©2014 Bogdan Nikolić – All rights reserved.

**Inspiration**

Image editing applications, of which Adobe Photoshop is the most famous, usually support drawing color gradients which are: (1) predefined, and (2) linear in nature. It means that, when a user wishes to draw a color gradient, he first needs to define a gradient, and only than he is able to apply gradient definition to the drawing surface, and thus make a modification of the image. Defining gradient is actually an action of inserting and arranging anchor points (with color values assigned to each of them) positioned in linear manner along a line (which represents a user interface). User thus arranges the colors in an array, and defines a relative distance between changing colors. On the other hand, when a user applies a gradient to the drawing surface, he defines the start and the end point of the gradient, so the software can render the image by interpolating colors according to parameters of gradient definition (anchor color values, color arrangement and relative distances).

Although image editing applications usually allow user to draw three different types of gradients (linear, radial and angled), the fundamental logic is the same. All three types of gradients are linear in nature because colors take turn one after another in the manner they were arranged in an array in gradient definition.



**Pictures 1, 2, and 3** – example of linear, radial and angled gradient

Vector-based graphics editing applications, of which Adobe Illustrator is the most famous, apart from classical linear predefined gradients, sometimes support so-called mesh gradients. Mesh gradients were introduced in vector applications as a way to create scalable no-loss representations of raster images. Algorithms behind them use mesh surfaces to model colors of the particular 2D area. Although they allow high flexibility, mesh gradients are rather unpopular in graphic design community. They require large amount of time for graphic designer to draw them (usually as an overlay of an actual photo), and they lacks natural continuity and flow which gradients are expected to have. Thus, mesh gradients are used rarely (almost always for simulating photography) and are not suitable for evoking environmental effects.



**Picture 4** – example of mesh gradient

Inspiration for creating this library came up to me once as I was working in Photoshop. Photoshop dialog for color selection incorporates square image (whose size is 256 per 256 pixels) where are in tabular manner represented all colors whose one component (red, green or blue) is constant. For example, if we set red value to 200, the mentioned image will automatically represent all colors which have red value of 200. It does so by setting every pixel’s red component to 200, and setting two other color components relative to position of a pixel in the image. That very image was colored in beautiful non-linear gradient in manner I have never seen on some work of graphic design. So I asked myself how I can programmatically create these kinds of gradients.



**Picture 5** – all colors where red component is 200

**Problem**

Let us suppose we are given an image (let it be 24bpp bitmap, but it does not need to be so) as a matrix of pixels. Let us also suppose that we have arbitrary selected pixels on the bitmap, and that color value for each of these pixels is defined. We may denote these pixels as and we can write down them in the following manner:

(Values and are arbitrary integers, and , and , represented by 8-bit numbers in 24bpp bitmap, are integers in range [0, 255], but it is not particularly important for us at this point. We can imagine for a while that we are working with real numbers.)

Let these pixels be our *anchor pixels*, which means that when we generate gradient, the colors of these pixels will stay unchanged (or more precisely: color values of pixels at positions will be). To draw a gradient we need to specify color of each pixel in the bitmap, which means that we need do define a rule such that

where represents pixel position space, and represents pixel color space. The rule will also map to , so we can write:

As , and are mutually independent we can break down function to 3 separate, but identical functions:

Our main concern will be and functions, which we will jointly refer to only as. Strictly formal, our problem can be summarized as classical interpolation problem for two-variable function and can be expressed in the following form:

Let we are given. Find function so that:

(1)

**Unique Solution**

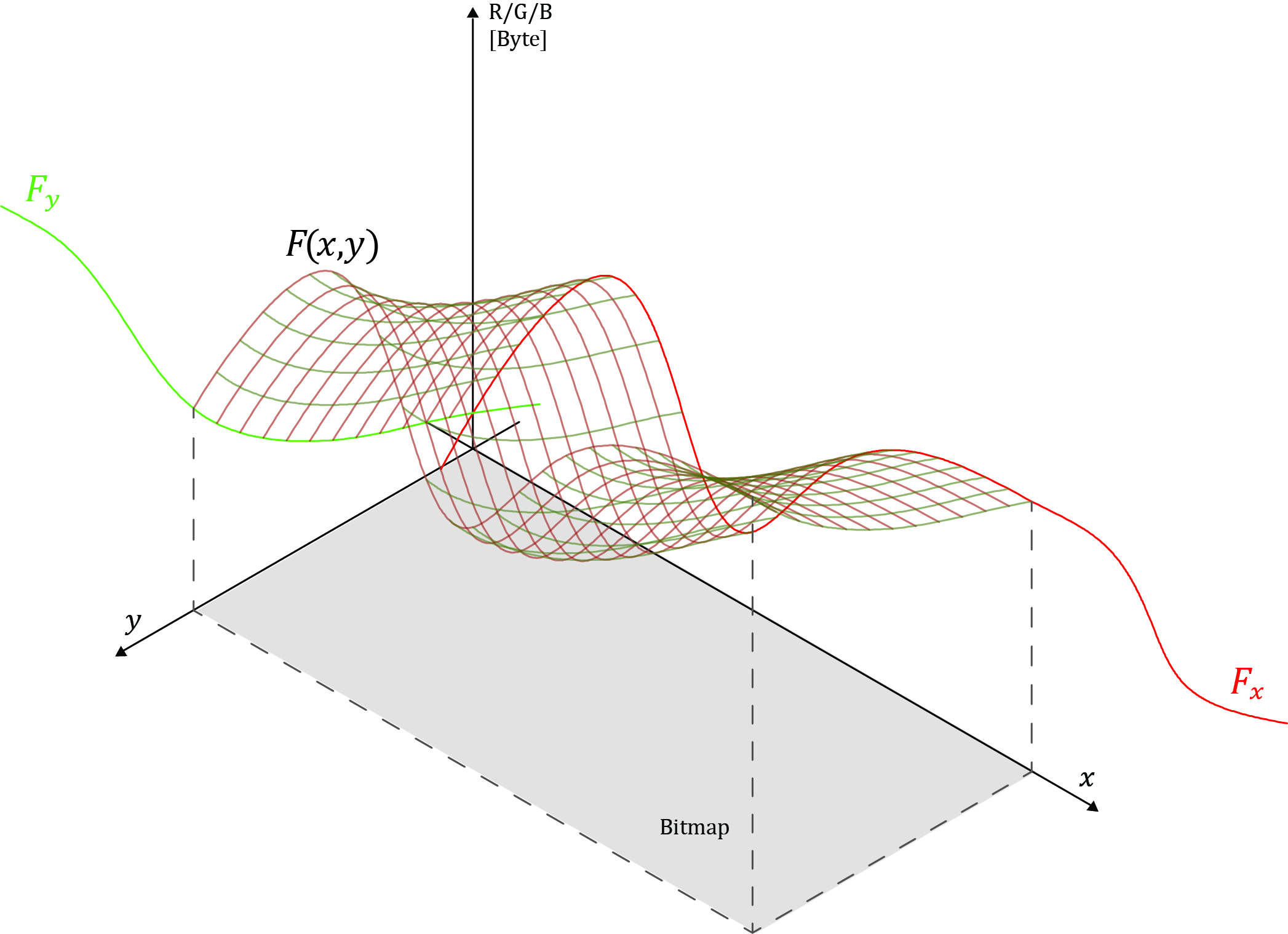
First thing we need to do is to determine in what class of functions we will look for .

It turned out that it is useful if has following form:

where , and is constant. Subsequently:

There are at least two good reasons why this form of function is appropriate for our problem.

First of all, it is relatively easy to geometrically visualize it as a surface because it is a simple sum of two polynomials. If we draw graph on a plane of 3D coordinate system, and graph on a plane, we can get the appropriate surface simply translating graph along graph (and vice versa). This allows us to relatively easily get an idea of how the surface looks and, subsequently, to get an impression how the gradient may look like. It may be very useful during coding stage of application development. Function visualized as a surface is shown in Picture 6.



**Picture 6**

Second and more important reason why this form of function is suitable for our problem is that it can be easily recurrently calculated, which, as it turned out, boosted algorithm performance. This will be explained in detail in section 5 “Performance Optimization”.

The system (1) will now be:

(2)

The system (2) can be presented in matrix form:

(4)

Let’s use , and according to following:

If all columns and rows are linearly independent, system will be uniquely determined if , which means:

(3)

It follows that for any particular number of anchor pixel n we can get n different solutions depending on our selection of and values.

With our new notation, equation (4) will be:

(5)

Equation (5) has a solution if and only if and that solution is:

So our function will now be:

where is a row vector defined such:

, and will now be:

where , and are column vectors defined such:

Let us define and matrices in the following manner:

Now, function ) will be:

*Remark.* It would be useful to determine in what conditions because our application may throw overflow exceptions when encounter with such conditions.

When (or ), matrix is degenerated to Vandermonde matrix which is used for classical, one-variable interpolation. It can be easily proved that determinant of Vandermonde matrix is product of all differences of variables of interpolating polynomial, namely:

However, I could not find in literature equivalent expression for value of determinant of matrix V, which is, in a sense, a “two-variable Vandermonde matrix”, and it seems that “the equivalent expression”, which would allow us to easily determine conditions when , does not exist. Nevertheless, we can determine at least some trivial conditions when , and these conditions can be presented in following disjunction:

Practically, it means that we cannot have all anchor pixels with same x or y coordinates, nor have two anchor pixels with both coordinates same.

When these conditions are met, determinant is 0, but when they are not met, we know nothing about value of determinant.

**Parametric Solution**

If the system (2) will not have unique solution, regardless of linear dependence/independence of columns/rows of matrix. If for example, the system (2) will have whole class of solutions and all solutions will be functionally dependent on at least one parameter, because will be at least for 1 lower than number of unknowns. Let us suppose that all columns and rows of matrix are linearly independent. In that case, all solutions of (2) will be functionally dependent on exactly one parameter. Theoretically, it means that we can parameterize quotients of function , and subsequently, get whole class of functions for which (1) will be true. Practically, it means that we can generate whole family of gradients, so that each member of family will have exact same anchor pixels as any other. We can mathematically paraphrase previous paragraph in following manner:

Function will now be:

Let us define and find as a function of . (9) will now be:

which is equivalent to:

Equation corresponding to (4) will now be:

(10)

Let use , and according to following:

so equation (10) will be:

Equation (11) has a solution if and only if and that solution is:

So our function will now be:

where is row vector defined such:

and will now be:

where and are column vectors defined such:

Let us define , and matrices in the following manner:

Now, function will be:

**Performance optimization**

In this section I will briefly describe how function can be recurrently calculated in order to achieve better computational performance. I will focus on the efficiency of algorithm itself, not on the manner how algorithm is implemented in the code, for example how memory is managed, etc.

There are two important facts regarding function . The first fact is that it is continuous differentiable function. The second fact is that arguments of function – as we implement it for generation of color gradients – are only integers, since pixels can only have integer coordinates. As we need to know values of only for integer arguments, and since is differentiable, it makes sense to try to calculate value of recurrently.

Let us use according to following definition:

where is a real order of members, is a -th member in , and . Now, let us try to find recurrent relationship between and .

To summarize:

As we have expected, the difference of two polynomials of power can be expressed as a polynomial of power .

We can utilize this recurrent relationship in computational optimization of function in following manner (according to (1.1) or (1.2)):

The important fact about our recurrent relationship is that every order has its corresponding orders and . Once we compute values of and , we can store them in memory and use multiple times.

Now we see more clearly why it is useful to choose two-variable polynomial functions that do not have mixed members.

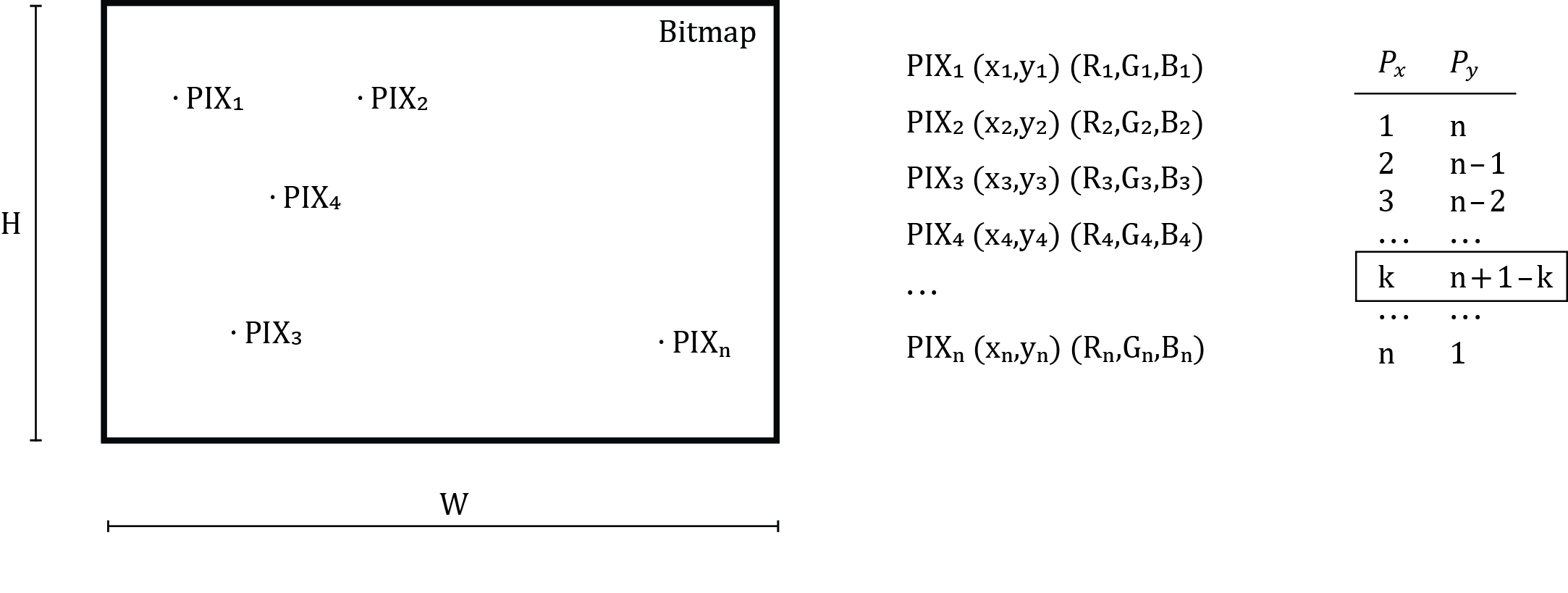
In the next chapter I will present draft of the algorithm where I will discuss and interpret mathematical findings we have come along in previous three chapters.

**Algorithm draft**

Let’s draft algorithm for generating non-parametric gradients.

First of all, let’s specify initial input we need: bitmap width and height, collection of anchor pixels, and and values whose sum must be for 1 lower than number of anchor pixels.

As we have mentioned, for any particular number of anchor pixels we can get different solutions depending on our selection of and values, because . It is useful to specify at least three anchor pixels, because if either or is 0, the gradient will be degenerated to linear gradient. Following picture shows schematic representation of our inputs.



**Picture 7 -** schematic representation of input.

The algorithm steps are roughly as follows (for any color channel):

1. Generate matrices and depending on anchor pixel input.
2. Check if has inverse. If so, get . Otherwise, assign to trivial value.
3. Using as an input, get and , recurrent quotients, using following expression:
4. Using and recurrent quotients, calculate orders and :
5. Get color for first pixel
6. Move through bitmap and calculate other pixels in following manner:

*Remark.* Value of function is not limited to interval [0,255], so it is necessary to adjust its value in order it to be in byte range. The simplest solution is to assign 0 to all pixels where , and 255 for all pixels where .

1. Return gradient bitmap.

**Limitations and Conclusion**

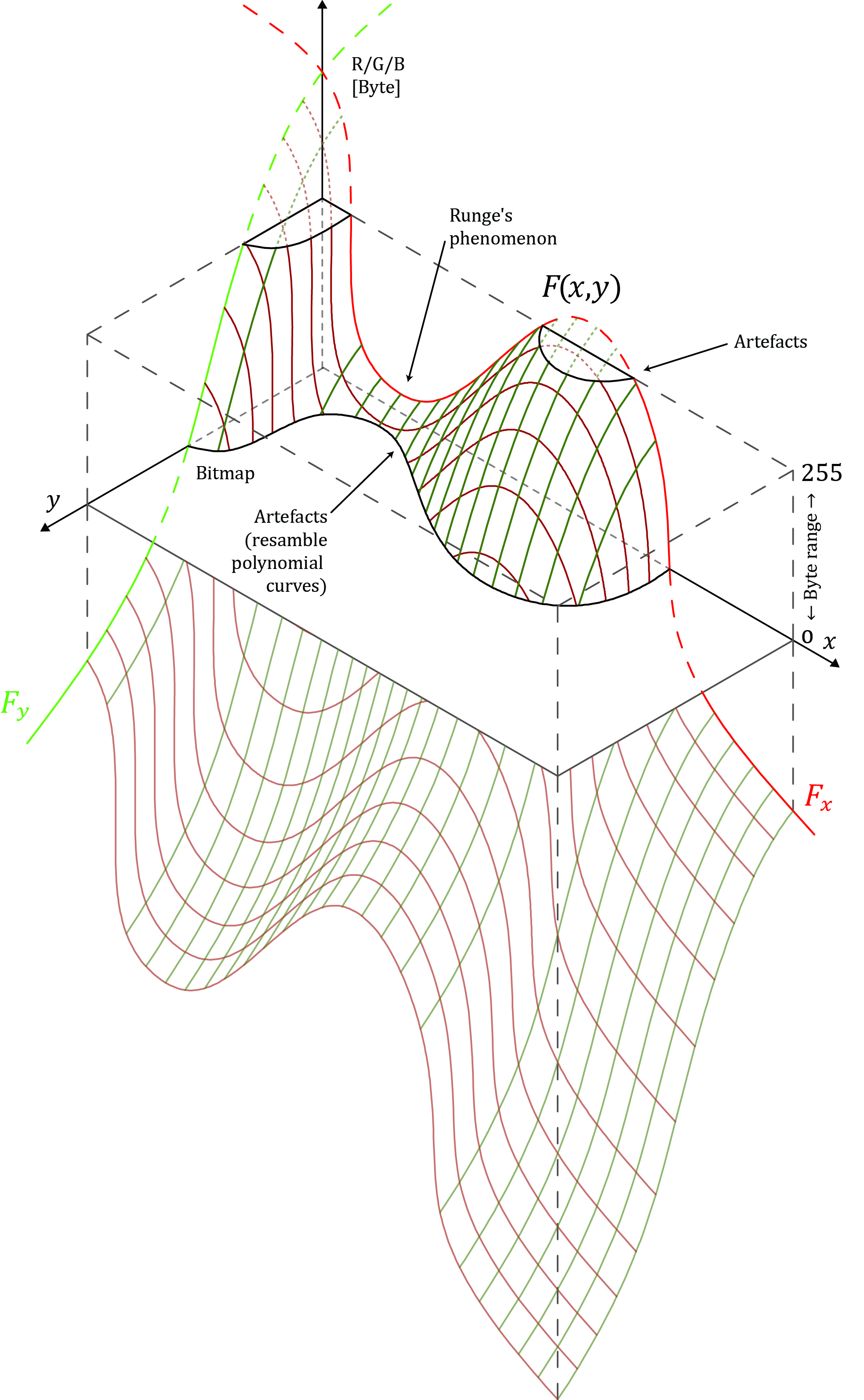
The algorithm presented in the previous chapter has proven to be fast, reliable and flexible. It indeed generates color gradients that have the same visual impact as the image from Photoshop color selection dialog which was my inspiration. The algorithm is much easier to use than mesh gradient engine available in Illustrator and gives more dynamic and vibrant results. However, it has two major drawbacks. Picture 10 shows typical situation.

The first drawback is related to the fact that algorithm is using polynomial interpolation which is prone to oscillations when using polynomials of high degree, which is known as Runge's phenomenon. This results in inaccuracy.

The second drawback is related to the fact that function does not return value in byte range, so it is necessary to reconstruct colors in areas where value cannot be used for color evaluation ( is lower than zero or higher than 255). The graph surface of function may be very steep so it enters and leaves byte range very quickly. The majority of pixels in the image must be reconstructed (assigned with color values of either 0 or 255). The results are strange artefacts which resemble polynomial curves. If artefacts coincide in all three color channels, gradient may be ruined as majority of pixels might be either black or white.

The possible solution (which is not integrated in library) is to smooth transitions from pixels whose value is in byte range and those whose value is not. However, the algorithm accuracy will be compromised.

In any case, the artifacts can be minimized or even removed by moving anchor pixels. It seems that for any selection of and , particular anchor pixel layout exists when practically there are no artefacts. However, the usability of the algorithm is limited as anchor pixels cannot be moved completely freely. Mesh gradients do not have this issue.



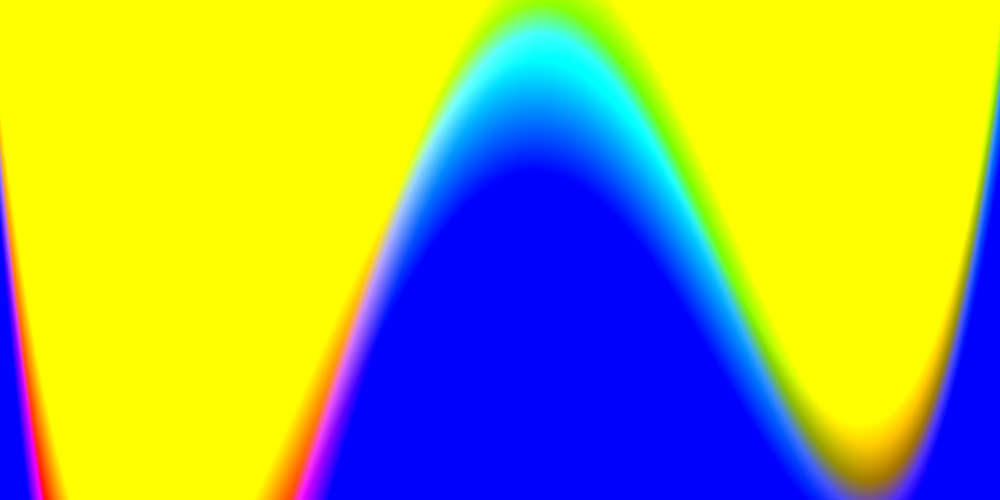
**Picture 10**



Low-level artefacts example



Moderate-level artefacts example



High-level artifacts example