

---

# Scale Space Edge Detection

---

Nick Draper, Jonathan Hayase  
Seminar in Differential Geometry  
Harvey Mudd College

## Abstract

Scale space representation is the idea that a two dimensional image can be represented by a collection of smoothed images. This paper documents how using such a representation can be useful for detecting edges in an image. The scale space allows for a classification of how strong different edges are in the image from very fine to coarse ones.

## 1 Edge Detection Background

Detecting edges in images has long been a problem with many unique solutions to approach it. Generally speaking, the majority of algorithms are usually checking the image for some of the following features:

- large discontinuities in luminance values
- discontinuities in different object orientations
- large discontinuities in the intensity gradient

Some of the common methods for edge detection include the Sobel, Canny, Prewitt, Roberts, and Fuzzy Logic algorithms. However, these methods do not yield a lot of information with regards to the strength of the edges detected. The majority of these algorithms will usually convolve the image with a static matrix to calculate edges and does not adapt enough to the image.

This is why for our edge detection method, we will be using the scale space approach. The benefit of using a scale space approach for edge detection, is we have the ability to classify the strength of the edges in the image. This allows for a range of edges from very large immediate changes in intensity to very gradual.

## 2 Scale Space and Its Derivatives

To understand how exactly we detect images in the scale space, we must first define what the scale space is. If we have a continuous function of multiple variables such as  $f(x, y)$ , then we define the scale space representation of such a function as

$$L(x; t) = g(x, y; t) * f(x, y) \tag{1}$$

Here  $t$  represents the scale parameter, and can be thought of how much smoothing is applied to the function. The function  $g$  is the Gaussian kernel given by

$$g(x, y; t) = \frac{1}{2\pi t} e^{-(x^2 + y^2)/(2t)} \tag{2}$$

With the scale space representation defined, we can now take derivatives of it as it is a continuous well-defined function. Spatial derivatives are relatively simple being defined as the following

$$L_{x^\alpha y^\beta}(\cdot; t) = \partial_{x^\alpha y^\beta} L(\cdot; t) = g_{x^\alpha y^\beta}(\cdot; t) * f(\cdot) \quad (3)$$

However, when taking the partial derivative with respect to the scale  $t$ , it becomes more interesting. The scale space representation collection is a solution for the diffusion equation. Therefore it has the useful property of

$$\partial_t L = \frac{1}{2} \nabla^2 L = \frac{1}{2} (\partial_{xx} + \partial_{yy}) L \quad (4)$$

with the initial condition of  $L(x, y; 0) = f(x, y)$ . So now, scale derivatives can be represented as spatial derivatives.

Now all these representations and operators are useful for continuous functions, but the images we deal with are discrete and contain quantized intensity values. So we must now understand how these operations and properties apply to the discrete domain.

The scale space representation is still defined in a similar fashion. The following is the discrete version of the scale space operation on the function  $f(x)$ , which only has a single spatial variable.

$$L(x; t) = (T(\cdot; t) * f(\cdot))(x; t) \quad (5)$$

In this expression,  $T$  represents the discrete version of the Gaussian kernel and is further evaluated as

$$T(n; t) = e^{-t} I_n(t) \quad (6)$$

where  $I_n$  is the modified Bessel functions of integer order given by

$$I_n(x) = i^{-\alpha} J_\alpha(ix) = \sum_{m=0}^{\infty} \frac{1}{m! \Gamma(m + \alpha + 1)} \left(\frac{x}{2}\right)^{2m + \alpha} \quad (7)$$

Now that the one dimensional case is understood for the scale space, we can expand this to two dimensions. After all, images are composed of two dimensions, so it makes sense that these operators can act on two dimensional functions. The two dimensional scale space representation for discrete variables is given by the following

$$L(x, y; t) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} T(m; t) T(n; t) f(x - m, y - n) \quad (8)$$

Even with the discrete case, the scale space representation must still satisfy the semidiscretized version of the diffusion equation. Therefore by taking a scale derivative of the function we must have the following

$$\partial_t L = \frac{1}{2} ((1 - \gamma) \nabla_5^2 L + \gamma \nabla_{\times}^2 L) \quad (9)$$

where  $\gamma \in [0, 1]$  is a hyperparameter and,

$$(\nabla_5^2 f)_{0,0} = f_{-1,0} + f_{+1,0} + f_{0,-1} + f_{0,+1} - 4f_{0,0} \quad (10)$$

$$(\nabla_{\times}^2 f)_{0,0} = \frac{1}{2} (f_{-1,-1} + f_{-1,+1} + f_{+1,-1} + f_{+1,+1} - 4f_{0,0}) \quad (11)$$

Note that  $f_{-1,1}$  represents  $f(x - 1, y + 1)$ . Now that we have defined what discrete scale spaces and their derivatives look like, we can start to define what an edge is.

### 3 Defining an Edge in Scale Space

We classify edge points as points which have gradient magnitudes that assume a maximum in the direction of the gradient. Let us denote the magnitude of the gradient of  $L$  as  $L_g$ . Then we define an edge in the scale space with the following conditions

$$\begin{aligned} L_{gg} &= 0 \\ L_{ggg} &< 0 \end{aligned} \tag{12}$$

The way we can then represent these conditions in terms of spatial derivatives is as follows

$$\begin{aligned} L_{gg} &= L_x^2 L_{xx} + 2L_x L_y L_{xy} + L_y^2 L_{yy} = 0 \\ L_{ggg} &= L_x^3 L_{xxx} + 3L_x^2 L_y L_{xxy} + 3L_x L_y^2 L_{xyy} + L_y^3 L_{yyy} < 0 \end{aligned} \tag{13}$$

This is useful for determining edges at a single scale, but if we are to determine edges over multiple scales, we must also develop an edge strength metric,  $\varepsilon_{norm} L$ . Thus this adds two more conditions that must be satisfied for an edge to be classified in the scale space.

$$\begin{aligned} \partial_t(\varepsilon_{norm} L(x, y; t)) &= 0 \\ \partial_{tt}(\varepsilon_{norm} L(x, y; t)) &< 0 \end{aligned} \tag{14}$$

Then equation 12 in conjunction with 14 form the necessary constraints for us to define an edge in the scale space. Now we must define what we mean by edge strength and give a clearer idea of  $\varepsilon_{norm}$ .

The approach we took was to let our edge strength metric be defined by the gradient magnitude that has been normalized by our  $\gamma$  factor. In this case we can define it as

$$\begin{aligned} G_\gamma L &= L_{g,\gamma}^2 \\ &= t^\gamma (L_x^2 + L_y^2) \end{aligned} \tag{15}$$

Then the strength of the edges is found by computing the path integral over the maximal connected edge  $\Gamma$  given by

$$G(\Gamma) = \int_{(x;t) \in \Gamma} \sqrt{(G_\gamma L)(x; t)} \, ds \tag{16}$$

where  $ds^2 = dx^2 + dy^2$ .

### 4 Implementation

For our implementation we will be using the following image to run our tests on



Figure 1: Original Grayscale Lena Image

Figure 1 is a standard test image for image processing and is good for our purposes since it contains a variety of different edges with different strengths.

So what we first do is define our convolution functions that we will be using in code.

```

1  function combine_kernels(kers...)
2      return reduce(conv2, kers)
3  end
4
5  function convolve_image(I, kers...)
6      kernel = combine_kernels(kers...)
7      return imfilter(I, centered(kernel))
8  end
9
10 function convolve_scale_space(L, kers...)
11     return mapslices(
12         scale_slice -> convolve_image(scale_slice, kers...),
13         L,
14         (1,2)
15     )
16 end
17
18 function convolve_gaussian(img, sigma)
19     # The dimension of the convolution matrix
20     length = 8*ceil(Int, sigma) + 1
21     return imfilter(img, reflect(Kernel.gaussian((sigma, sigma), (length
22         , length))))
23 end

```

These will be used for computing all the necessary two dimensional matrix convolutions that we will use throughout the rest of the code. The next step is to then define our range of scales used and which derivative kernels we will be using to compute two dimensional discrete spatial derivatives. The following code shows the definitions for our variables that will be used for all further calculations.

```

1  # Parameters
2  gamma = 1
3  scales = exp.(linspace(0, log(50), 40))
4
5  # Load the image
6  img = float.(ColorTypes.Gray.(testimage("lena_color_512")))
7
8  # Define derivative convolution matrices
9  Dy = Array(parent(Kernel.ando5()[1]))
10 Dx = Array(parent(Kernel.ando5()[2]))
11
12 # Normalized the derivatives
13 Dx /= sum(Dx .* (Dx .> 0))
14 Dy /= sum(Dy .* (Dy .> 0))

```

The derivative matrices used for our project are the ando5 matrices which are defined as the following

$$[D_y] = \begin{bmatrix} -0.003776 & -0.010199 & 0.0 & 0.010199 & 0.003776 \\ -0.026786 & -0.070844 & 0.0 & 0.070844 & 0.026786 \\ -0.046548 & -0.122572 & 0.0 & 0.122572 & 0.046548 \\ -0.026786 & -0.070844 & 0.0 & 0.070844 & 0.026786 \\ -0.003776 & -0.010199 & 0.0 & 0.010199 & 0.003776 \end{bmatrix} \quad (17)$$

$$[D_x] = [D_y]^T$$

From this point, we then can compute the scale space representation of our image and compute its spatial derivatives. The following section of code defines just how to do that.

```

1  # Scale space representation
2  L = cat(3, (convolve_gaussian(img, sigma) for sigma in scales)...)
3
4  # First order derivatives
5  Lx = convolve_scale_space(L, Dx)
6  Ly = convolve_scale_space(L, Dy)
7
8  # Second order derivatives
9  Lxx = convolve_scale_space(Lx, Dx)
10 Lxy = convolve_scale_space(Lx, Dy)
11 Lyy = convolve_scale_space(Ly, Dy)
12
13 # Third order derivatives
14 Lxxx = convolve_scale_space(Lxx, Dx)
15 Lxxy = convolve_scale_space(Lxx, Dy)
16 Lxyy = convolve_scale_space(Lxy, Dy)
17 Lyyy = convolve_scale_space(Lyy, Dy)

```

By convolving our ando5 derivative matrix with the scale space representation, we can take spatial derivatives in both the  $x$  and  $y$  dimension. To give an idea of what the images look like, we have the following spatial derivatives of the image below.

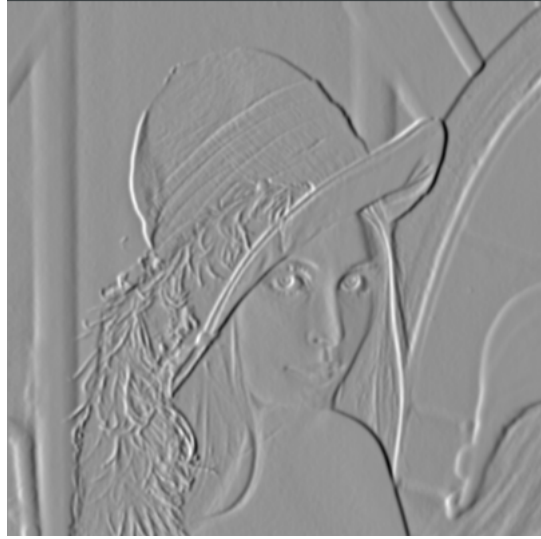


Figure 2: Derivative of image in  $x$  direction



Figure 3: Derivative of image in  $y$  direction

Now that we have computed the spatial derivatives of the scale space representation of the image, we can check if we fulfill the conditions in (12). The next step is to then calculate the edge strength derivatives necessary to fulfill the conditions of (14). The code block below shows the computations for the edge strength based on the gradient magnitude and its derivatives.

```

1 # Shape the scales vector to be a vector with depth
2 scales3 = reshape(scales, 1, 1, length(scales))
3
4 # Definition of the gradient edge strength (magnitude)
5 const GL = scales3.^(gamma).*(Lx.^2+Ly.^2)
6
7 # Derivative of edge strength gradient with respect to scale
8 const GLt = @. gamma*scales3^(gamma-1)*(Lx^2 + Ly^2) + scales3^gamma*(
    Lx*(Lxxx + Lxyy) + Ly*(Lxyy + Lyyy))
9
10 # Second derivative of edge strength gradient with respect to scale
11 const GLtt = @. (gamma*(gamma - 1)*scales3^(gamma - 2)*(Lx^2 + Ly^2) +
    2gamma*scales3^(gamma-1)*(Lx*(Lxxx + Lxyy) + Ly*(Lxyy + Lyyy)) +
    scales3^gamma/2*((Lxxx + Lxyy)^2 + (Lxyy + Lyyy)^2 + Lx*(Lxxxxx + 2
    Lxxxxy + Lxyyyy) + Ly*(Lxxxxy + 2Lxyyyy + Lyyyyy))) < 0

```

## Appendix

Listing 1: edge\_detect.jl

```

1 using ImageFiltering
2 using TestImages
3 using ImageView
4 using Base.Cartesian
5 using ProgressMeter
6 using FileIO
7
8 # Parameters
9 gamma = 1
10 scales = exp.(linspace(0, log(50), 40))
11
12 # Load the image
13 img = float.(ColorTypes.Gray.(testimage("lena_color_512")))
14
15 # Define derivative convolution matrices
16 Dy = Array(parent(Kernel.ando5()[1]))
17 Dx = Array(parent(Kernel.ando5()[2]))
18
19 Dx /= sum(Dx .* (Dx .> 0))
20 Dy /= sum(Dy .* (Dy .> 0))
21
22 function combine_kernels(kers...)
23     return reduce(conv2, kers)
24 end
25
26 function convolve_image(I, kers...)
27     kernel = combine_kernels(kers...)
28     return imfilter(I, centered(kernel))
29 end
30
31 function convolve_scale_space(L, kers...)
32     return mapslices(
33         scale_slice -> convolve_image(scale_slice, kers...),
34         L,
35         (1,2)
36     )
37 end

```

```

38
39 function convolve_gaussian(img, sigma)
40     # The dimension of the convolution matrix
41     length = 8*ceil(Int, sigma) + 1
42     return imfilter(img, reflect(Kernel.gaussian((sigma, sigma), (length
        , length))))
43 end
44
45 L = cat(3, (convolve_gaussian(img, sigma) for sigma in scales)...)
46
47 Lx = convolve_scale_space(L, Dx)
48 Ly = convolve_scale_space(L, Dy)
49
50 Lxx = convolve_scale_space(Lx, Dx)
51 Lxy = convolve_scale_space(Lx, Dy)
52 Lyy = convolve_scale_space(Ly, Dy)
53
54 Lxxx = convolve_scale_space(Lxx, Dx)
55 Lxxy = convolve_scale_space(Lxx, Dy)
56 Lxyy = convolve_scale_space(Lxy, Dy)
57 Lyyy = convolve_scale_space(Lyy, Dy)
58
59 # Lxxxx = convolve_scale_space(Lxxx, Dx)
60 # Lxxxy = convolve_scale_space(Lxxx, Dy)
61 # Lxxyy = convolve_scale_space(Lxxy, Dy)
62 # Lxyyy = convolve_scale_space(Lxyy, Dy)
63 # Lyyyy = convolve_scale_space(Lyyy, Dy)
64
65 Lxxxxx = convolve_scale_space(Lxxx, Dx, Dx)
66 Lxxxxy = convolve_scale_space(Lxxx, Dx, Dy)
67 Lxxxxyy = convolve_scale_space(Lxxx, Dy, Dy)
68 Lxxxyy = convolve_scale_space(Lxxy, Dy, Dy)
69 Lxyyyy = convolve_scale_space(Lxyy, Dy, Dy)
70 Lyyyyy = convolve_scale_space(Lyyy, Dy, Dy)
71
72 const Lvv = @. Lx^2*Lxx + 2Lx*Ly*Lxy + Ly^2*Lyy
73 const Lvvv = @. (Lx^3*Lxxx + 3Lx^2*Ly*Lxxy + 3Lx*Ly^2*Lxyy + Ly^3*Lyyy)
    < 0
74
75 # Shape the scales vector to be a vector with depth
76 scales3 = reshape(scales, 1, 1, length(scales))
77
78 # Definition of the gradient edge strength (magnitude)
79 const GL = scales3.^(gamma).*(Lx.^2+Ly.^2)
80
81 # Derivative of edge strength gradinet with respect to scale
82 const GLt = @. gamma*scales3^(gamma-1)*(Lx^2 + Ly^2) + scales3^gamma*(
    Lx*(Lxxx + Lxxy) + Ly*(Lxxy + Lyyy))
83
84 # Second derivative of edge strength gradinet with respect to scale
85 const GLtt = @. (gamma*(gamma - 1)*scales3^(gamma - 2)*(Lx^2 + Ly^2) +
    2gamma*scales3^(gamma-1)*(Lx*(Lxxx + Lxyy) + Ly*(Lxxy + Lyyy)) +
    scales3^gamma/2*((Lxxx + Lxyy)^2 + (Lxxy + Lyyy)^2 + Lx*(Lxxxxx + 2
    Lxxxxy + Lxyyyy) + Ly*(Lxxxyy + 2Lxxyyy + Lyyyyy))) < 0
86
87 Z12 = Lvvv .& GLtt
88
89 function linear_interpolate(p1, p2, v1, v2)

```



```

90     return (abs(v1)*collect(p1) + abs(v2)*collect(p2))/(abs(v1) + abs(v2
    ))
91 end
92
93 function segment_intersect(p1, p2, p3, p4, e)
94     p13, p43, p21 = p1 - p3, p4 - p3, p2 - p1
95
96     # If the line segments have zero length
97     if norm(p43) < e || norm(p21) < e
98         return Nullable()
99     end
100
101     d1343 = dot(p13, p43)
102     d4321 = dot(p43, p21)
103     d1321 = dot(p13, p21)
104     d4343 = dot(p43, p43)
105     d2121 = dot(p21, p21)
106
107     numer = d1343 * d4321 - d1321 * d4343;
108     denom = d2121 * d4343 - d4321 * d4321;
109
110     if abs(denom) < e
111         return Nullable()
112     end
113
114     mua = numer/denom
115     mub = (d1343 + d4321 * mua) / d4343
116     pa = p1 + mua * p21
117     pb = p3 + mub * p43
118
119     return Nullable((norm(pa - pb), (pa + pb)/2))
120 end
121
122 const cube_edges = [
123     # bottom edges
124     ((1,1,1), (1,2,1)),
125     ((1,2,1), (2,2,1)),
126     ((2,2,1), (2,1,1)),
127     ((2,1,1), (1,1,1)),
128
129     # side edges
130     ((1,1,1), (1,1,2)),
131     ((1,2,1), (1,2,2)),
132     ((2,2,1), (2,2,2)),
133     ((2,1,1), (2,1,2)),
134
135     # top edges
136     ((1,1,2), (1,2,2)),
137     ((1,2,2), (2,2,2)),
138     ((2,2,2), (2,1,2)),
139     ((2,1,2), (1,1,2))
140 ]
141
142 const cube_faces = [
143     ([ 0, 0, -1], ((1,1,1), (2,1,1), (1,2,1), (2,2,1))),
144     ([ 0, -1, 0], ((1,1,1), (2,1,1), (1,1,2), (2,1,2))),
145     ([ 1, 0, 0], ((2,1,1), (2,1,2), (2,2,2), (2,2,1))),
146     ([-1, 0, 0], ((1,1,1), (1,1,2), (1,2,1), (1,2,2))),
147     ([ 0, 1, 0], ((1,2,1), (2,2,1), (1,2,2), (2,2,2))),

```

```

148     ([ 0, 0, 1], ((1,1,2), (1,2,2), (2,2,2), (2,1,2)))
149 ]
150
151 function marching_cubes(x, y, t, visited)
152     if visited[x, y, t]
153         return Set()
154     end
155
156     visited[x, y, t] = true
157     const corners = (x:x+1, y:y+1, t:t+1)
158
159     # Note: Maybe they don't need to be in the same corner
160     if !(any(view(GLtt, corners...)) && any(view(Lvvv, corners...)))
161         return Set()
162     end
163
164     @views Z1, Z2 = Lvv[corners...], GLt[corners...]
165     Z1_crossings = Array{Tuple{NTuple{3,Int}, NTuple{3,Int}, Array{
        Float64, 1}}, 1}()
166     Z2_crossings = Array{Tuple{NTuple{3,Int}, NTuple{3,Int}, Array{
        Float64, 1}}, 1}()
167
168     # Find all sign crossings w/ linear interpolation
169     for (a, b) in cube_edges
170         if signbit(Z1[a...]) != signbit(Z1[b...])
171             push!(Z1_crossings, (a, b, linear_interpolate(a, b, Z1[a...],
                Z1[b...])))
172         end
173
174         if signbit(Z2[a...]) != signbit(Z2[b...])
175             push!(Z2_crossings, (a, b, linear_interpolate(a, b, Z2[a...],
                Z2[b...])))
176         end
177     end
178
179     const epsilon = 10 * eps()
180
181     face_intersections = []
182     result = Set()
183     for (normal, face) in cube_faces
184         Z1_zeros, Z2_zeros = [], []
185         for (a, b, mid) in Z1_crossings
186             if a in face && b in face
187                 push!(Z1_zeros, mid)
188             end
189         end
190
191         for (a, b, mid) in Z2_crossings
192             if a in face && b in face
193                 push!(Z2_zeros, mid)
194             end
195         end
196
197         # Reject if there are more than two crossings for either
198         # invariant
199         if !(length(Z1_zeros) == length(Z2_zeros) == 2)
200             continue
201         end

```

```

202         # Check the intersection of the segments defined by the two
           lines
203         intersect = segment_intersect(Z1_zeros..., Z2_zeros..., epsilon)
204         if isnull(intersect)
205             continue
206         end
207
208         # Check that the intersection lies on a face
209         distance, midpoint = get(intersect)
210         if distance > epsilon || !all(1 - epsilon .<= midpoint .<= 2 +
           epsilon)
211             continue
212         end
213
214         push!(face_intersections, normal)
215     end
216
217     if length(face_intersections) == 2
218         for normal in face_intersections
219             next_voxel = [x, y, t] + normal
220             if all(1 .<= next_voxel .<= size(visited))
221                 union!(result, marching_cubes(next_voxel..., visited))
222             end
223         end
224         push!(result, (x, y, t))
225     end
226
227     return result
228 end
229
230 function find_edges()
231     voxel_visited = falses((x->x-1).(size(L)))
232     edges = []
233     p = Progress(prod(size(voxel_visited)), 1)
234     @nloops 3 i voxel_visited begin
235         edge = marching_cubes((@ntuple 3 i)..., voxel_visited)
236         if length(edge) > 0
237             push!(edges, edge)
238         end
239         next!(p)
240     end
241     return edges
242 end
243
244 function flatten_edges(edges)
245     edge_flat = reduce(union, edges)
246     edge_map = falses((x->x-1).(size(L)))
247     for (x, y, t) in union(edge_flat)
248         edge_map[x,y,t] = true
249     end
250     return edge_map
251 end
252
253 function planar_zeros(Lp)
254     Lp_pos = signbit.(Lp)
255     Lp_zeros = falses(Lp)
256     for (i, scale) in enumerate(scales)
257         for x in 2:(size(Lp)[1]-1)
258             for y in 2:(size(Lp)[2]-1)

```

```

259         @views neighbors = Lp_pos[x-1:x+1, y-1:y+1, i]
260         if (Lp_pos[x,y,i] && !all(neighbors)) ||
261             (!Lp_pos[x,y,i] && any(neighbors))
262             Lp_zeros[x,y,i] = true
263         end
264     end
265 end
266 end
267 return Lp_zeros
268 end
269
270 function scale_zeros(Lp)
271     Lp_pos = signbit.(Lp)
272     Lp_zeros = falses(Lp)
273     for i in 2:length(scales)-1
274         Lp_zeros[:, :, i] = (Lp_pos[:, :, i-1] .!= Lp_pos[:, :, i]) .| (Lp_pos
275            [:, :, i] .!= Lp_pos[:, :, i+1])
276     end
277     return Lp_zeros
278 end
279
280 function scale_maxima(Lp)
281     Lp_pos = signbit.(Lp)
282     Lp_zeros = falses(Lp)
283     for i in 2:length(scales)
284         Lp_zeros[:, :, i] = (Lp_pos[:, :, i-1] .& .!Lp_pos[:, :, i])
285     end
286     return Lp_zeros
287 end
288
289 function edge_importance(edge)
290     total = 0
291     for (x, y, t) in edge
292         total += sqrt(GL[x,y,t])
293     end
294     return total
295 end
296
297 function n_strongest_edges(edges, n)
298     return sort(edges, by=edge_importance, rev=true)[1:n]
299 end
300
301 function flatten_scale(Lp)
302     return mapslices(any, Lp, 3)
303 end
304
305 function main()
306     edges = find_edges()
307     n_strongest = n_strongest_edges(edges, 500)
308     edge_flat = flatten_edges(n_strongest)
309     save("output.png", flatten_scale(edge_flat))
310 end

```

## References