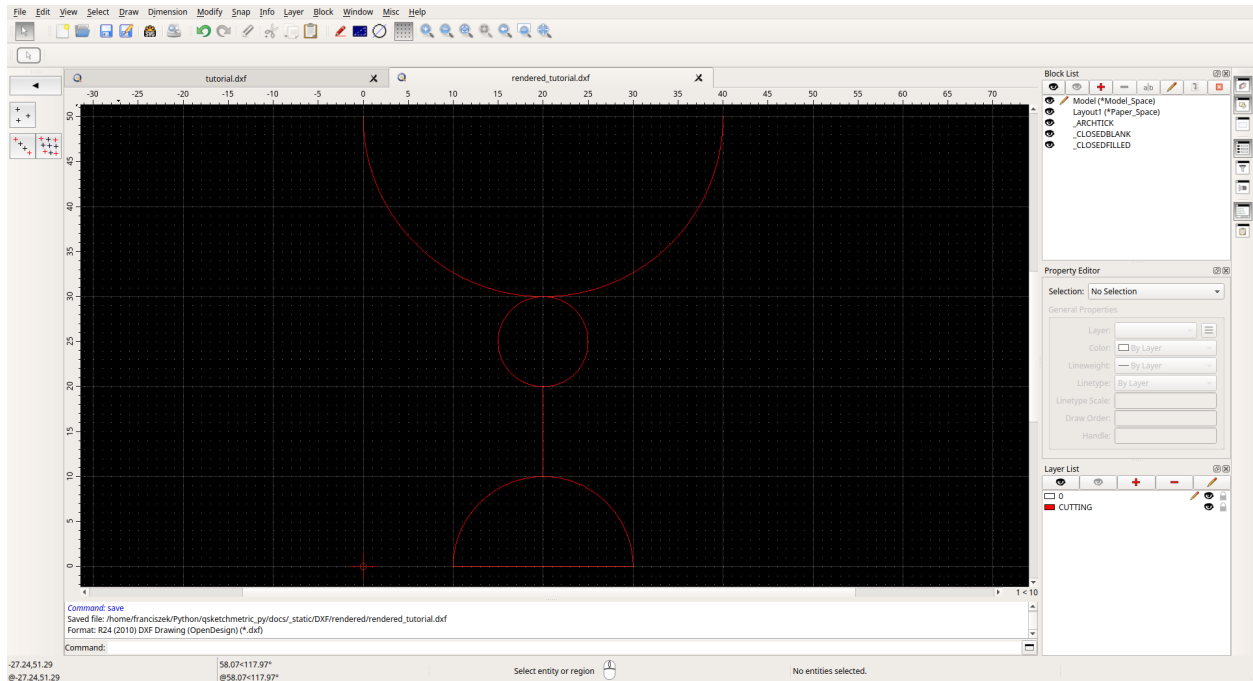


Fig. 2: rendered_tutorial.dxf opened in QCAD Professional

Congratulation you renderer your first parametric DXF file!

3.2 Tutorial - Manual parametrizing your first DXF file

Let's learn by example.

In this tutorial, we will learn how to parametrize a DXF file.

We'll assume you have *QSketchMetric installed* already as well as *QCAD Professional*

First download the [tutorial_param.dxf](#) file from the *QSketchMetric repository*. It is an example of a DXF file that we will parametrize in this tutorial.

To do so, open it and click *Ctrl+S* to save it to your computer. As a convention, we'll assume you saved it in a file called *tutorial_param.dxf*.

First, let's open the file in QCAD Professional. (*File -> Open*)

As you can see, it is a simple drawing of a chalice. With every entity placed on the CUTTING layer.

We would like to parametrize it depending on the given size of the chalice. We will call chalice height variable: *h*.

Let's start by adding a *MTEXT*. entity to the drawing and placing it at the left of the chalice. (*Draw -> Text*) To the text input field add the following text:

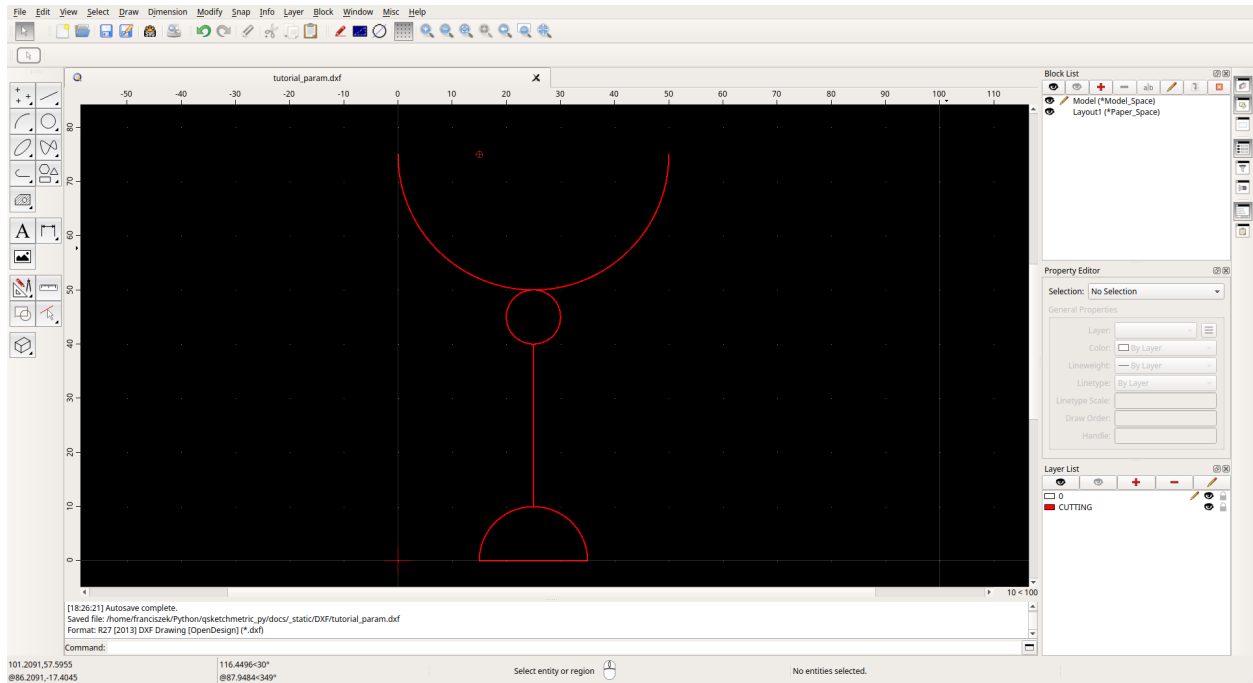
Available variables:

```
----- build in -----
```

```
c: const
```

```
?: undefined
```

(continues on next page)



(continued from previous page)

```
h: chalice_height
```

```
----- custom -----
```

Where:

- ‘`—` build in `—`’: are the built-in variables it servers as a documentation for the user. We added `h` variable to know what is the variable name for the chalice height.
- ‘`—` custom `—`’: are the helpers variables that we will add later.

The next step will be adding a new layer called *VIRTUAL_LAYER* (Layer -> Add Layer) and drawing *LINE* entities on it. (Draw -> Line -> Line from 2 Points) With those lines join all the points of a chalice to each other to form a cohesive graph.

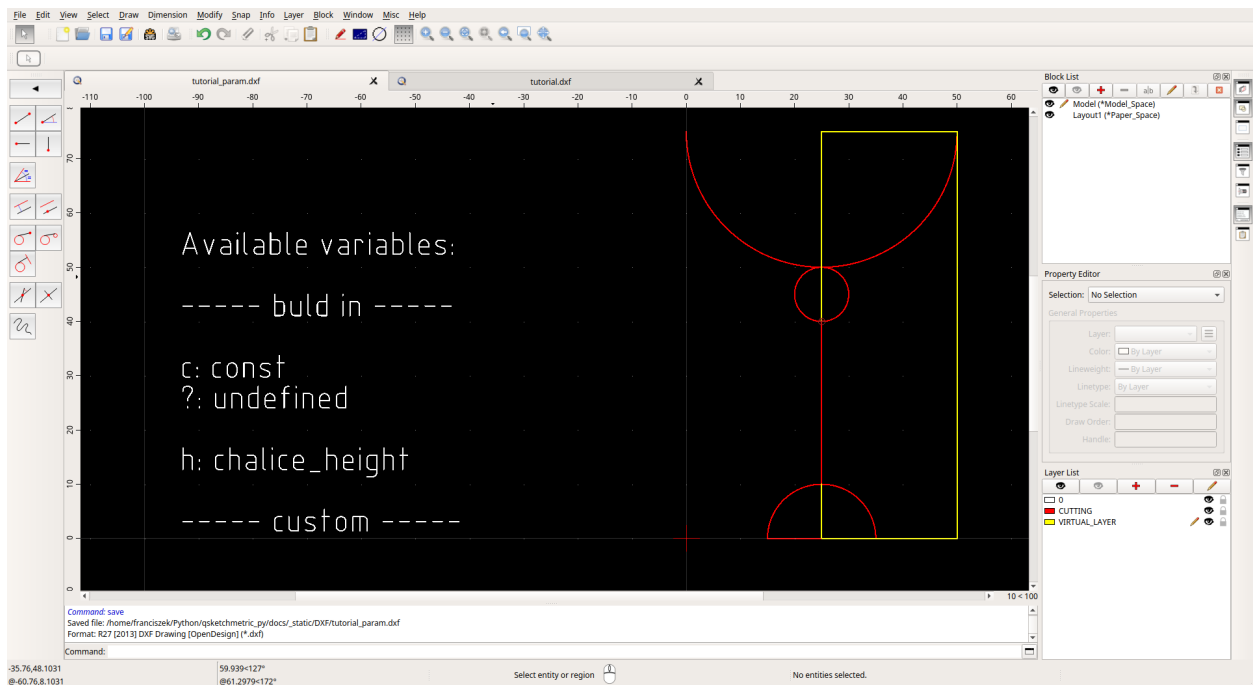
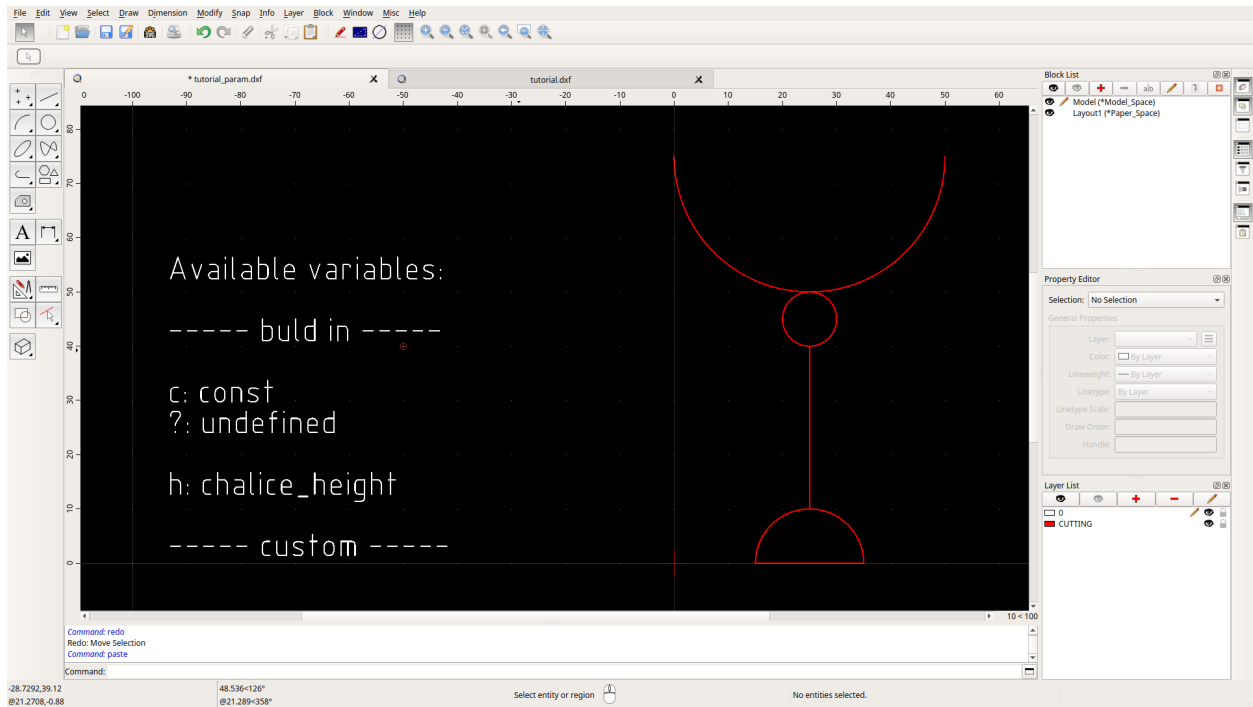
- CIRCLE - by their center point
- LINES - by at least one of their end points
- ARC - by their center point

After you are done, you should have something like this:

We are nearly finished. The last step is to add parameters to the drawing. But first, let’s make our job easier by defining a few helper variables. In the `— custom —` section of the *MTEXT* entity add the following variables:

```
----- custom -----
```

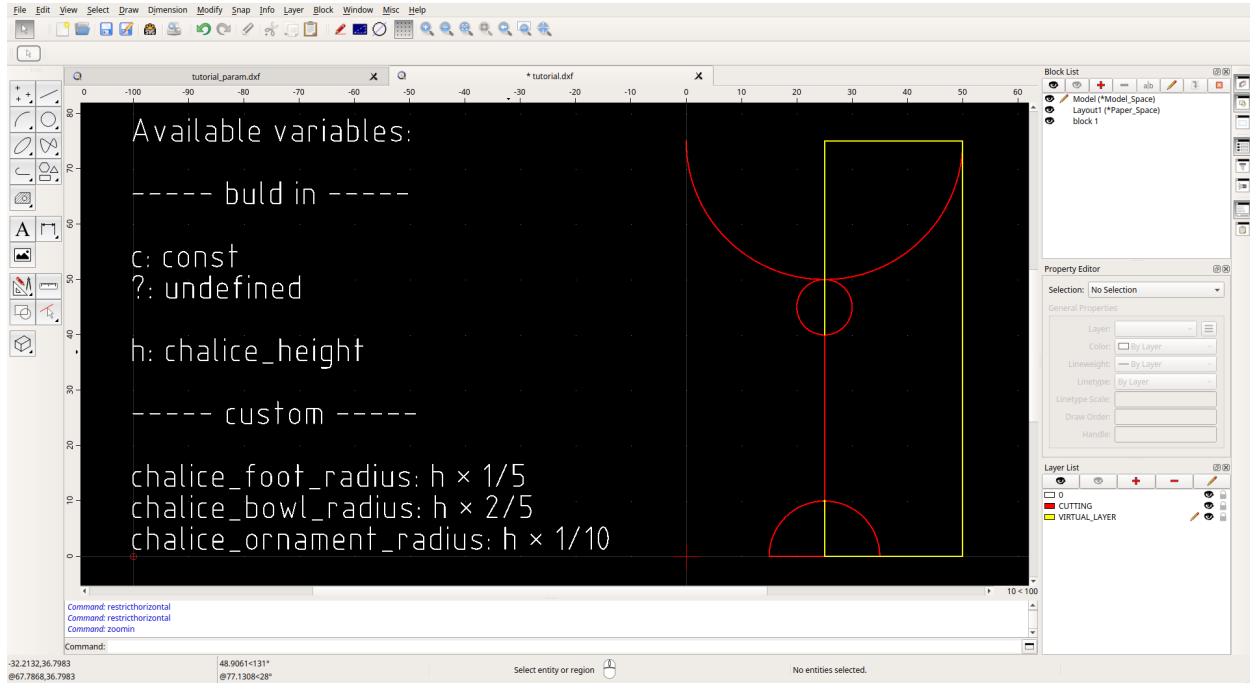
```
chalice_foot_radius: h * 1/5
chalice_bowl_radius: h * 2/5
chalice_ornament_radius: h * 1/10
```



Chalice arc-bowl, arc-foot and circle-ornament radius's are defined as a fraction of the chalice height. This way, if we change the chalice height, the radius's will change accordingly.

We did not define the chalice leg length because it will be calculated automatically by the renderer.

After adding the variables, everything should look like this:



Now we can add parameters to the drawing. To do so select the entities one by one and scroll down the Property Editor to the Custom section. Click on the red plus button and add the parameter.

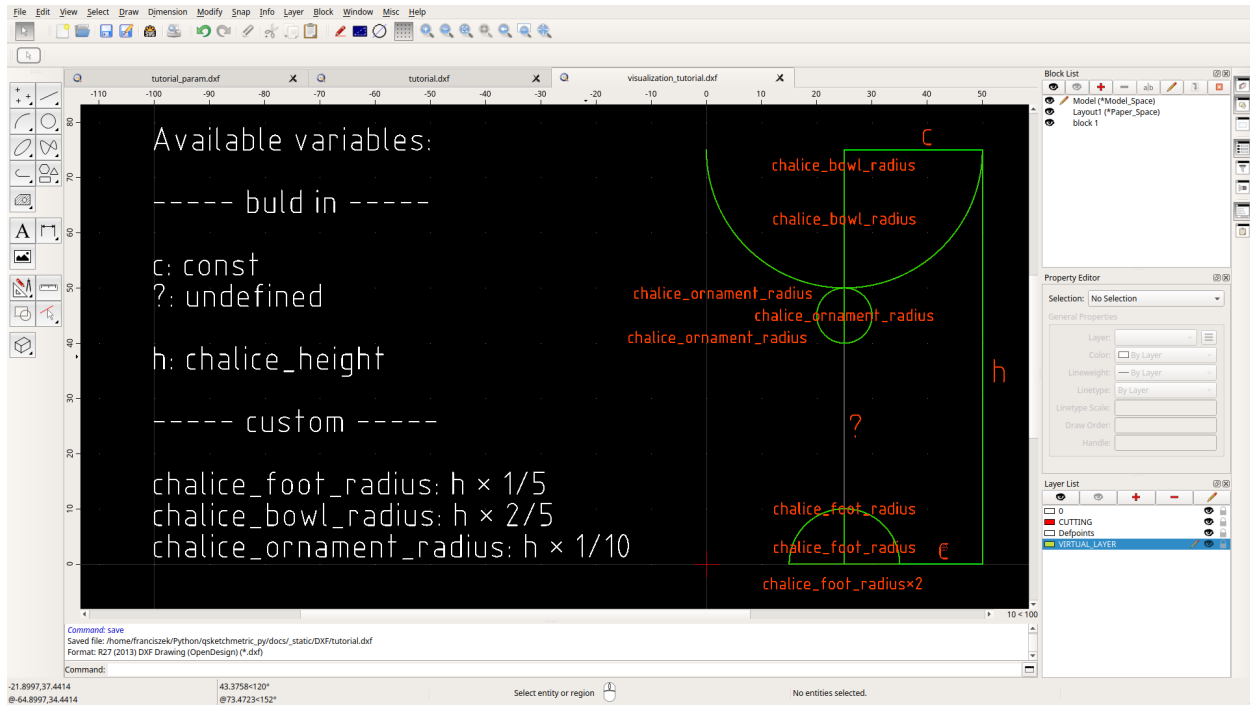
- Name must be: *c*.
- Value contains the expression describing the entity. According to this table below

Value	Description
c	(constant) Entity length will not change
?	(undefined) Entity length will be calculated by the renderer. Only if there is other path to the both end points of the line!
c/ h*2	(math expression) Entity length will be calculated from the math expression

Attention: Remember that our goal is to parametrize the drawing depending on the chalice height.

To parametrize the drawing depending on the chalice height, the Value for the virtual line on the right side of the chalice must be *h* and for the chalice leg line must be *?*. By doing so, we are telling the renderer to calculate the length of the chalice leg line from two end points of the line.

Visual representation of the parametrized drawing:



Warning: It is just a visual representation of the parametrized drawing. It does not represent the actual look of the parametrized drawing. Actual look of the parametrized drawing doesn't change after the parametrization!

Now we can save the parametrized DXF file (*File -> Save*) and render it. Finished file should be similar to [tutorial.dxf](#) file, that you can [download](#) from the [QSketchMetric repository](#).

Lastly, we check if the parametrization is correct by validating it. To do so, follow the [Validation your first DXF file](#) tutorial.

Congratulation you created your first parametric DXF file!

3.3 Tutorial - Semi-automatic parametrization of a DXF file

Let's learn by example.

In this tutorial, we will learn how to semi-automatic parametrize a DXF file.

We'll assume you have [QSketchMetric installed](#) already as well as [QCAD Professional](#) and have already done the [rendering tutorial](#) as well as [parametrization tutorial](#)

First download the [tutorial_param.dxf](#) file from the [QSketchMetric repository](#). It is an example of a DXF file that we will parametrize in this tutorial. It is a simple drawing of a chalice. With every entity placed on the CUTTING layer.

To do so, open it and click *Ctrl+S* to save it to your computer. As a convention, we'll assume you saved it in a file called [tutorial_param.dxf](#).

Let's start by firing up command line and starting the python interpreter:

```
python
```

Warning: Remember to activate your virtual environment if you are using one:

```
source .venv/bin/activate
```

Next we need to import the `qsketchmetric.semiautomatic.SemiAutomaticParameterization` module:

```
from qsketchmetric.semiautomatic import SemiAutomaticParameterization
```

Now define the path to the input file we downloaded earlier:

```
input_dxf_path = "tutorial_param.dxf"
```

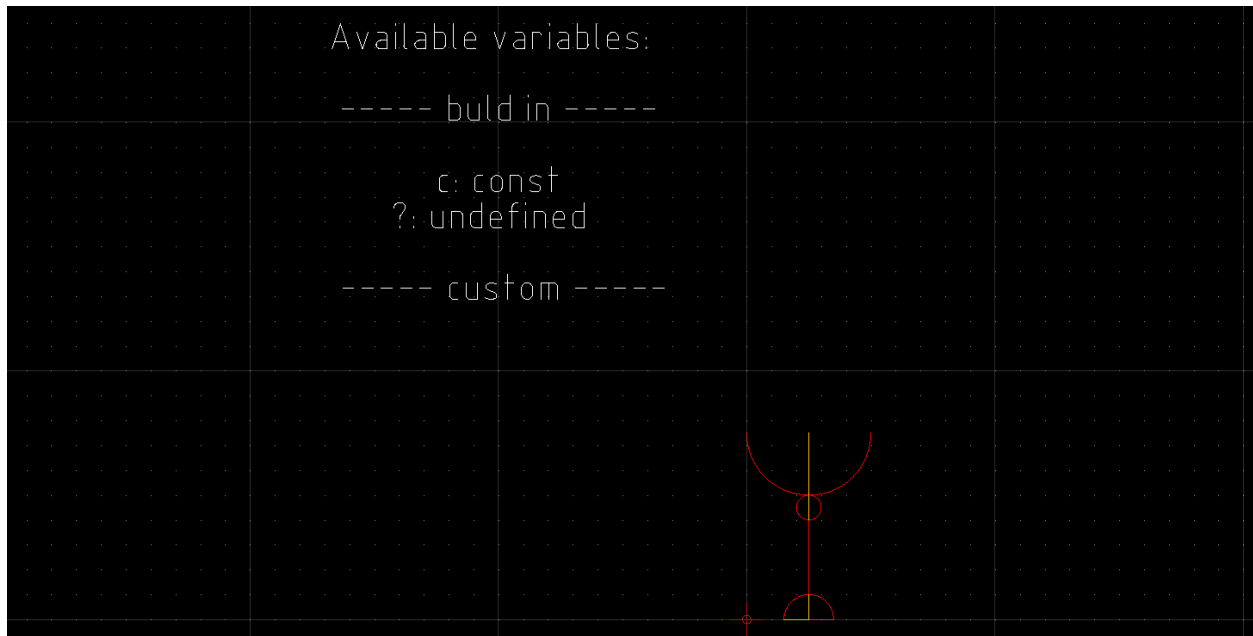
For the output file path and default parameter we will use default settings.

Now we are ready to roll. Let's parametrize the DXF file:

```
sap = SemiAutomaticParameterization(input_dxf_path)
sap.parameterize()
```

Parametrized file will be saved in the `parametric` directory, in the same directory as the input file.

Open the parametrized file in `QCAD Professional`, and edit the parameters.



Every entity parameter is now set to the default parameter. Let's change their values.

To do so select the entities one by one and scroll down the **Property Editor** to the **Custom** section. Click on `c` parameter to edit it.

Value contains the expression describing the entity. According to this table below:

Value	Description
c	(constant) Entity length will not change
?	(undefined) Entity length will be calculated by the renderer. Only if there is other path to the both end points of the line!
c*2/ 5	(math expression) Entity length will be calculated from the math expression

You can change the value of the parameter to any of the above. For example, let's change the value of the chalice leg line to $2*c$. This will make the leg line 2 times longer.

Lastly, we check if the parametrization is correct by validating it. To do so, follow the [Validation your first DXF file](#) tutorial.

That is it! You have successfully parametrized a DXF file. As you can see semi-automatic parametrization is much faster and easier than manual parametrization.

Congratulation!

3.4 Tutorial - Validation your first DXF file

Let's learn by example.

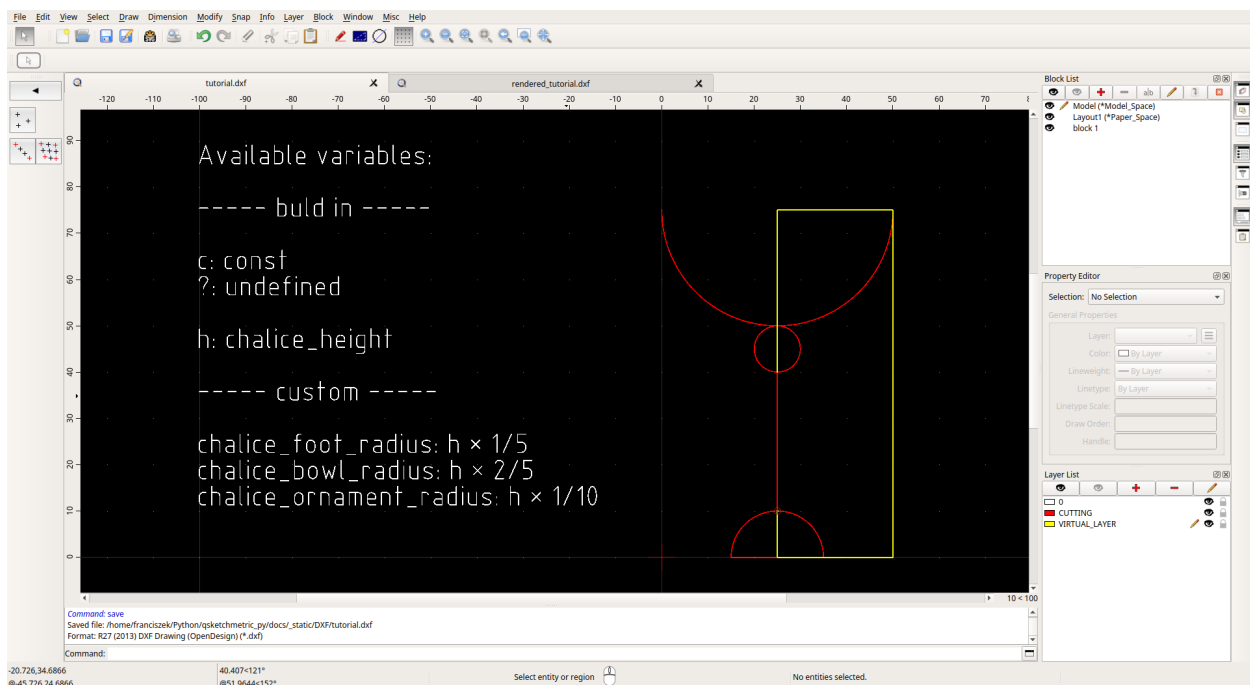
In this tutorial, we will learn how to validate a parametrized DXF file.

We'll assume you have *QSketchMetric installed* already as well as *QCAD Professional*

First (If you don't have already from other tutorials) download the [tutorial.dxf](#) file from the *QSketchMetric* repository. It is an example of a DXF file that we will validate in this tutorial.

To do so, open it and click *Ctrl+S* to save it to your computer. As a convention, we'll assume you saved it in a file called *tutorial.dxf*.

First, let's open the file in QCAD Professional. (*File -> Open*)



As you can see, it is a simple drawing of a chalice. It is parametrized depending on the chalice height. Changing the chalice height h variable will render the drawing accordingly.

Let's validate the drawing. To do so, open the [QSketchMetric Validator](#) and login using your **GitHub** account. It is as simple as clicking the **GitHub** button. The first time you login, you will be asked to authorize the application to access your GitHub account.

After logging in, you will see the following screen:



Click the *choose a file* button and select the *tutorial.dxf* file you saved earlier.

Next, click the *Validate* button. **And what this! An error!**

The error is telling us that the h variable is not defined. This is because the validator does not know what the h variable is while calculating the `chalice_foot_radius` variable.

Download the debug report by clicking the [Debug report](#) button and open it in QCAD Professional.

We can see that every entity got greyed out except of the MTEXT entity. It is because the MTEXT entity is the place where the error occurred while calculating the `chalice_foot_radius` variable. Also the error message is displayed in the right bottom corner of the drawing.

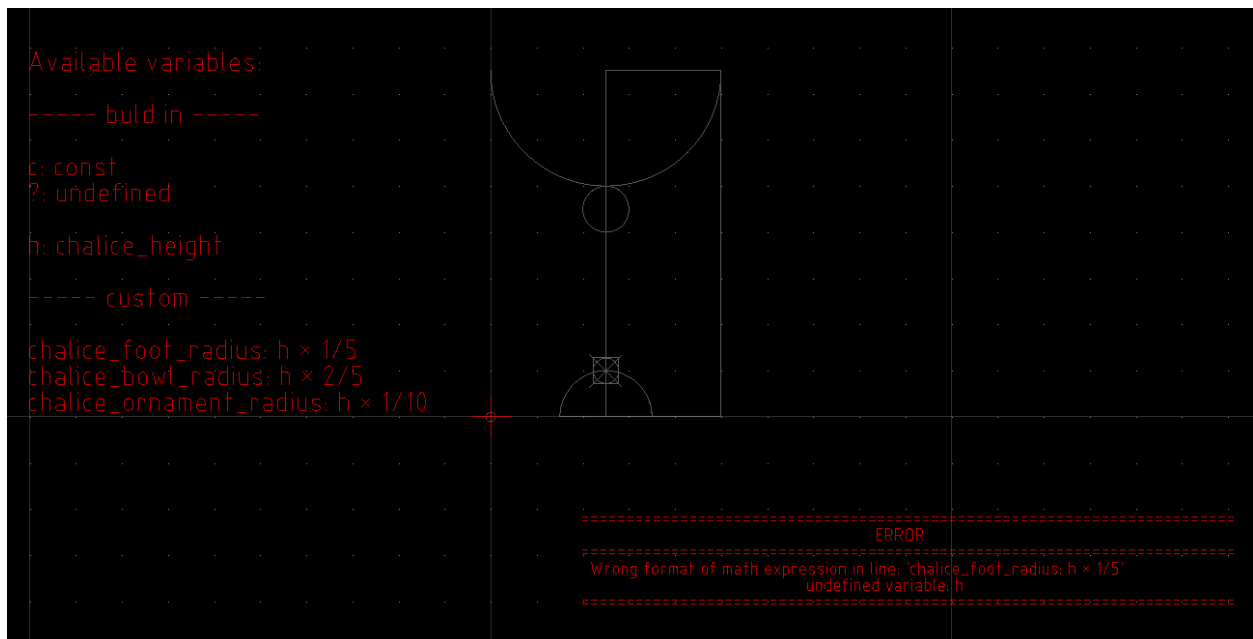
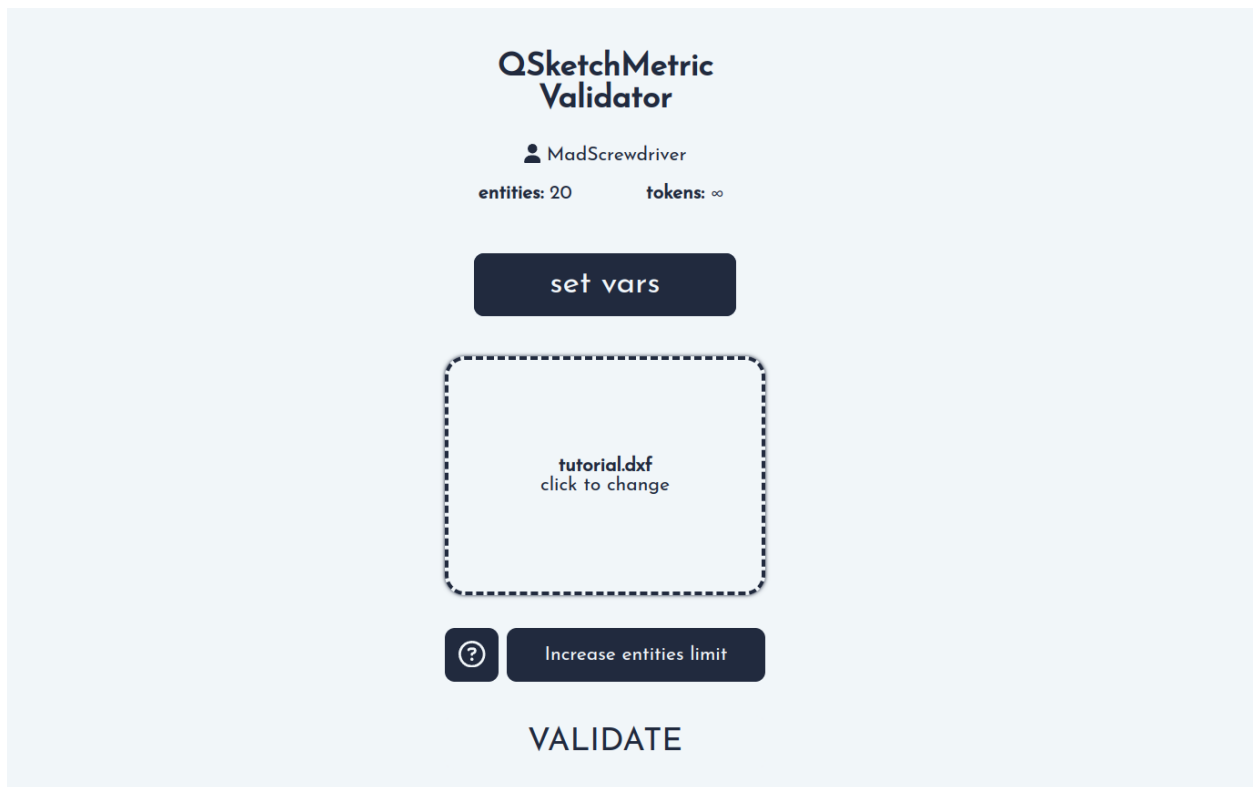
Let's fix the error!

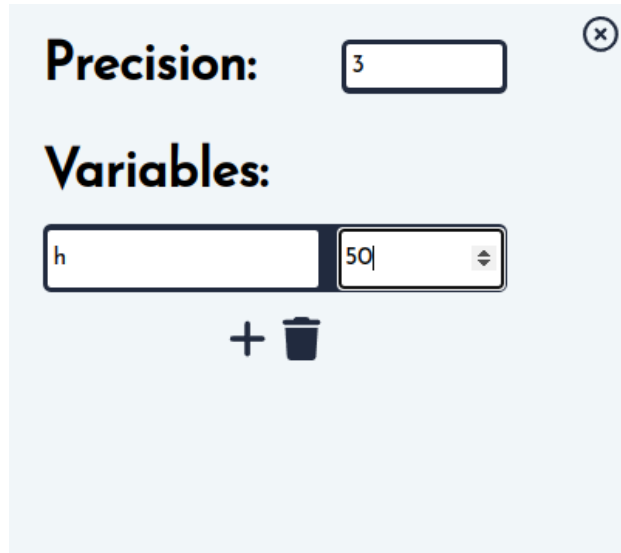
To do so, we need to define the h variable. Click the *validate another file* button and select the *tutorial.dxf* file again. This time, before clicking the *Validate* button, click the *set vars* button. The modal window will appear.

Add new variable using the **+** symbol. In the *name* field, type h and in the *value* field, type 50.

Close the modal window and click the *Validate* button. **A success!**

Entities, Variables and Cohesion checks are all green and we are presented with a success message.





Congratulation you validated your first parametric DXF file!

3.5 Tutorial - Rendering a point

Let's learn by example.

In this tutorial, we will learn how to render a point using the `qsketchmetric.renderer.Renderer` module.

We'll assume you have *QSketchMetric installed* already as well as *QCAD Professional* and have already done the *rendering tutorial* as well as *parametrization tutorial*

First download the `tutorial.dxf` file from the *QSketchMetric repository*. It is an example of a parametric DXF file that we will use in a tutorial.

To do so, open it and click Ctrl+S to save it to your computer. As a convention, we'll assume you saved it in a file called `tutorial.dxf`.

Open `tutorial.dxf` in QCAD Professional, the result should look like this:

Rendering a point is dead simple with QSketchMetric. All you need to do is to create a POINT entity on the *VIRTUAL_LAYER*. (Draw -> Point -> Single Point) **The ``POINT`` must be connected to the other entities!**

Next you need to add a parameter to the point. To do so select the point and scroll down the Property Editor to the Custom section. Click on the red plus button and add the parameter.

- Name should be: *name*.
- Value should be: *variable_name* you desire.

variable_name will be returned by the renderer with new rendered coordination of the point.

Added point should look like this:

That is all! Now you can save the file and render it with `qsketchmetric.renderer.Renderer.render()` method:

```
from qsketchmetric.renderer import Renderer
from ezdxf import new
from ezdxf import units
```

(continues on next page)

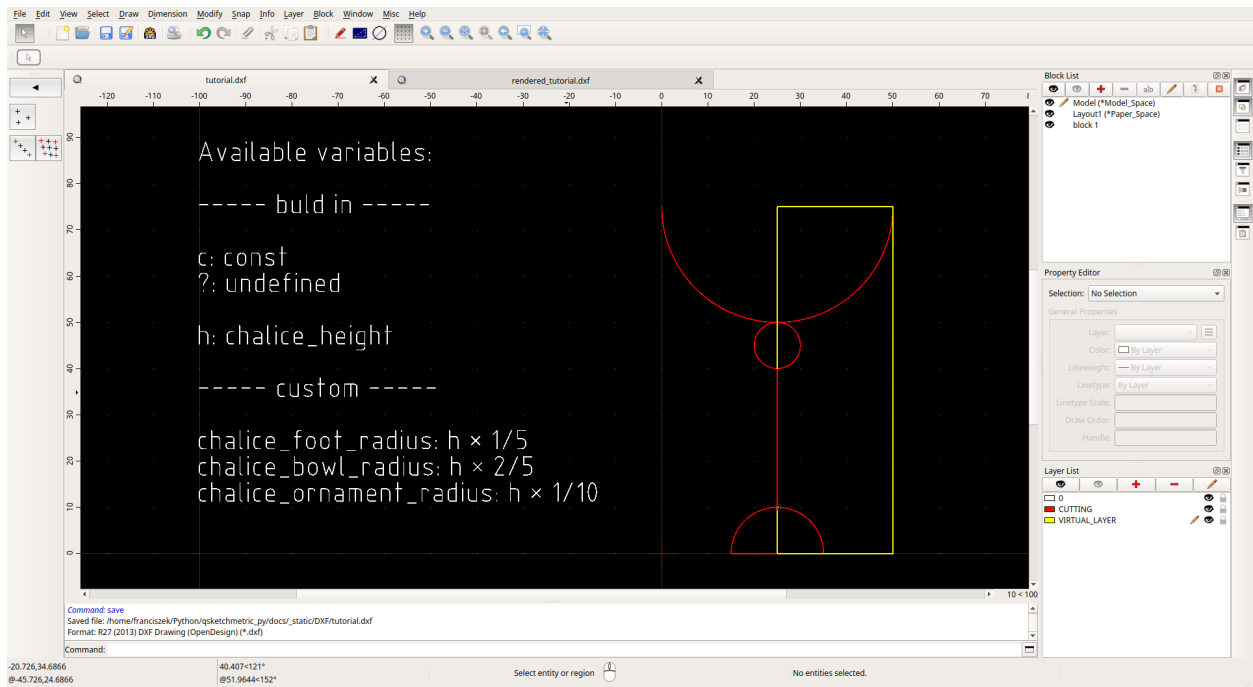


Fig. 3: tutorial.dxf opened in QCAD Professional

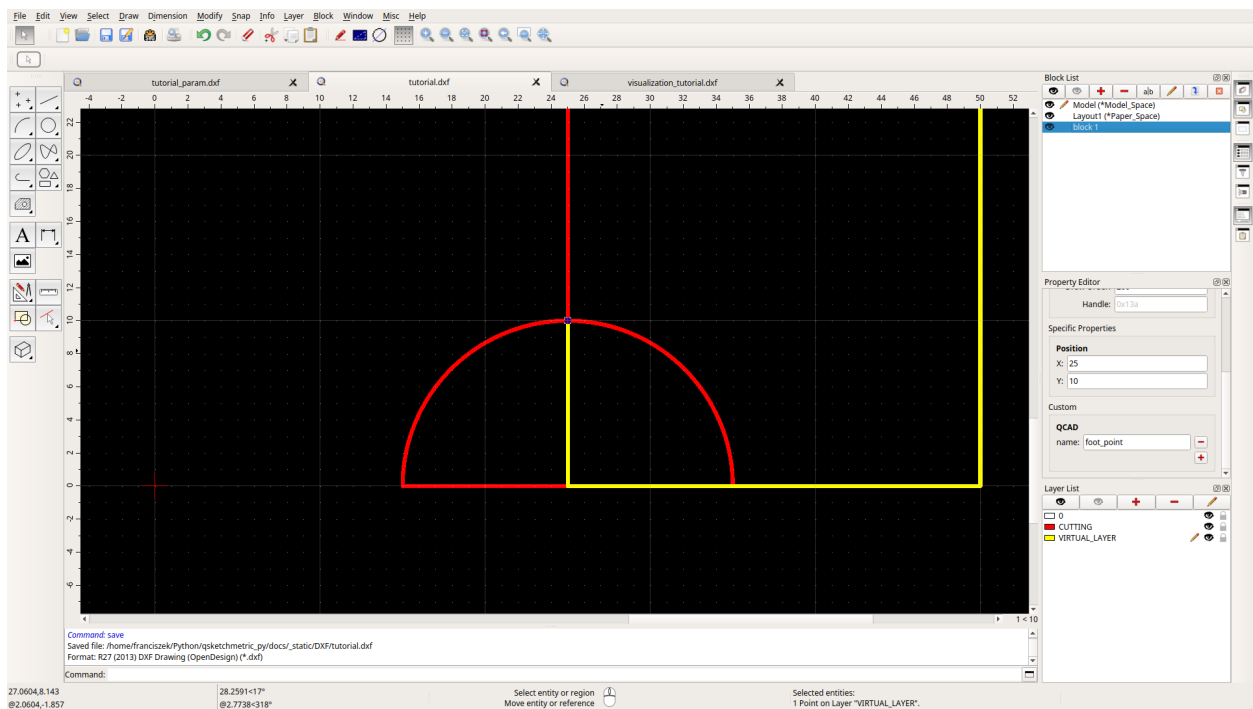


Fig. 4: Added point

(continued from previous page)

```

output_dxf = new()
output_dxf.units = units.MM
variables = {'h': 50}
renderer = Renderer('tutorial.dxf', output_dxf, variables)
variables = renderer.render()
print(variables)

output_dxf.saveas('rendered_tutorial.dxf')

```

tutorial.dxf will be rendered on to the output_dxf `ezdxf.document.Drawing` and rendered variable from the `VIRTUAL_LAYER` will be contained in the variables dictionary with the following content:

```

{
    "foot_point": (20, 10)
}

```

(20, 10) is the rendered coordinate of the point.

Congratulation you rendered your first point using `:meth:`qsketchmetric.renderer.Renderer.render`` method!

3.6 Tutorial - Rendering a custom line style

Let's learn by example.

In this tutorial, we will learn how to render a custom lines style using the `qsketchmetric.renderer.Renderer` module.

We'll assume you have *QSketchMetric installed* already as well as *QCAD Professional* and have already done the *rendering tutorial* as well as *parametrization tutorial*

First download the `tutorial.dxf` file from the *QSketchMetric repository*. It is an example of a parametric DXF file that we will use in a tutorial.

To do so, open it and click Ctrl+S to save it to your computer. As a convention, we'll assume you saved it in a file called `tutorial.dxf`.

Open `tutorial.dxf` in QCAD Professional, the result should look like this:

As you can see, it is a simple drawing of a chalice. With every entity placed on the CUTTING layer.

Rendering a custom line style is easy with QSketchMetric. All you need to do is to add a parameter to the entities. Select the entitie where you want a custom line format and scroll down the **Property Editor** to the **Custom** section. Click on the red plus button and add the parameter.

- Name should be: *line*.
- Value should be a `ezdxf custom complex line pattern format`.

In our example, we will use a a line pattern that looks like this: — BOWL — BOWL —. We will use it for the bowl of the chalice. To do so, Value should be: `A,2,-1,["BOWL",STANDARD,S=.5,U=0.0,X=-0.1,Y=-.05],-2.5`

Where:

- A - every line pattern starts with A
- 2 - line length
- -1 - space length

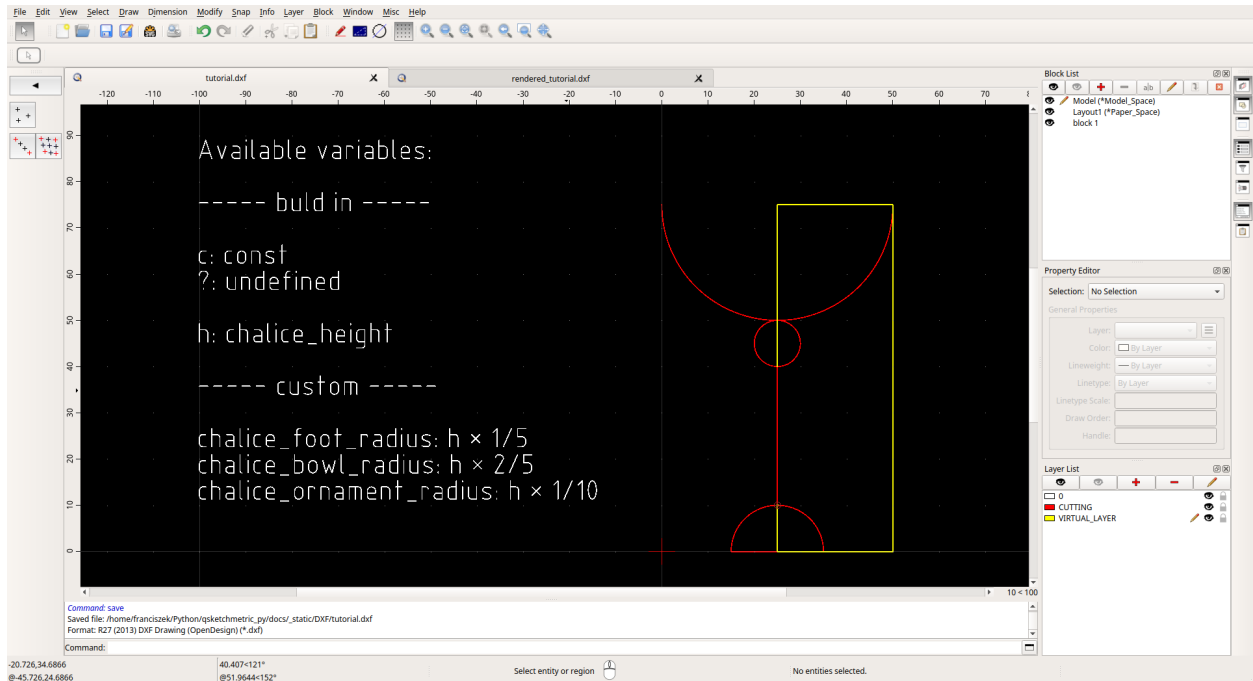


Fig. 5: tutorial.dxf opened in QCAD Professional

- [*"BOWL", STANDARD, S=.5, U=0.0, X=-0.1, Y=-.05*] - *BOWL* part definition
- -2.5 - space length after the *BOWL* part

Note: Remember to add a comma after every parameter and do not use whitespaces.

That is all! Now you can save the file and render it with `qsketchmetric.renderer.Renderer.render()` method:

```
from qsketchmetric.renderer import Renderer
from ezdxf import new
from ezdxf import units

output_dxf = new()
variables = {'h': 50}
output_dxf.units = units.MM
renderer = Renderer('tutorial.dxf', output_dxf, variables)
renderer.render()
output_dxf.saveas('rendered_custom_line_tutorial.dxf')
```

Note: Remember to make sure that the output and input DXF files are configured in the same units. That is why we set the units of the output DXF file to MM.

Rendered file should look like this:

Congratulation you renderer your first custom line!

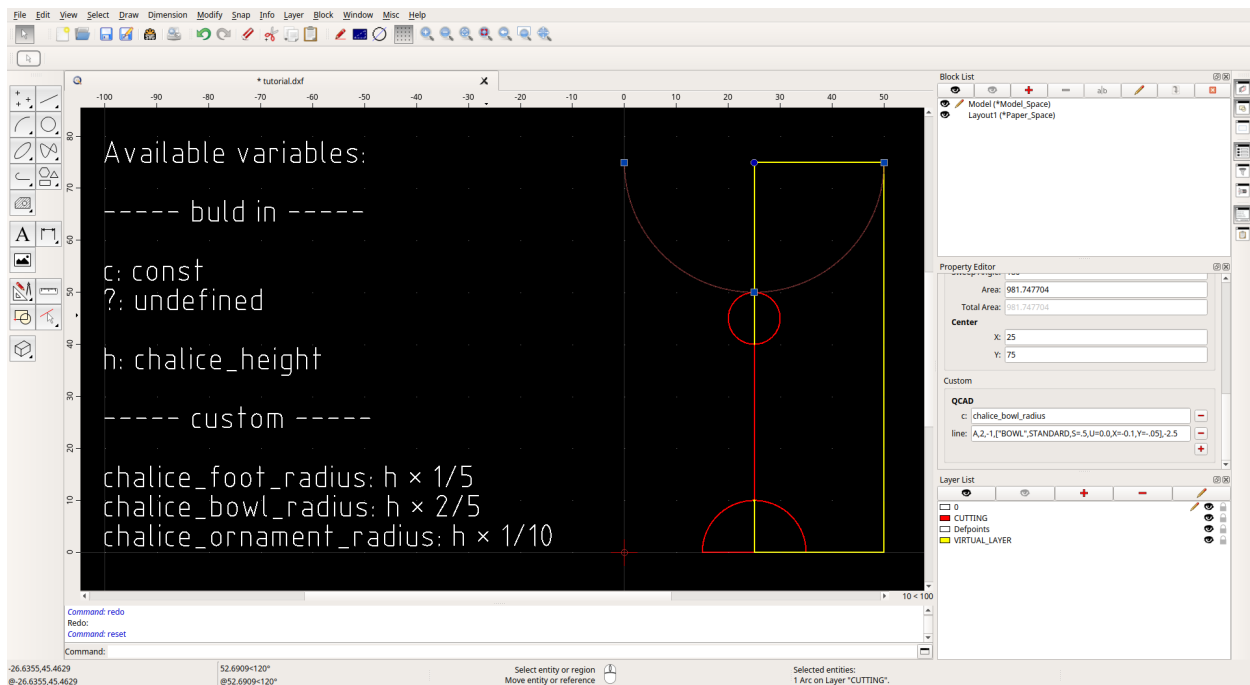


Fig. 6: `tutorial.dxf` with added `line` parameter

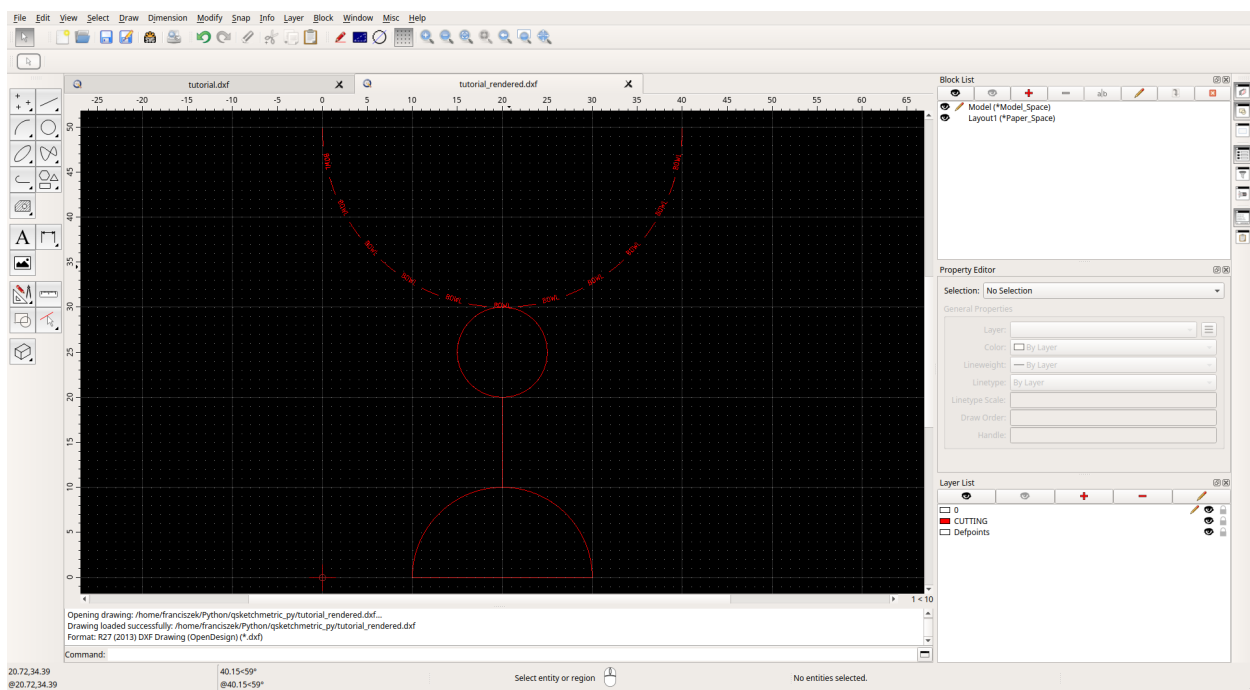


Fig. 7: `rendered_custom_line_tutorial.dxf` opened in QCAD Professional

HOW-TO GUIDES

How-to guides are recipes. They guide you through the steps involved in addressing key problems and use-cases. They are more advanced than tutorials and assume some knowledge of how QSketchMetric works.

4.1 Manual parametrization

4.1.1 Supported DXF entities

QSketchMetric explicit supports the following DXF entities: LINE, CIRCLE, ARC, POINT, INSERT entities. Other entities such as LWPOLYLINE, POLYLINE, SPLINE, ELLIPSE, MTEXT, TEXT **etc.** can be also parametrized, using the INSERT entity.

4.1.2 What is needed?

- [QCAD Professional](#) is a commercial software, but it offers a free trial version. It is needed to embed the parameters into the DXF file. Community version of QCAD does not support this feature.
- A [DXF](#) file to parametrize.

4.1.3 Manual parametrization

1. Open the DXF file in QCAD Professional. (*File -> Open*)
2. Add new layer called [VIRTUAL_LAYER](#) (*Layer -> Add Layer*)
3. Add [MTEXT](#) entity containing names of the variables passed to the renderer and variables added during parametrization. It can be placed anywhere. See [MTEXT](#) to get more information about the format of the entity. (*Draw -> Text*)
4. **Connect entities. Entities must be connected to each other.**
 - **CIRCLES** - by their center point
 - **LINES** - by at lest one of their end points
 - **ARCS** - by their center point
 - **POINTS** - by their center point
 - **INSERTS** - by their insertion point

To achieve this add [LINE](#) entities (*Draw -> Line*) on to the [VIRTUAL_LAYER](#). Those lines will connect the entities together and form one coherent graph. They won't be rendered in the final DXF file.

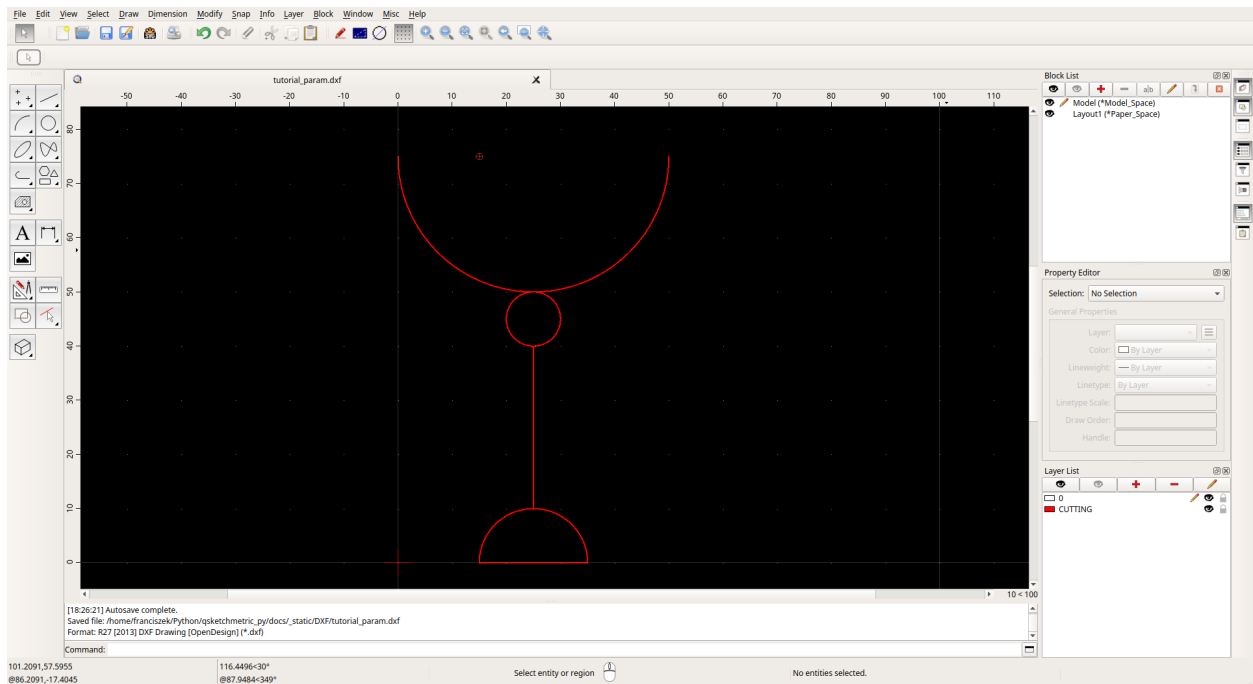


Fig. 1: QCAD Professional with DXF file to parametrize opened

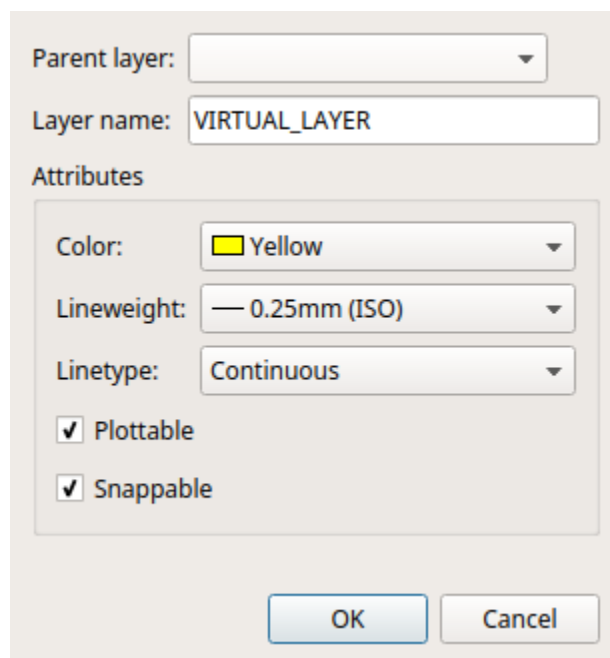


Fig. 2: QCAD Professional layer adding window

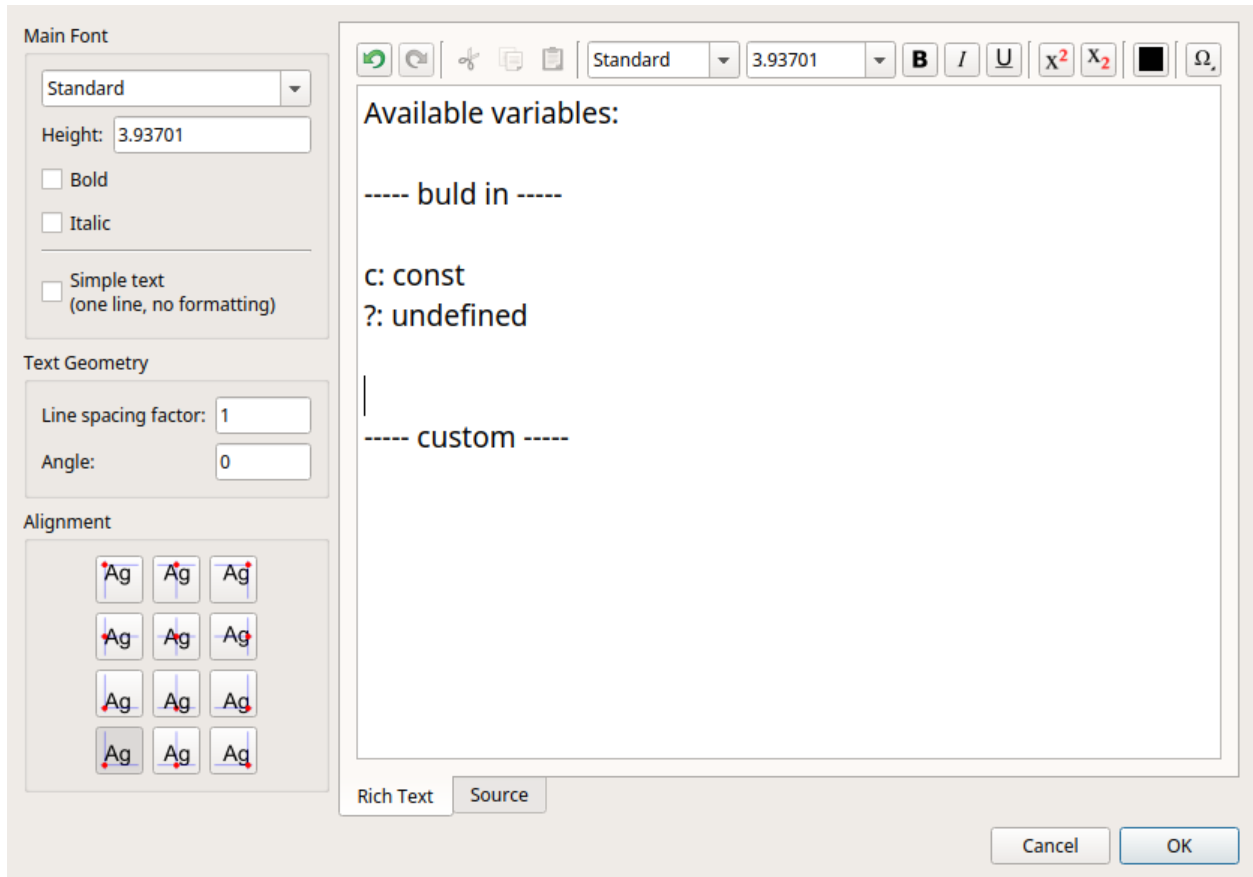


Fig. 3: QCAD Professional with MTEXT dialog window opened

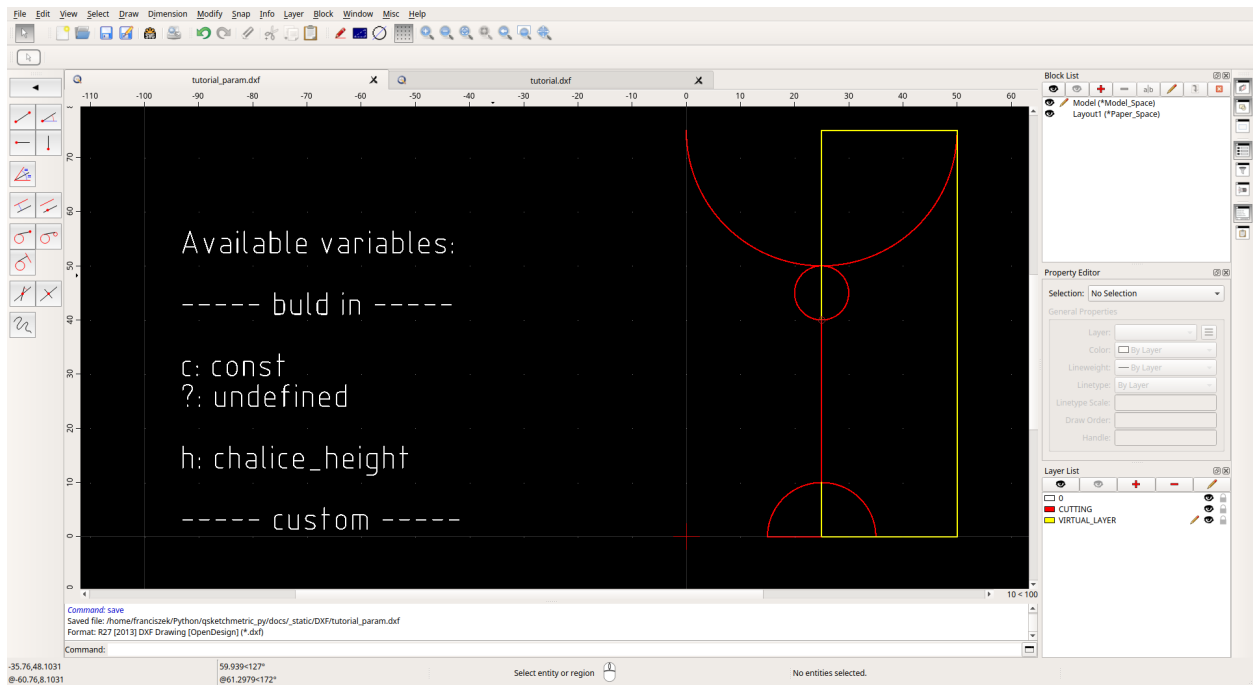


Fig. 4: DXF drawing connected with lines

5. Final step is to add parameters to the entities. To do so select the entity and scroll down the Property Editor to the Custom section. Click on the red plus button and add the parameter.

- **LINE, CIRCLE and ARC**

- Name must be: *c*.
- Value contains the expression describing the entity. According to the table below.

Variables from the *MTEXT* entity can be used, as well as math expressions provided by [this list](#).

There is an option to add **optional line variable**. This variable states the custom line style of the entity. Value should be in a format of *ezdxf complex line pattern format*. See [ezdxf documentation](#) for more information about the format. Value example: *A,2,-1,["BOWL",STANDARD,S=,5,U=0.0,X=-0.1,Y=-.05],-2.5*

- **INSERT**

- Name must be: *c*.
- Value contains the expression describing width and height of the entity split by a @ sign. In the format: *width@height*. Both width and height are math expressions (see above) where ? is only allowed for the one of the dimensions. For example: *c*3@?* or *?@200*sqrt(20)*. For the ? dimension the renderer will calculate the value to fit the aspect ratio of the entity.

Note: Entities on *VIRTUAL_LAYER* contained in INSERT entity will not be rendered but they will be taken into account while calculating the width and height of the INSERT entity. This is useful to make calculations easier.

For example: To parametrize a part of the ellipse, full ellipse on the *VIRTUAL_LAYER* can be drawn on top. This way by parametrizing the full ellipse the part will be rendered according to the full ellipse size. In many scenarios it is easier to parametrize.

- **LWPOLILINE, POLYLINE, SPLINE, ELLIPSE, MTEXT etc.**

- Those entities must be packed into INSERT entity and parametrized as described above.

- **POINT**

- Name must be: *name*.
- Value contains the name of the variable. This variable will be returned by the *qsketchmetric.Renderer.Renderer.render()* in a dictionary.

6. Validate the file. This can be done by using the *QSketchMetric Validator*. See [Validating a parameterized DXF file](#) for more information.

4.2 Semi-automatic parametrization

The *qsketchmetric.semiautomatic.SemiAutomaticParameterization* module is used to semi-automatic parameterize a DXF file. By semi-automatic, it means that the user has to manually customize the parameters of each entity after the automatic parameterization process. Process includes:

- Adding *MTEXT* entity.
- Adding *VIRTUAL_LAYER* layer.
- Adding default expression to each entity.
- Joining entities with virtual lines in to the one coherent graph.

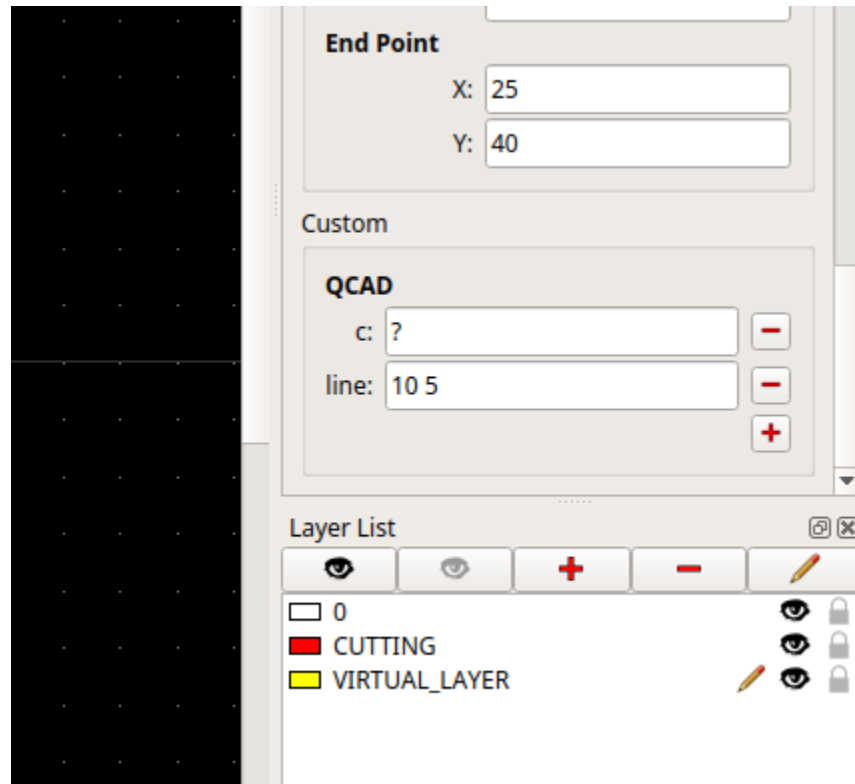


Fig. 5: LINE entity with parameters

1. Make sure entities that are not *explicitly supported* are packed in to an INSERT entity (block). If not, pack them using **QCAD Professional** software. Otherwise, the entities will be deleted during the semi-automatic parametrization process.
2. If don't have already fire up a terminal and run python console:

```
$ python
```

3. Define the path to the DXF file to parametrize:

```
input_dxf_file = 'path/to/dxf/file.dxf'
```

4. **(Optional) Define the path to the output parametrized DXF file.**

Default is `input_dxf_file` with `_param` appended to the file name contained in the *parametric* directory:

```
output_dxf = 'path/to/output/parametrized_dxf/file.dxf'
```

5. **(Optional) Define the expression that will be used to parametrize each entity. Default is `c` which stand for current length.** See allowed expressions on the *manual parametrization page*..

```
expression = '?'
```

6. Parametrize the DXF file:

```
from qsketchmetric.semiautomatic import SemiAutomaticParameterization
input_dxf_file = 'path/to/dxf/file.dxf'
```

(continues on next page)

(continued from previous page)

```
# Optional
output_dxf = 'path/to/output/parametrized_dxf/file.dxf'
expression = '?'

sap = SemiAutomaticParameterization(input_dxf_file, default_value=expression,
    ↪output_dxf_path=output_dxf)
sap.parametrize()
```

7. Open the parametrized file in [QCAD Professional](#), and customize the parameters. Same as in the *manual parametrization* process.
8. Validate the file. This can be done by using the [QSketchMetric Validator](#). See *Validating a parameterized DXF file* for more information.

4.3 Validating a parametrized DXF file

4.3.1 QSketchMetric Validator

To verify the proper parametrization of a DXF file, use the [QSketchMetric Validator](#). It is a web application that allows to upload DXF file and check if it is properly parametrized. In the event of an error, the app will provide full debug report. Including place where the error occurred in the DXF file and the error message.

4.3.2 Validation process

1. Go to [QSketchMetric Validator](#) and login using your **GitHub** account. For widgets and fields explanation (for example: tokens) see the *Widgets* section.

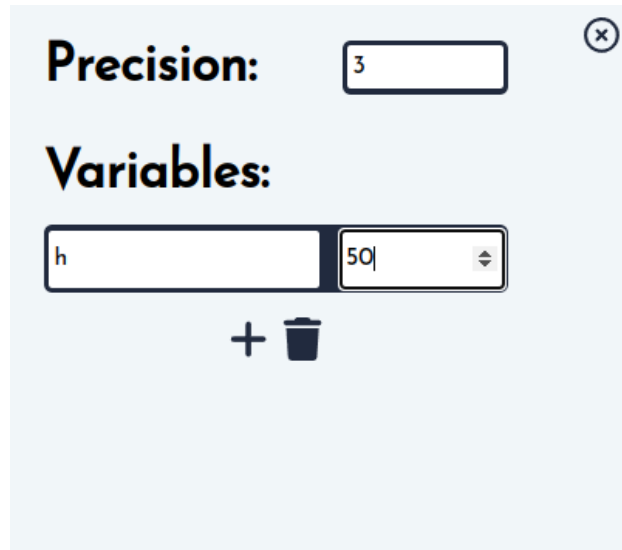


2. Upload a DXF file by clicking on the **Choose a file** button or drag and drop a file into the upload area.



3. Provide a variables needed for the parametrization. This are the variables on upon which the file is rendered. To do so utilize the **set vars** button and add as many variables as needed.

Here the precision of the calculations can be set up as well. Default precision is 3 decimal places.



Variables:	
h	50

4. Click on the **Validate** button.

VALIDATE

5.
 1. If the file is properly parametrized, the app will display a message **validating succeeded** and the rendered file will be available for download.
 2. If the file is not properly parametrized, the app will display an *error message* and a *debug report* will be available for download.
 3. If the DXF file contains more entities than your user account allows, the app will display an **error message** saying that the file contains more entities than the account allows. In this case, see the *Increase entities limit* section.

4.3.3 Increase entities limit

Should your project require a higher entity count, kindly reach out to franciszek@lajszczak.dev . Please provide your desired **entity limit**, your account **username**, and the number of validations (**tokens**) needed.

4.3.4 Widgets

- **Choose a file** - button that allows to choose a file from your computer.



- **Validate** - button that starts the validation process.
- **Set vars** - button that allows to set variables needed for the parametrization.
- **Entities** - field that displays the number of entities in a DXF file that can be validated with the account.
- **Tokens** - field that displays the number of validations that can be performed with current entity limit. After each validation the number of tokens is decreased by one. When the number of tokens reaches zero, the user will revert to the default entity limit of 20 entities.
- **Increase entities limit** - button that tooltip with a *Increase entities limit* section.
- **Question mark** - button that displays a tooltip with a help center.
- **Logout** - button that logs out the user.

4.4 Rendering a DXF file

0. Consider validating the DXF file before rendering it. This can be done by using the [QSketchMetric Validator](#). See <Validating a parameterized DXF file validator> for more information.

1. If don't have already, create a `ezdxf.document.Drawing` object to which the renderer will render:

```
from ezdxf import new
output_dxf = new()
```

Warning: Remember to make sure that the output and input DXF files are configured in the same units. If not, you can change the units of the output DXF file by:

```
output_dxf.units = units.MM
```

`ezdxf` by default uses meters as the unit of measurement.

2. **(Optional)** Define the variables that are described in ---- **build in** ----- section of *MTEXT* entity:

VALIDATE

set vars

```
variables = {'variable_name': 100}
```

3. Define the path to the parametrized DXF file:

```
path = 'how_to_guide.dxf'
```

4. **(Optional)** Define an offset for the rendered entities:

```
offset = (50, 50)
```

5. Render the file:

```
from ezdxf import new
output_dxf = new()
output_dxf.units = units.MM

path = 'how_to_guide.dxf'

# Optional
variables = {'variable_name': 100}
offset = (50, 50)

renderer = Renderer(
    path,
    output_dxf,
    variables=variables,
    offset=offset
)
rendered_points = renderer.render()
```

Note: `qsketchmetric.renderer.Renderer.render()` method renders parametric DXF on to the `output_dxf` and returns rendered points from `VIRTUAL_LAYER` as a Dictionary

entities: 20

tokens: ∞

Increase entities limit



EXPLANATION

Explanation guides discuss key topics and concepts of QSketchMetric.

5.1 Debug report

5.1.1 Error messages

If error is related to specific entity, debug report contains detailed error message with **handle** of problematic entity.

- **Variables validation error** - Error occurs in various situations when parsing variables. **For example:** wrong variable name, wrong variable type, undefined variable etc.

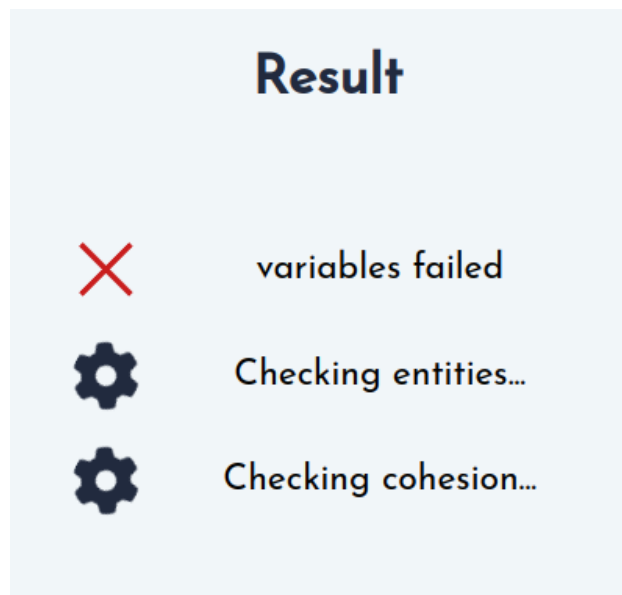


Fig. 1: Variables error

- **Entities validation error** - Error occurs when validating entities. **For example:** wrong entity type, wrong QCAD description, zero length line etc.
- **Cohesion validation error** - Error occurs when validating cohesion. By cohesion we mean that all entities are connected to each other. Either directly or indirectly using *VIRTUAL_LAYER*. It is crucial that all entities form one connected graph to be able to find relative position of all entities. **For example:** line parametrized with '?' is not connected on both ends.

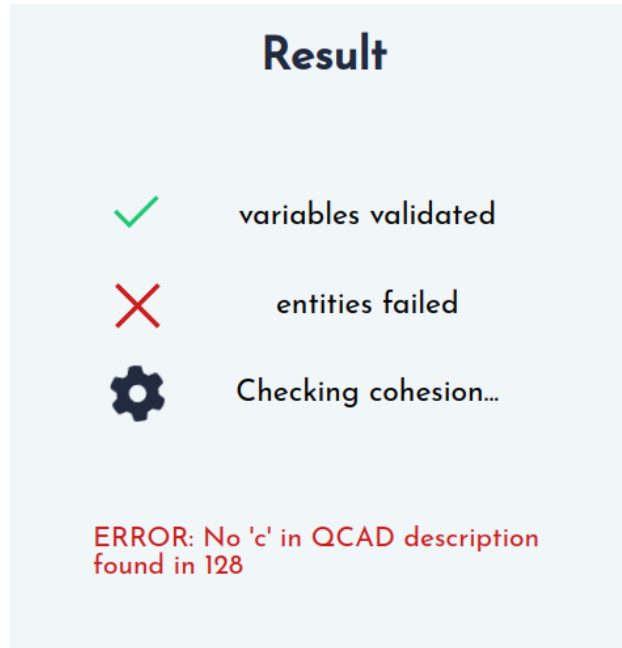


Fig. 2: Entity error

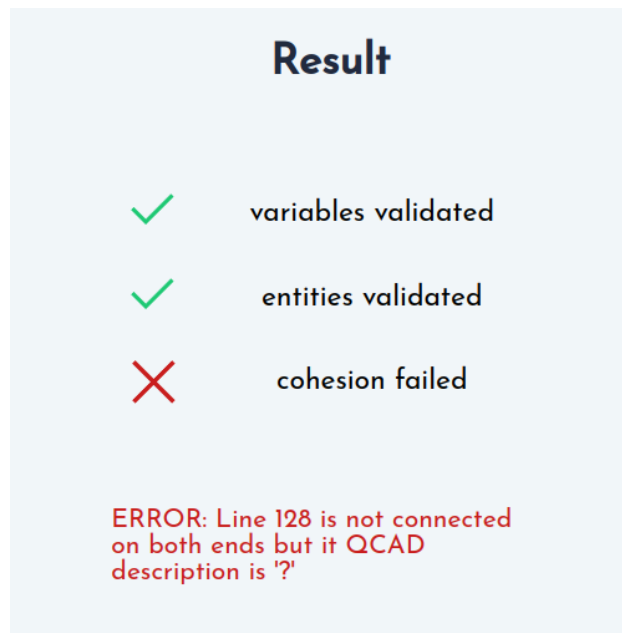


Fig. 3: Cohesion error

- **Limit validation error** - Error occurs when user exceeds his limit of entities. **For example:** user has limit of 100 entities and his drawing has 150 entities. In this case see *Increase entities limit*.

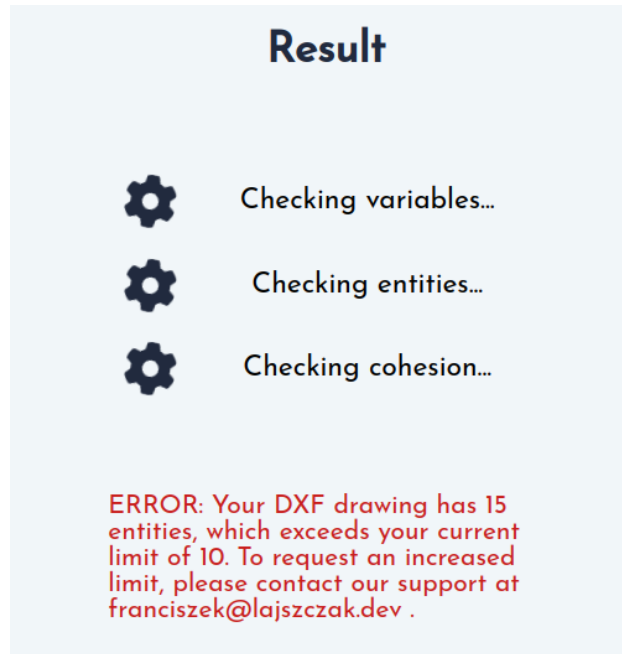


Fig. 4: Limit error

5.1.2 Dxf debug report

Debug report itself is a DXF file. It contains all entities from input dxf file and additional information such as error message in the bottom right corner of the report.

Every entity is grayed out except problematic entity. Problematic entity is highlighted with signature color and is placed on **DEBUG** layer.

In the case when more than one entity is problematic all those entities are placed on **DEBUG** layer and are highlighted with signature color. **For example:** if there is cohesion error such as there are two separate graphs, both graphs are highlighted with different color.

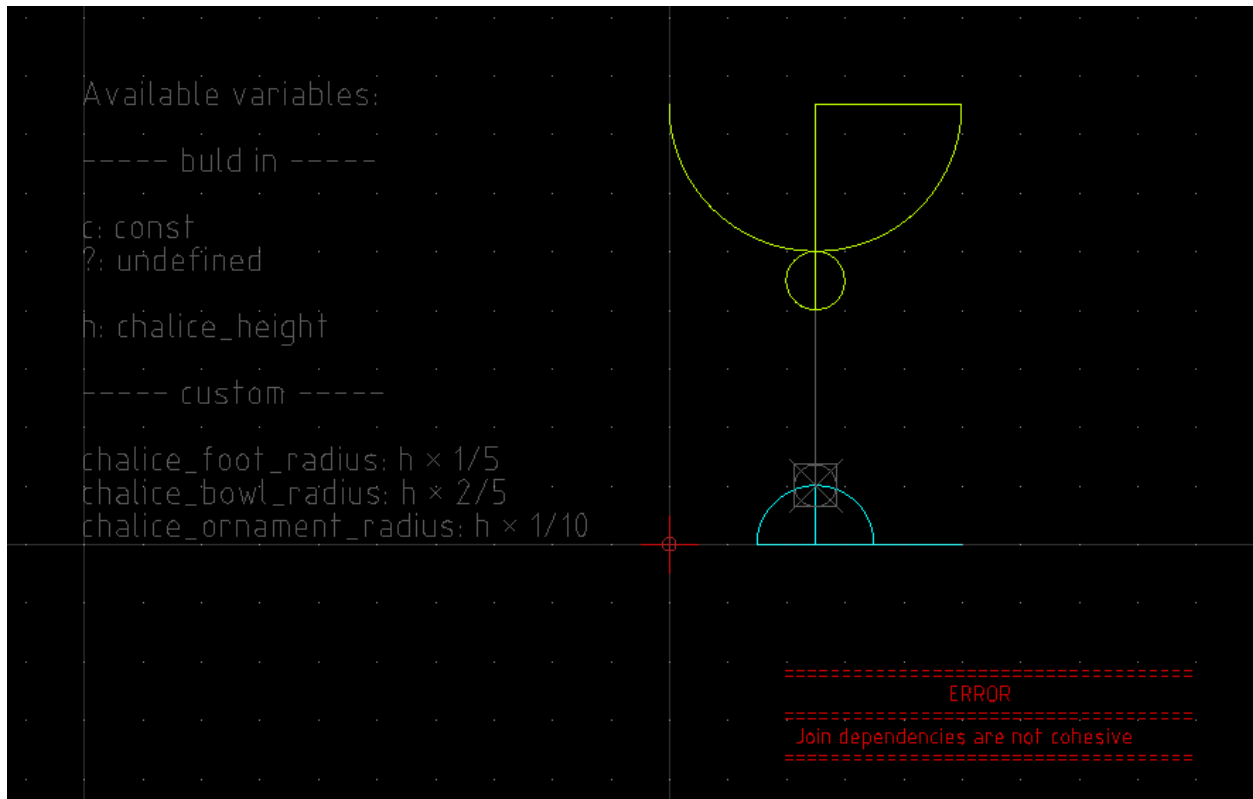
5.2 MTEXT

Entity MTEXT is used to store the variables passed to the renderer and variables added during parametrization. The format of the text in the entity is as follows:

```
Available variables:

----- build in -----
c: const
?: undefined
<variable_name>: <short_description>

----- custom -----
<variable_name>: <variable_value>
```



Variables in the ----- build in ----- in section are the variables that are passed to the renderer. They can be added for better readability of parametrized DXF file. Custom variables can be also added to the ----- custom ----- section. Those variables might come in handy during the parametrization process - to simplify the expressions describing the entities. During parametrization variables can be used from the ----- build in ----- section as well as from the ----- custom ----- section.

Warning: Only **ONE** explicit MTEXT entity is allowed in the DXF file. If there is a need to add more MTEXT entities they need to be packed into INSERT entity as a part of a block and then parametrized.

Warning: MTEXT entity must be in a exact format as described above. Otherwise the parametrization process will fail.

5.3 VIRTUAL_LAYER

VIRTUAL_LAYER will not be rendered in the final DXF file. It is used to store the virtual entities, which are needed to parametrize the DXF file. Virtual layer will contain LINES and POINTS entities.

- POINTS entities will be rendered and returned as a dict by `qsketchmetric.renderer.Renderer.render()` method in a form of:

```
{
  "variable name": (x, y)
}
```

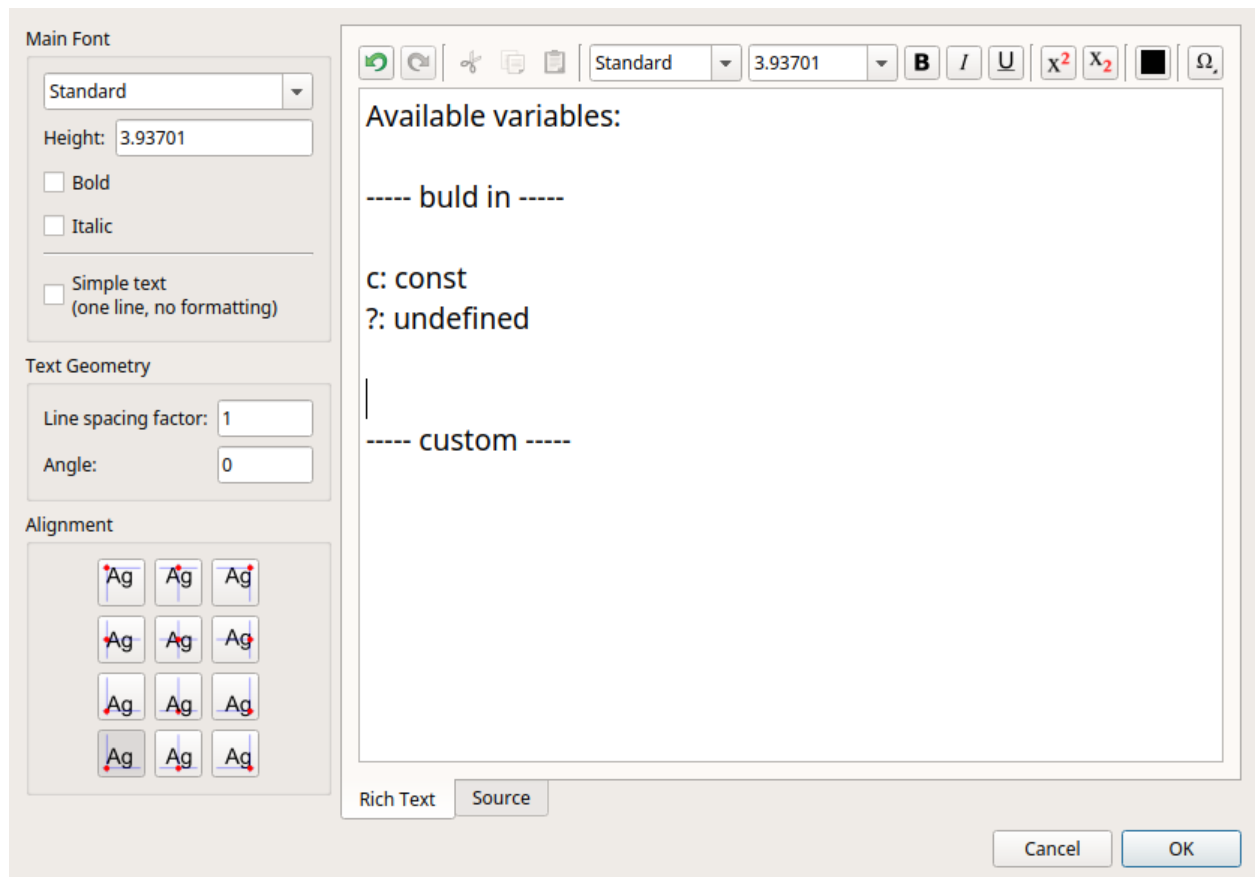



Fig. 5: QCAD Professional with MTEXT dialog window opened

where x and y are the new coordinates of the renderer point.

- LINES entities will be used to join entities together. To form one coherent graph. They will be parametrized but not rendered. Used only to store the information about the relative position between entities.

REFERENCE

Reference guides contain technical reference for all aspects of QSketchMetric's machinery. They describe how it works but assume that you have a basic understanding of key concepts.

6.1 Renderer

```
class qsketchmetric.renderer.Renderer(input_parametric_path, output_rendered_object, variables=None,
                                     offset=(0, 0), accuracy=3)
```

Bases: object

Parameters

- **input_parametric_path** (*Path*) – Path to the parametric file intended for rendering.
- **output_rendered_object** (*Drawing*) – A pre-initialized `ezdxf.document.Drawing` drawing object. You can initialize such an object using methods like `ezdxf.readfile()` or `ezdxf.new()`. By providing an already existing drawing, users can merge multiple visual elements into a singular representation.
- **variables** (*dict[str, float] | None*) – **(Optional)** Supplementary constant variables that can enhance the mathematical representations used. Defaults to an empty dictionary.
- **offset** (*tuple[int, int]*) – **(Optional)** Provides offsets for the parametric visualization. Defaults to (0, 0).
- **accuracy** (*int*) – **(Optional)** The precision used for calculations, represented by the number of decimal places. Defaults to 3.

The *Renderer* class interprets parametric DXF files, transforming them into visual representations.

Warning: Remember to make sure that the output and input DXF files are configured in the same units

See also:

[ezdxf Documentation](#) - A comprehensive library to manage DXF drawings, allowing users to read, write, and modify DXF content efficiently.

get_bb_dimensions(*custom_msp=None*)

Retrieve the bounding box dimensions of the output DXF.

This method calculates the width and height of the bounding box that encompasses all entities within the given Model Space (MSP) or defaults to the output MSP if none is provided.

Parameters

custom_msp – The Model Space to calculate bounding box dimensions for. Defaults to `output_msp`.

Returns

A tuple containing the width and height of the bounding box.

Return type

tuple[float, float]

render()

The main method of the [Renderer](#) class. Transforms the input parametric DXF drawing and produces a rendered output on the output DXF.

Returns

A dictionary containing rendered points marked in the parametric drawing.

Return type

dict[str, tuple[float, float]]

6.2 Semi-automatic parametrization

```
class qsketchmetric.semiautomatic.SemiAutomaticParameterization(input_dxf_path,  
                                                                  default_value='c',  
                                                                  output_dxf_path=None,  
                                                                  accuracy=3)
```

Bases: object

Parameters

- **input_dxf_path** (*Path*) – Path to the DXF file to be parameterized.
- **default_value** (*str*) – **(Optional)** Default expression describing the entities. Defaults to “c”.
- **output_dxf_path** (*Path* | *None*) – **(Optional)** Path for the output parameterized DXF file. If not provided, the output file will be saved in the *parametric* directory, in the same directory as the input file. With the name *input_file_name* + *_param*.
- **accuracy** (*int*) – **(Optional)** The precision used for calculations, represented by the number of decimal places. Defaults to 3.

The `SemiAutomaticParameterize` class is used to semi-automatic parameterize a DXF file. By semi-automatic, it means that the user has to manually customize the parameters of each entity after the automatic parameterization process. Process includes:

- Adding [MTEXT](#) entity.
- Adding [VIRTUAL_LAYER](#) layer.
- Adding default expression to each entity.
- Joining entities with virtual lines in to the one coherent graph.

parametrize()

The main method of the `SemiAutomaticParameterize` class. Parametrizes the DXF file and saves it to the output path.

GETTING HELP

If you have any questions about QSketchMetric or feature request, consider starting a discussion on the [Issues](#) page. You can also contact the author directly via email at franciszek@lajszczak.dev.

Together we can resolve your problem!

HOW TO CONTRIBUTE

Thank you for investing your time in contributing to our project! Any contribution you make will be reflected on [QSketchMetric GitHub Page](#) Hall of Fame . In this guide you will get an overview of the contribution workflow from opening an issue, creating a PR, reviewing, and merging the PR.

8.1 Issues / Feature requests

8.1.1 Create a new issue

If you spot a problem with the package, you have a question or want to request a new feature, it's a good idea to add it as an issue. [Search if an issue already exists](#). If a related issue doesn't exist, you can open a new issue using a relevant issue form.

8.1.2 Solve an issue

Scan through our [existing issues](#) to find one that interests you. You can narrow down the search using labels as filters. See [labels](#) for more information. As a general rule, we don't assign issues to anyone. If you find an issue to work on, you are welcome to open a PR with a fix.

8.2 Make changes

1. [Fork the repo](#) so that you can make your changes without affecting the original project until you're ready to merge them.
2. [Create a new virtual environment](#) for the project and source it.
3. Install the project dependencies using:

```
pip install -r requirements-dev.txt
```

4. Create a working branch and start with your changes!

8.3 Tests

We use `pytest` for testing. You can run the tests using:

```
pytest
```

8.4 Commit your update

Commit the changes once you are happy with them. Don't forget to self-review to speed up the review process. Here are some tips for self-review:

- Confirm that you added tests for your code
- Confirm that you added documentation for your code
- Confirm that the changes meet the user experience and goals outlined in the issue description
- Review the changes for technical accuracy.
- Confirm that the changes are consistent with the project's style and standards adherence to the `mypy`.
- If there are any failing checks in your PR, troubleshoot them until they're all passing.

8.5 Pull request

When you're finished with the changes, create a pull request, also known as a PR.

- Don't forget to link PR to issue if you are solving one.
- We may ask for changes to be made before a PR can be merged. As you update your PR and apply changes, mark each conversation as resolved.
- If you run into any merge issues, checkout this [git tutorial](#) to help you resolve merge conflicts and other issues.

8.6 Your PR is merged!

Congratulations! QSketchMetric develops thanks to people like you. Thank you for your contribution! Once your PR is merged, your contributions will be publicly visible on the [QSketchMetric GitHub Page](#) Hall of Fame .

PYTHON MODULE INDEX

q

`qsketchmetric.renderer`, [39](#)

`qsketchmetric.semiautomatic`, [40](#)

INDEX

G

`get_bb_dimensions()` (*qsketch-metric.renderer.Renderer* method), 39

M

module

`qsketchmetric.renderer`, 39

`qsketchmetric.semiautomatic`, 40

P

`parametrize()` (*qsketch-metric.semiautomatic.SemiAutomaticParameterization* method), 40

Q

`qsketchmetric.renderer`
module, 39

`qsketchmetric.semiautomatic`
module, 40

R

`render()` (*qsketchmetric.renderer.Renderer* method), 40

`Renderer` (class in *qsketchmetric.renderer*), 39

S

`SemiAutomaticParameterization` (class in *qsketch-metric.semiautomatic*), 40