# Domain Coloring of Complex Functions

Konstantin Poelke and Konrad Polthier

## 1 Introduction

Visualizing functions is an omnipresent task in many sciences and almost every day we are confronted with diagrams in newspapers and magazines showing functions of all possible flavours. Usually such functions are visualized by plotting their *function graph* inside an appropriate coordinate system, with the probably most prominent choice being the cartesian coordinate system. This allows us to get an overall impression of the function's behaviour as well as to detect certain distinctive features such as minimal or maximal points or points where the direction of curvature changes. In particular, we can "see" the dependence between input and output. However, this technique is limited to three dimensions, simply because we do not know how to draw higher-dimensional cartesian coordinate systems.

This article gives a short overview about the method of *domain coloring* for *complex functions*, which have four-dimensional function graphs and therefore cannot be visualized traditionally. We discuss several color schemes focussing on various aspects of complex functions, and provide Java-like pseudocode examples explaining the crucial ideas of the coloring algorithms to allow for easy reproduction. For a thorough treatment of domain colorings from a more mathematical point of view see [1].

## 2 What is a Function?

Let us briefly recap the definition of a function to fix terminology. A *function* $f$ consists of three parts: first, a set $D$ of input values, which is called the *domain of the function*, second, a set $Y$ called the *range of* $f$ and third, for every input value $x \in D$, a unique value $y \in Y$, called the *function value of* $f$ *at* $x$, denoted $f(x)$. The set $\Gamma(f)$ of all pairs $(a, f(a))$, $a \in D$, is a subset of the product set $D \times Y$ and called the *function graph of* $f$.

One particular type of functions that are widely used in engineering and physics are *complex functions*, i.e. functions $f : D \subseteq \mathbb{C} \to Y \subseteq \mathbb{C}$ whose domain and range are subsets of the complex numbers, and we will focus on complex functions in the following. We will identify $\mathbb{C}$ with the Euclidean plane $\mathbb{R}^2$ by the bijective assignment $x + yi \mapsto (x, y)$, justifying the term *complex plane*. Note that the function graph $\Gamma(f)$ of a complex function lives inside the product space $\mathbb{C} \times \mathbb{C} \cong \mathbb{R}^4$.

## 3 Domain Coloring

In contrast to function graph plotting inside Euclidean two or three space, the technique of *domain coloring* does not need additional *spatial dimensions* for the range but rather uses color dimensions to encode the function values. More precisely, given a complex function $f : D \to Y$, one defines a *color scheme* as a function $\texttt{col} : Y \to HSB$, which assigns to every value $y \in Y$ a color $\texttt{col}(y)$, specified in HSB color space

for instance. In practice $D$ is usually a rectangular region of $\mathbb{R}^2$. Then an initially chosen resolution divides $D$ into a discretized domain $D_h$ of pixels corresponding to small rectangles of fixed width and height. Every pixel $i$ is identified with a point $z_i$ in $D$ where the function is evaluated, for example the midpoint or a corner. Then one evaluates the *pull-back function* $f^*\texttt{col} := \texttt{col} \circ f$ at every point $z_i \in D$ and assigns the resulting color value $f^*\texttt{col}(z_i)$ to the pixel $i$, leading to a coloring of the whole domain $D_h$. This procedure is shown in Figure 1, Listing 1 gives an easy implementation.

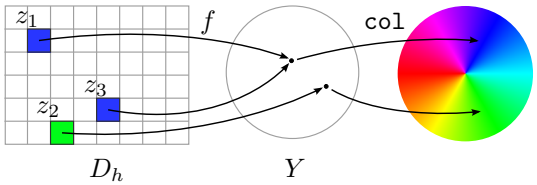


Figure 1: Every pixel $i$ in the discretized domain $D_h$ corresponds to a value $z_i \in D$ and is colored with the color $f^*\texttt{col}(z_i)$.

Listing 1: Plotting procedure. The domain $D$ is a rectangular region specified by its lower left and upper right corner, whose $(x, y)$-coordinates are given in the array `double[] dom` of size 4. The resolution parameters determine the number of pixels of the resulting image.

```
Image plot(double[] dom, Fun f, Fun col, int xRes,
  int yRes){
 // create new image object of size xRes*yRes
 Image im = new Image(xRes,yRes);
 // Increments in width and height direction
 double xInc = (dom[2]-dom[0])/xRes;
 double yInc = (dom[3]-dom[1])/yRes;
 for(y=0; y<yRes; y++){
  for(x=0; x<xRes; x++){
   // midpoint where f is evaluated
   z = (dom[0]+(x+0.5)*xInc, dom[1]+(y+0.5)*yInc);
   // assign computed color to pixel
   im.setColor(x,y,col(f(z))); }}
 return im;
}
```

We just mention for completeness that this idea can easily be transferred to surfaces $D$ embedded in $\mathbb{R}^3$ by using local coordinate charts and texture mappings and reducing the problem to the flat case in $\mathbb{R}^2$.

# 4 Color Schemes

## 4.1 Color Wheels

Every non-zero point $z = (u, v)$ in the Euclidean plane has a unique representation in terms of *polar coordinates* as $z = r \cdot \exp(i\varphi)$, if one requires the angle $\varphi$ to lie in the intervall $[-\pi, \pi)$. $\varphi$ is called the *argument of* $z$, denoted $\arg z$, and $r$ is the *modulus* or *absolute value* $|z|$. The advantage of this representation is that the argument can be directly identified with a hue value, which is usually given as an absolute angle value between 0 and 360 or as a relative value between 0 and 1. We will make steady use of the function `HSBColor(float hue, float sat, float bri)` which constructs an integer color representation from the specified hue, saturation and brightness values, each of them lying in the interval $[0, 1]$. Such functions are usually contained in the standard libraries of modern programming languages, e.g. `java.awt.Color.HSBtoRGB` in Java or `colorsys.hsv_to_rgb` in Python.

This leads to a first color scheme as shown in Figure 2, using the argument $\arg z$ as its only parameter. Listing 2 gives its implementation.

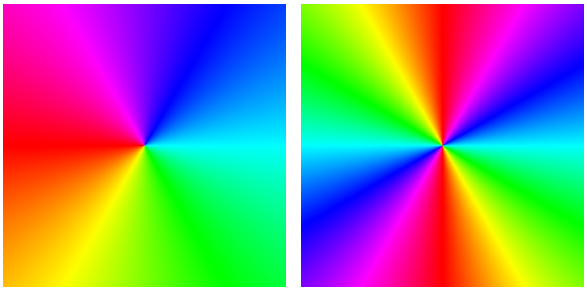

Figure 2: Hue-based color wheel and resulting domain coloring for $z \mapsto z^2$. The function duplicates the argument of every point in the plane. As a result the upper half plane is mapped to the whole plane, as every color already appears on the upper half.

Listing 2: A hue based color wheel using the argument $\arg z$ as its only input parameter.

```
int col(Point z){
 //arg only defined for non-zero value
 if (z==0) return black;
 // h between 0 and 1
 h = (arg(z)+π)/2π;
 return HSBColor(h, 1,1);
}
```

In case the argument is undefined, i.e. if $z = 0$, the function returns black. We will neglect this form of easy exception handling as well as the conversion of $\arg z$ into the relative parameter $h \in [0, 1]$ in the following.

Instead of using the hue value as a continuous parameter, a discretized version of the color wheel divides the plane into $n$ equally-sized cells, where in each cell the color is constant. A natural choice is $n = 4$, dividing the Euclidean plane into its four quadrants. This makes it easy to detect which points are mapped into which quadrant, see Figure 3 and Listing 3.

Listing 3: A discretized color wheel implementation dividing the plane into four quadrants.

```
int col(Point z){
 // array holding colors for cells
 Color[] colors = {blue,red,yellow,green};
 // compute cell index
 int i = floor(h*4) % 4;
 return colors[i];
}
```

Alternatively one can use the argument as a parameter for interpolating between two or more colors leading to color gradients wrapped around the origin. A simple gradient color scheme between white and black, discontinuous along the negative real axis, is then given by a function `blend(black,white, h)` which accepts two colors and uses the relative hue value $h$ as a weight for a linear interpolation between the colors' RGB values such that `blend(black,white,0)` returns black and `blend(black,white,1)` returns white. Figure 4 shows an application of this scheme to functions of a type particularly important in algebra and complex geometry.

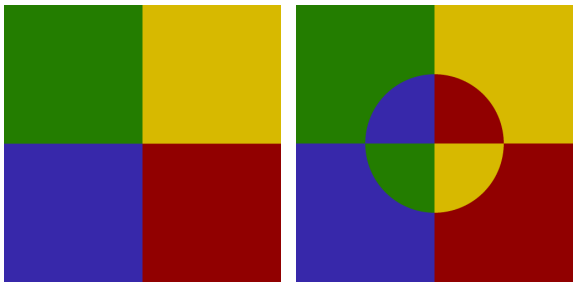

Figure 3: The discrete color wheel divides the plane into its quadrants. The image on the right shows a plot of the *Joukowski function* $(z+1/z)/2$. Inside a circle of radius 1 the plane appears mirrored along the horizontal real axis.

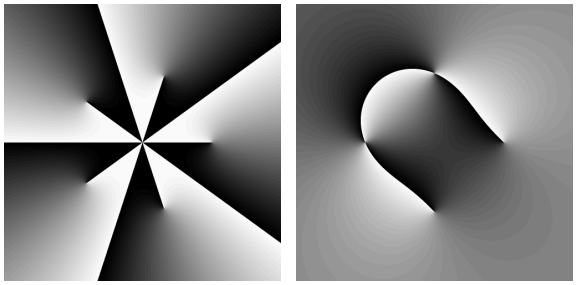

Figure 4: A simple gradient color scheme. The left picture shows the function $f(z) = z^5 - 1$, whose zeros are exactly the fifth roots of unity. The five endpoints are the zeros whereas the other endpoints all meet at infinity, where $f$ has a pole of order five. The horse-shoe-shaped image on the right displays the *meromorphic function* $f_m(z) = (z-1)(z+1)^2/((z+i)(z-i)^2)$. The two endpoints are of order one and correspond to the terms $(z-1)$ and $(z+i)$. The two points where black and white interchange are of order two, determined by the remaining terms.

Although the preceding color schemes only used the argument of a point $z = r\exp(i\varphi)$, they already gave fairly good representations for complex functions. By taking the modulus into account we can enrich these color schemes with several additional features. As an example, Listing 4

marks zeros and poles as black and white spots to make it easy to detect these distinguished points of a complex function, improving the color wheel scheme from Listing 2.

Listing 4: Color wheel with highlighted poles and zeros. Points $z$ with $|z| \leq r$ or $|z| \geq R$ are colored black and white, respectively. For $r < |z| < R$, saturation and brightness are interpolated.

```
int col(Point z){
 if(abs(z)<=r) return black;
 else if(abs(z)>=R) return white;
 else{
  sat = fadeOut(abs(z));
  bri = fadeIn(abs(z));
  return HSBColor(h,sat, bri);
 }
}
```

Note that in HSB color space white corresponds to full brightness with zero saturation whereas black corresponds to zero brightness. The methods `fadeIn` and `fadeOut` interpolate between 0 and 1 to provide smooth transitions from black to fully saturated bright colors to white, for instance using smooth bump-type functions as given in Figure 5.

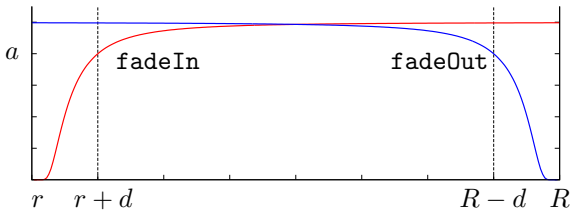


Figure 5: Classical bump functions of the type $\exp(\log a\,(l/(x-z_0))^2)$, smoothly approaching zero at $x = z_0$ and attaining a predefined value $0 < a < 1$ at $x = z_0 \pm l$, here shown for $z_0 = r$ and $z_0 = R$, respectively, and $l = d$. They provide smooth interpolation functions for brightness and saturation.

## 4.2 Grids

Complex functions that are differentiable (so called *holomorphic functions*) and have nowhere vanishing derivative are of particular geometric interest: they are *conformal*, i.e. they preserve angles. To illustrate this phenomenom, it is helpful to implement grid-like color schemes such as rectangular grids or polar grids. We explain a polar grid consisting of concentric circles as contour lines around the origin and $n$ rays starting at the origin going through the $n$-th roots of unity $\exp(2k\pi i/n)$, as shown in Figure 6.

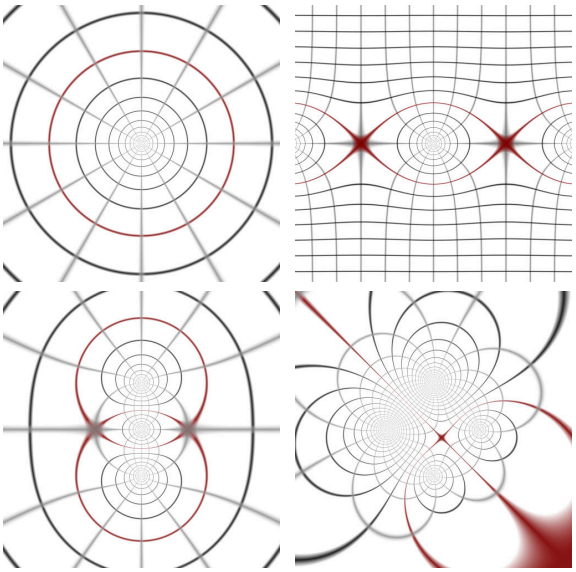

Figure 6: A polar grid color scheme with red contour line of modulus 1 (top left) applied to the complex sin (top right), the Joukowski function from Figure 3 (bottom left) and the function $f_m$ from Figure 4 (bottom right).

There are two things to do: first, compute the distance $d_\theta$ of $\arg z$ (as a relative value between 0 and 1) to the closest fraction $k/n$ for $k = 0, \ldots, n-1$. Second, compute the distance $d_c$ of the modulus $|z|$ to the closest multiple $l \cdot D$, $l \in \mathbb{N}$, where $D$ is the distance between any two consecutive concentric circles. Depending on the distances $d_\theta$ and $d_c$, return either white if the distances are too large, or a grayish color if the distances are smaller than some user defined thresholds. Since the implementation is straight-forward, we do not give code listings. Rectangular grids can be created in a

similar way, but it is more convenient to use tilings as discussed in the next section.

### 4.3 Tilings

Tilings provide another source for numerous color schemes. For instance, a rectangular grid can be easily implemented using a seamless texture showing a single cell of the grid. Texture coordinates then establish an identification of the texture with a (rectangular) region in the plane, leading to a tiling of the whole plane. Listing 5 and Figure 7 explain this procedure, applications are shown in Figures 8 and 9.

Listing 5: A color scheme tiling the plane using a rectangular texture stored in the image variable `tile` whose width and height are given as the number of pixels. The structure `rect` describes a rectangular region in Euclidean coordinates on which the tile is mapped to. Every point outside this rectangle corresponds to a unique point inside the rectangle by modulo calculation.

```
int col(Point z=x+yi){
 // point modulo tile rectangle
 double xInRect = (x-rect.LEFT) % rect.width;
 double yInRect = (y-rect.BOTTOM) % rect.height;
 // transform position into pixel coordinates
 int xPixel = (xInRect*tile.width)/rect.width;
 int yPixel = (yInRect*tile.height)/rect.height;
 // return RGB value of corresponding pixel
 return tile.getRGB(xPixel,yPixel);
}
```

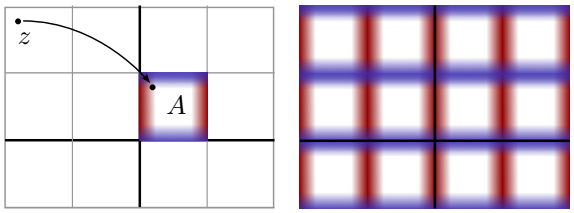


Figure 7: A tile is identified with a rectangular region $A$ inside the domain, giving rise to a lattice whose cells are copies of $A$ (left). Every point $z$ corresponds to a unique point inside $A$ by taking its coordinates modulo the lattice, yielding a regular tiling of the domain (right).

### 4.4 Combining Color Schemes

A simple way to create new color schemes is merging already existing ones into a single scheme. For instance, it makes sense to blend the polar grid in Figure 6 onto a color wheel background to combine the information about the argument given by the color wheel with the information about the induced plane deformation given by the grid. This can be easily done by taking alpha values into account. The polar grid would return a fully transparent color, i.e. a color having alpha value 0, whenever a point neither lies on a radial line nor on a concentric circle, and a dark color with increasing alpha value if the point lies close to a grid line,

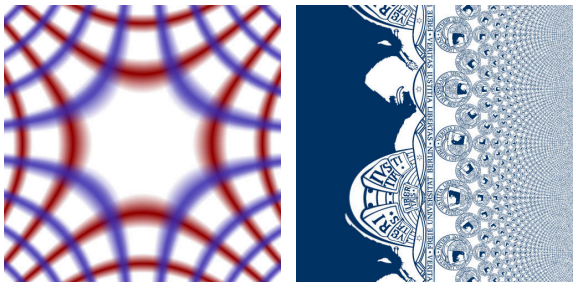

Figure 8: The function $z \mapsto z^2$ using a rectangular grid. All blue lines are mapped to horizontal lines, whereas the red lines are mapped to vertical lines (left). The complex exponential function $z \mapsto \exp(z)$ deforming the originally circular FU emblem (right). The emblem is unfolded in the negative real half plane whereas the number of emblems is exponentially growing in positive real direction. Note that this pattern is repeating in imaginary direction due to the periodicity of the complex exponential function.

Figure 9: Again the function $z \mapsto z^2$ (left) and the function $f_m$ (right), this time using a hand-drawn pencil sketch of a twig.

until the color is fully opaque for points lying directly on the grid. So assume we are given a list of $n$ color functions $c_1, \ldots, c_n$ where we want to use $c_1$ as the background. Then the resulting combined color scheme is the function

$$\alpha(c_n(z), \alpha(\ldots, \alpha(c_3(z), \alpha(c_2(z), c_1(z)))))$$

where $\alpha$ takes two colors as arguments and blends the first color onto the second color according to their alpha values, cf. [2]. The following figures give some ideas what such blended schemes may look like.

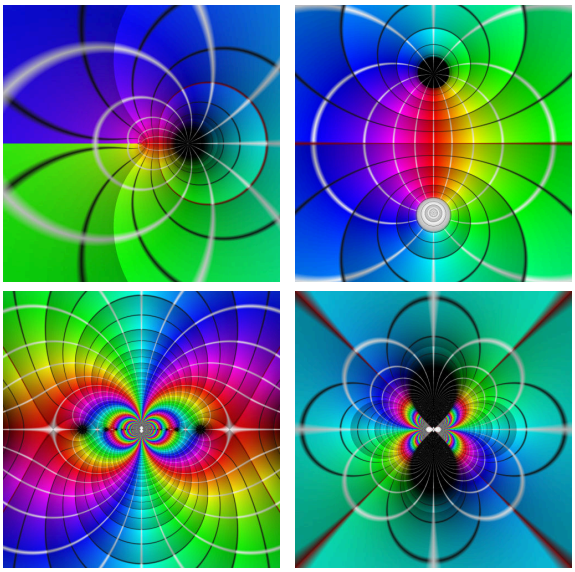

Figure 10: The "spider" (top left) is the complex logarithm $\log(z)$, the *Cayley map* $z \mapsto (z-i)/(z+i)$ (top right) has many applications in complex geometry. The functions $z \mapsto \cos(1/z)$ (bottom left) and $z \mapsto \exp(1/z^2)$ have *essential singularities* at zero, recognizable by the ever-repeating rainbows shrinking to a point. All images are rendered using a continuous color wheel onto which a polar grid and semi-transparent rings indicating the direction of growth of the modulus are blended, following an idea from [3].

Of course this list of suggestions is by far not exhaustive and provides just a very brief introduction. However it should give you enough basics to implement your own color schemes and experiment with the parameters.

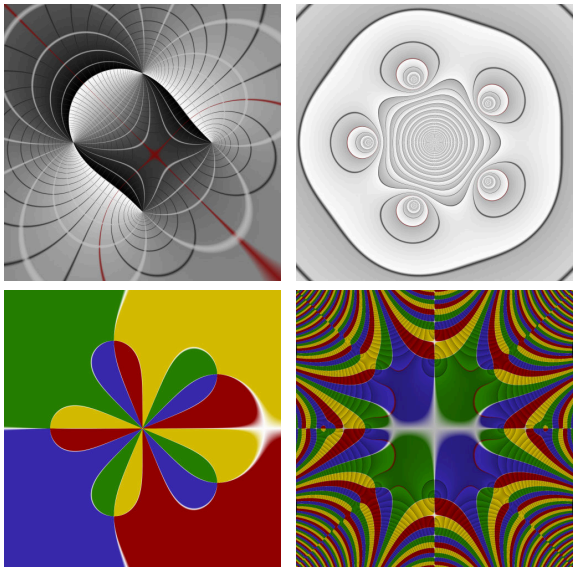

Figure 11: The function $f_m$ using a gradient blended with a polar grid. The white rays connect zeros with poles (top left). Contour lines enclose the five zeros and the pole of order 4 at the origin of the function $z \mapsto z + 1/z^4$, using a solid background onto which the concentric circles of the polar grid and semi-transparent rings are blended (top right). The same function is shown using a discrete color wheel with additional polar grid lines (bottom left), also used for the rather artificial function $z \mapsto \cos(z)/(\sin(z^4 - 1))$ (bottom right).

## References


[1] K. Poelke and K. Polthier, "Lifted domain coloring," *Computer Graphics Forum*, vol. 28, no. 3, pp. 735–742, 2009.

[2] T. Portner and T. Duff, "Compositing Digital Images," *Computer Graphics*, vol. 18, no. 3, pp. 253–259, 1984.

[3] H. Lundmark, "Visualizing complex analytic functions using domain coloring." `//www.mai.liu.se/~halun/complex/domain_coloring-unicode.html`, 2004.