Algorithms and Stuff

Fastest Gaussian Blur (in linear time)

I needed really fast Gaussian blur for one of my projects. After hours of struggling and browsing the internet, I finally found the best solution.

Beginning

My solution is based on Fast image convolutions by Wojciech Jarosz. Presented ideas are very simple and I don't know who is the original author. I am going to describe it a little better and add some mathematics. To get motivated, take a glance at the results. I have implemented this code into Photopea under Filter - Blur - Gaussian Blur.

Definition

The convolution of two 2D functions f and g is defined as the volume of product of f and "shifted" g. The second function g is sometimes called "weight", since it determines, how much of f will get into the result

The Gaussian blur of a 2D function can be defined as a convolution of that function with 2D Gaussian function. Our gaussian function has an integral 1 (volume under surface) and is uniquely defined by one parameter σ called standard deviation. We will also call it "radius" in the text below.

In our discrete finite case, we represent our 2D functions as matrices of values. We compute the volume (integral) as a sum. Gaussian function has near to zero values behind some radius, so we will use only the values $-r \le x \le r, -r \le y \le r$. This "useful" part of weight is also called the kernel. The value of convolution at [i, j] is the weighted average, i. e. sum of function values around [i, j] multiplied by weight.

Algorithm 1

For a general discrete convolution of f and weight function w, we can compute the result b as:

$$b[i,j] = \sum_{y=i-r}^{i+r} \sum_{x=j-r}^{j+r} f[y,x] * w[y,x]$$

For gaussian weight, we can compute only weights around [i, j] (area of $4 \cdot r^2$). When our matrix has n values, the time complexity is $O(n \cdot r^2)$. For large radii, e. g. $r = 10$, we have to do $n * 400$ operations, which correspond to 400 loops over the whole matrix and that is ugly.

```
// source channel, target channel, width, height, radius
function gaussBlur_1 (scl, tcl, w, h, r) {
    var rs = Math.ceil(r * 2.57);     // significant radius
    for(var i=0; i<h; i++)
        for(var j=0; j<w; j++) {
            var val = 0, wsum = 0;
            for(var iy = i-rs; iy<i+rs+1; iy++)
                for(var ix = j-rs; ix<j+rs+1; ix++) {
                    var x = Math.min(w-1, Math.max(0, ix));
                    var y = Math.min(h-1, Math.max(0, iy));
                    var dsq = (ix-j)*(ix-j)+(iy-i)*(iy-i);
                    var wght = Math.exp( -dsq / (2*r*r) ) / (Math.PI*2*r*r);
                    val += scl[y*w+x] * wght;  wsum += wght;
                }
            tcl[i*w+j] = Math.round(val/wsum);
        }
}
```

Algorithm 2

Let's introduce the box blur. It is the convolution of function f and weight w, but weight is constant and lies within a square (box). The nice feature of box blur is, that when you have some weight function having the same variance, it converges to gaussian blur after several passes.

In this algorithm, we will simulate the gaussian blur with 3 passes of box blur. Let's denote the half of size of square as br ("box radius"). The constant value of weight is $1/(2 \cdot br)^2$ (so the sum over the whole weight is 1). We can define box blur as:

$$bb[i,j] = \sum_{y=i-br}^{i+br} \sum_{x=j-br}^{j+br} f[y,x]/(2 \cdot br)^2$$

We have to convert the standard deviation of gaussian blur r into dimensions of boxes for box blur. I am not very good at calculus, but fortunatelly I have found this website and used their implementation.

```
function boxesForGauss(sigma, n)  // standard deviation, number of boxes
{
    var wIdeal = Math.sqrt((12*sigma*sigma/n)+1);  // Ideal averaging filter width
    var wl = Math.floor(wIdeal);  if(wl%2==0) wl--;
    var wu = wl+2;

    var mIdeal = (12*sigma*sigma - n*wl*wl - 4*n*wl - 3*n)/(-4*wl - 4);
    var m = Math.round(mIdeal);
    // var sigmaActual = Math.sqrt( (m*wl*wl + (n-m)*wu*wu - n)/12 );

    var sizes = [];  for(var i=0; i<n; i++) sizes.push(i<m?wl:wu);
    return sizes;
}
```

Our algorithm has still the same complexity $O(n \cdot r^2)$, but it has two advantages: first, the area is much smaller (br is almost equal to σ, while significant radius for gaussian is much larger). The second advantage is, that the weight is constant. Even though we have to do it 3 times, it performs faster.

```
function gaussBlur_2 (scl, tcl, w, h, r) {
    var bxs = boxesForGauss(r, 3);
    boxBlur_2 (scl, tcl, w, h, (bxs[0]-1)/2);
    boxBlur_2 (tcl, scl, w, h, (bxs[1]-1)/2);
    boxBlur_2 (scl, tcl, w, h, (bxs[2]-1)/2);
}
function boxBlur_2 (scl, tcl, w, h, r) {
    for(var i=0; i<h; i++)
        for(var j=0; j<w; j++) {
            var val = 0;
            for(var iy=i-r; iy<i+r+1; iy++)
                for(var ix=j-r; ix<j+r+1; ix++) {
                    var x = Math.min(w-1, Math.max(0, ix));
                    var y = Math.min(h-1, Math.max(0, iy));
                    val += scl[y*w+x];
                }
            tcl[i*w+j] = val/((r+r+1)*(r+r+1));
        }
}
```

Algorithm 3

We have already simplified gaussian blur into 3 passes of box blur. Let's do a little experiment. Let's define a horizontal blur and total blur:

$$b_h[i,j] = \sum_{x=j-br}^{j+br} f[i,x]/(2\cdot br)$$

$$b_t[i,j] = \sum_{y=j-br}^{j+br} b_h[y,j]/(2\cdot br)$$

Those two functions are "looping" in a line, producing "one-dimensional blur". Let's see, what total blur corresponds to:

$$b_t[i,j] = \sum_{y=i-br}^{i+br} b_h[y,j]/(2\cdot br) = \sum_{y=j-br}^{j+br} \left(\sum_{x=j-br}^{j+br} f[y,x]/(2\cdot br)\right)/(2\cdot br)$$

$$= \sum_{y=i-br}^{i+br} \sum_{x=j-br}^{j+br} f[y,x]/(2\cdot br)^2$$

We just discovered, that our total blur is box blur! Both total blur and horizontal blur have a complexity $O(n\cdot r)$, so the whole box blur has $O(n\cdot r)$.

```
function gaussBlur_3 (scl, tcl, w, h, r) {
    var bxs = boxesForGauss(r, 3);
    boxBlur_3 (scl, tcl, w, h, (bxs[0]-1)/2);
    boxBlur_3 (tcl, scl, w, h, (bxs[1]-1)/2);
    boxBlur_3 (scl, tcl, w, h, (bxs[2]-1)/2);
}
function boxBlur_3 (scl, tcl, w, h, r) {
    for(var i=0; i<scl.length; i++) tcl[i] = scl[i];
    boxBlurH_3(tcl, scl, w, h, r);
    boxBlurT_3(scl, tcl, w, h, r);
}
function boxBlurH_3 (scl, tcl, w, h, r) {
    for(var i=0; i<h; i++)
        for(var j=0; j<w; j++)  {
            var val = 0;
            for(var ix=j-r; ix<j+r+1; ix++) {
                var x = Math.min(w-1, Math.max(0, ix));
                val += scl[i*w+x];
            }
            tcl[i*w+j] = val/(r+r+1);
        }
}
function boxBlurT_3 (scl, tcl, w, h, r) {
    for(var i=0; i<h; i++)
        for(var j=0; j<w; j++) {
            var val = 0;
            for(var iy=i-r; iy<i+r+1; iy++) {
                var y = Math.min(h-1, Math.max(0, iy));
                val += scl[y*w+j];
            }
            tcl[i*w+j] = val/(r+r+1);
        }
}
```

Algorithm 4

One-dimensional blur can be computed even faster. E. g. we want to compute horizontal blur. We compute $b_h[i,j], b_h[i,j+1], b_h[i,j+2], \ldots$. But the neighboring values $b_h[i,j]$ and $b_h[i,j+1]$ are almost the same. The only difference is in one left-most value and one right-most value. So $b_h[i,j+1] = b_h[i,j] + f[i,j+r+1] - f[i,j-r]$.

In our new algorithm, we will compute the one-dimensional blur by creating the accumulator. First, we put the value of left-most cell into it. Then we will compute next values just by editing the previous value in constant time. This 1D blur has the complexity $O(n)$ (independent on r). But it is performed twice to get box blur, which is performed 3 times to get gaussian blur. So the complexity of this gaussian blur is 6 * $O(n)$.

```
function gaussBlur_4 (scl, tcl, w, h, r) {
    var bxs = boxesForGauss(r, 3);
    boxBlur_4 (scl, tcl, w, h, (bxs[0]-1)/2);
    boxBlur_4 (tcl, scl, w, h, (bxs[1]-1)/2);
    boxBlur_4 (scl, tcl, w, h, (bxs[2]-1)/2);
}
function boxBlur_4 (scl, tcl, w, h, r) {
    for(var i=0; i<scl.length; i++) tcl[i] = scl[i];
    boxBlurH_4(tcl, scl, w, h, r);
    boxBlurT_4(scl, tcl, w, h, r);
}
function boxBlurH_4 (scl, tcl, w, h, r) {
    var iarr = 1 / (r+r+1);
    for(var i=0; i<h; i++) {
        var ti = i*w, li = ti, ri = ti+r;
        var fv = scl[ti], lv = scl[ti+w-1], val = (r+1)*fv;
        for(var j=0; j<r; j++) val += scl[ti+j];
        for(var j=0  ; j<=r ; j++) { val += scl[ri++] - fv       ;   tcl[ti++] = Math.round(val*iarr); }
        for(var j=r+1; j<w-r; j++) { val += scl[ri++] - scl[li++];   tcl[ti++] = Math.round(val*iarr); }
        for(var j=w-r; j<w  ; j++) { val += lv        - scl[li++];   tcl[ti++] = Math.round(val*iarr); }
    }
}
function boxBlurT_4 (scl, tcl, w, h, r) {
    var iarr = 1 / (r+r+1);
    for(var i=0; i<w; i++) {
        var ti = i, li = ti, ri = ti+r*w;
        var fv = scl[ti], lv = scl[ti+w*(h-1)], val = (r+1)*fv;
        for(var j=0; j<r; j++) val += scl[ti+j*w];
        for(var j=0  ; j<=r ; j++) { val += scl[ri] - fv     ;  tcl[ti] = Math.round(val*iarr);  ri+=w; ti+=w; }
        for(var j=r+1; j<h-r; j++) { val += scl[ri] - scl[li];  tcl[ti] = Math.round(val*iarr);  li+=w; ri+=w; ti+=w; }
        for(var j=h-r; j<h  ; j++) { val += lv      - scl[li];  tcl[ti] = Math.round(val*iarr);  li+=w; ti+=w; }
    }
}
```

Results

I was testing all 4 algorithms on an image below (4 channels, 800x200 pixels). Here are the results:

Algorithm	Time, r=5	Time, r=10	Time complexity
Algorithm 1	7 077 ms	27 021 ms	$O(n \cdot r^2)$
Algorithm 1 (pre-computed weight)	2 452 ms	8 990 ms	$O(n \cdot r^2)$
Algorithm 2	586 ms	2 437 ms	$O(n \cdot r^2)$
Algorithm 3	230 ms	435 ms	$O(n \cdot r)$
Algorithm 4	32 ms	34 ms	$O(n)$

Note, that Alg 1 is computing the true Gaussian blur using gaussian kernel, while Alg 2,3,4 are only approximating it with 3 passes of box blur. The difference between Alg 2,3,4 is in complexity of computing box blur, their outputs are the same.