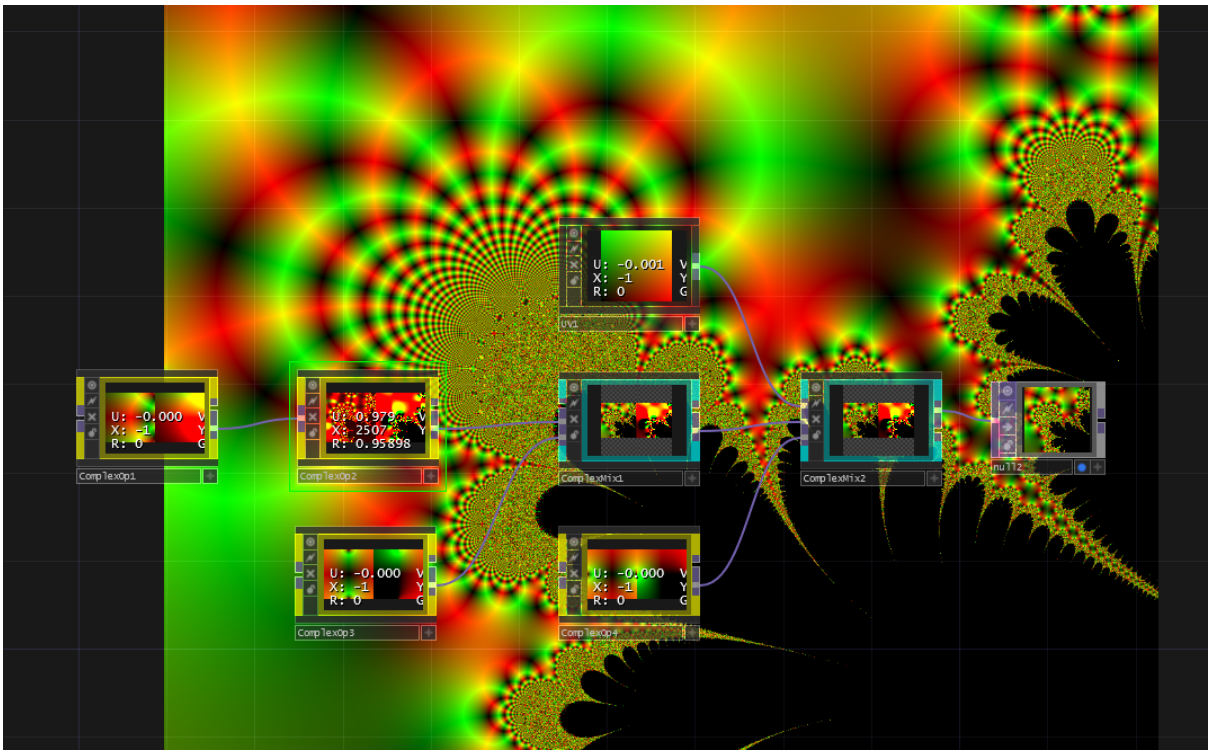
ComplexFunction.store UV toolbox

Made by [Function Store (@function.str)]

The **ComplexFunction.store** toolbox is a family of three Touchdesigner components (.tox files that you can grab & drop) meant to interact with each other.
Its purpose is to substitute writing GLSL code for UV mapping based on complex numbers to generate mathematical patterns and fractals. The components of the toolbox are modular, you can chain them as you wish!



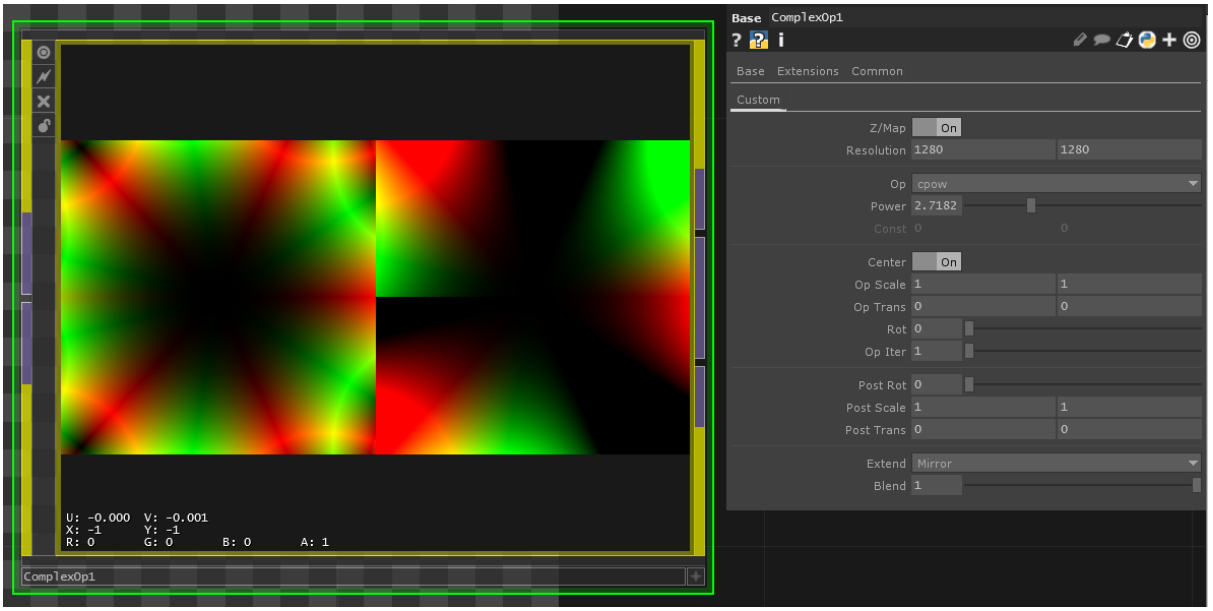
I've recorded a tutorial so you can see it in action and get some explanation if you don't want to read this doc: https://youtu.be/pM6iHx7xD7c

Also see this tutorial by Exsstas (which was the inspiration to do this) to get more understanding of the GLSL and the math behind: Crazy warping effects with a complex numbers. TouchDesigner tutorial

This document and the video tutorial is probably more complicated than actually using the toolbox, the main purpose of it is to treat it as a playground, often not knowing what you are doing.
There is an example project with multiple Containers, you can enable/disable cooking with the 'X' button on the side of the Containers.

Let's see the three components that make up the toolbox.

ComplexOp



This component lets you choose from a list of complex operators and perform operations on them. The input can have two purposes:

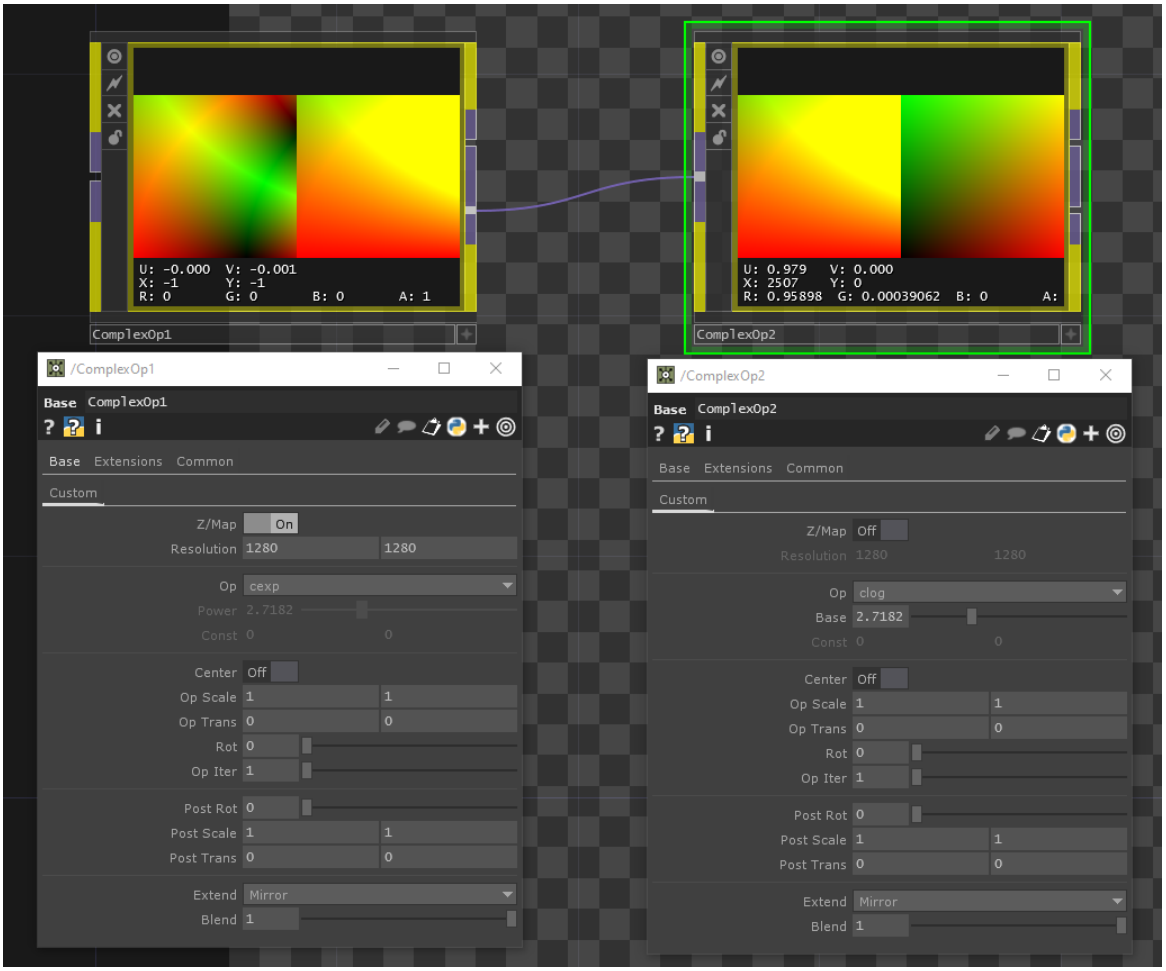
- Chaining functions **OR**
- UV mapping of an image (default UV(0-1) image)

The outputs:

- Mapped output of the input image
- UV coordinates (for chaining)

If the **‘Z/Map’** switch is off the operator will instead take the input image as the operand of the function (instead of the default 0-1 UV space), but the meaning of the outputs stay the same. You would generally want to connect the (2nd) UV coordinate output of another component from the toolkit.

For example if you connect a **‘cexp’** op with a **‘clog’** (with a base of **‘e’**) whose ‘**Z/Map**’ is in off setting you will get back the standard 0-1 UV image.

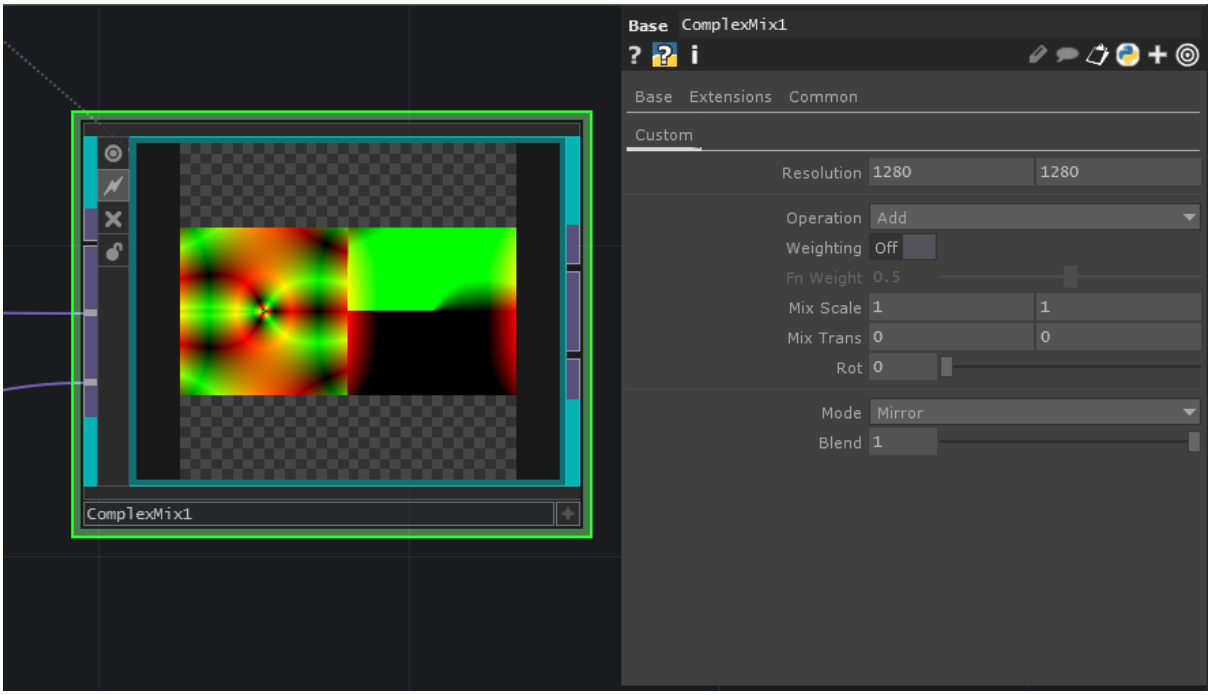


As for the other parameters: the functions **cpow, clog, cconst** have additional input fields. You can quickly center the UV space with the **Center** toggle (before transforms), then **Scale**, **Translate**, **Rotate**, and finally **Iterate** (meaning performing the same operation multiple times).

Post Rot, Post Scale, Post Trans parameters perform R/G channel transformations after the function is called.

Finally you have the standard **Extend** options (Hold/Zero/Mirror/Repeat), and **Blending** with the mapped image (sort of like a Wet/Dry).

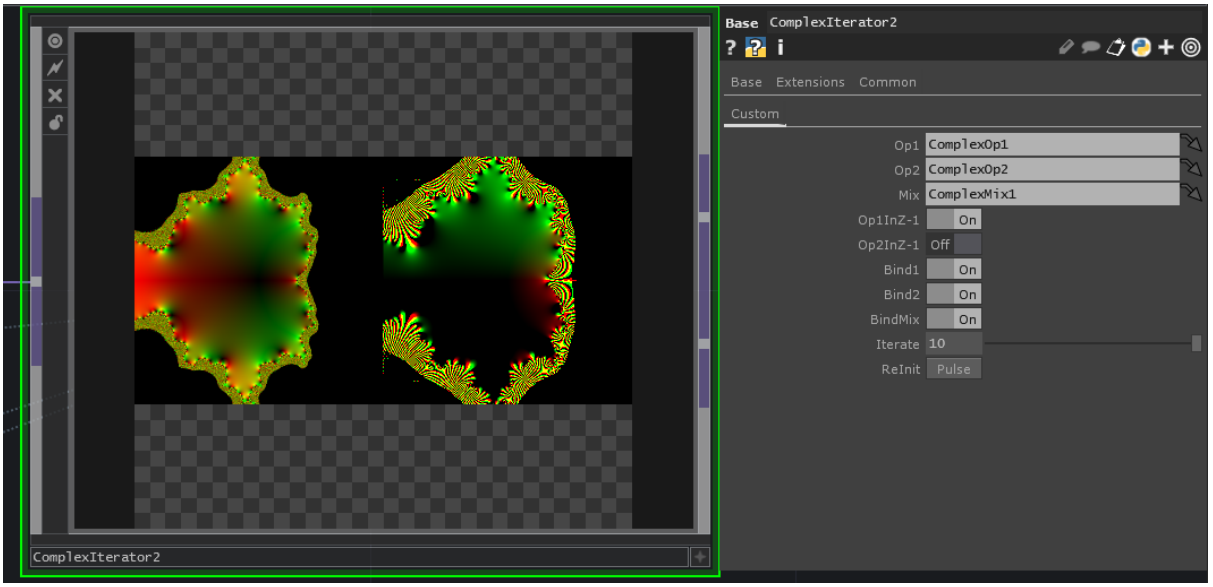
ComplexMix



ComplexMix is meant to map the image connected to its first input and take two **ComplexOps** (inputs 2 and 3), treat them as operands and execute the **Operation** which can be: **Add, Subtract, Multiply, Divide, Op1^Op2, Op2^Op1,** or selecting only one or the other (**Op1, Op2**).
There's an option to **Weight** (or blend) between the two UV inputs . You then have the ability to **Scale**, **Translate**, and **Rotate** on the R/G channels after the operation.

The outputs once again: The first output maps the first input with the UV coordinates of the second output. Also the same UV mapping **Modes** and **Blending** with the original image.

ComplexIterator



ComplexIterator is meant to save you some manual labor of chaining and repeating the same functions, essentially how you generate fractals.
You first have to give references to the components that you want to **Iterate** N times. What it does inside is create N copies of the same reference OPs and connect them.
Op1InZ-1 and **Op2InZ-1** help with connecting iterations, and this is where the **Z/Map** switch of the operator in off position comes handy. To easier explain take this Mandelbrot set:

```
Z = vUV.st
Z_orig = Z

for(int i; i < Iterate; i++) {
      Z = cpow(Z) + Z_orig;
}
```

To be able to do this with our modular approach we have to be able to chain operations (Z = cpow(Z)). What the Op*InZ-1 parameters do is put the **Z/Map** switch to off and connect the UV (second) output of the previous operation to the input of the respective operators.

With the **Bind*** toggles on the parameters of the reference OPs are referenced by the copies/iterations. (the first copy will always be bound)

The limitations come to surface here: in GLSL you can go pretty high with the iterations, but because here in this modular approach we are replicating operators plus generating and keeping all the intermediate iterations (and operands) it can become cumbersome for the CPU and GPU.

IMPORTANT:
If the path changes where the **Iterator** and its references lie or it doesn't work as expected hit **Reinit**!

License

Please tag me on IG **@function.str** if you end up using this toolbox!