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 uniques 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 imformation with regards to the strength of the edges detected. The majority of these algorithms will usually convolve the image with a static matrix to caclulate edges and does 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 appraoch 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

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

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

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

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

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 is has the useful property of

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

with the inital 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 inetsnity 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 operationg on the function $f(x)$, which only has a single spatial variable.

$$L(x;t) = (T(\cdot;t) * f(\cdot))(x;t) \tag{5}$$

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

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

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

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

Now that the one dimensional case is understood for the scale space, we can expand this to two dimensions. After all, images are compsed of two dimesnisons, 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

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

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

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

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

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

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

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 magnitueds 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 satisified 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 neccessary 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 ε_{norm}.

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

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

The first scale derivative of the gradient magnitude is calculated as follows

$$\partial_t(G_\gamma L) = \gamma t^{\gamma-1}(L_x^2 + L_y^2) + t^\gamma (L_x(L_{xxx} + L_{yyy}) + L_y(L_{xxy} + L_{yyy})) \tag{16}$$

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

$$G(\Gamma) = \int_{(x;t)\in\Gamma} \sqrt{(G_\gamma L)(x; t)}\ \mathrm{d}s \tag{17}$$

where $\mathrm{d}s^2 = \mathrm{d}x^2 + \mathrm{d}y^2$.

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.

```
function combine_kernels(kers...)
    return reduce(conv2, kers)
end

function convolve_image(I, kers...)
    kernel = combine_kernels(kers...)
    return imfilter(I, centered(kernel))
end

function convolve_scale_space(L, kers...)
    return mapslices(
        scale_slice -> convolve_image(scale_slice, kers...),
        L,
        (1,2)
    )
end

function convolve_gaussian(img, sigma)
    # The dimension of the convolution matrix
    length = 8*ceil(Int, sigma) + 1
    return imfilter(img, reflect(Kernel.gaussian((sigma, sigma), (length
        , length))))
end
```

These will be used for computing all the neccessary 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.

```
# Parameters
gamma = 1
scales = exp.(linspace(0, log(50), 40))

# Load the image
img = float.(ColorTypes.Gray.(testimage("lena_color_512")))

# Define derivative convolution matrices
Dy = Array(parent(Kernel.ando5()[1]))
Dx = Array(parent(Kernel.ando5()[2]))

# Normalized the derivatives
Dx /= sum(Dx .* (Dx .> 0))
Dy /= sum(Dy .* (Dy .> 0))
```

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

$$\begin{aligned}
[D_y] &= \begin{bmatrix} -0.003776 & -0.010199 & 0.0 & 0.010199 & 0.003776 \\ -0.026786 & -0.070844 & 0.0 & 0.070844 & 0.026786 \\ -0.046548 & -0.122572 & 0.0 & 0.122572 & 0.046548 \\ -0.026786 & -0.070844 & 0.0 & 0.070844 & 0.026786 \\ -0.003776 & -0.010199 & 0.0 & 0.010199 & 0.003776 \end{bmatrix} \\
[D_x] &= [D_y]^T
\end{aligned} \tag{18}$$

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.

```
# Scale space representation
L = cat(3, (convolve_gaussian(img, sigma) for sigma in scales)...)

# First order derivatives
Lx = convolve_scale_space(L, Dx)
Ly = convolve_scale_space(L, Dy)

# Second order derivatives
Lxx = convolve_scale_space(Lx, Dx)
Lxy = convolve_scale_space(Lx, Dy)
Lyy = convolve_scale_space(Ly, Dy)

# Third order derivatives
Lxxx = convolve_scale_space(Lxx, Dx)
Lxxy = convolve_scale_space(Lxx, Dy)
Lxyy = convolve_scale_space(Lxy, Dy)
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.

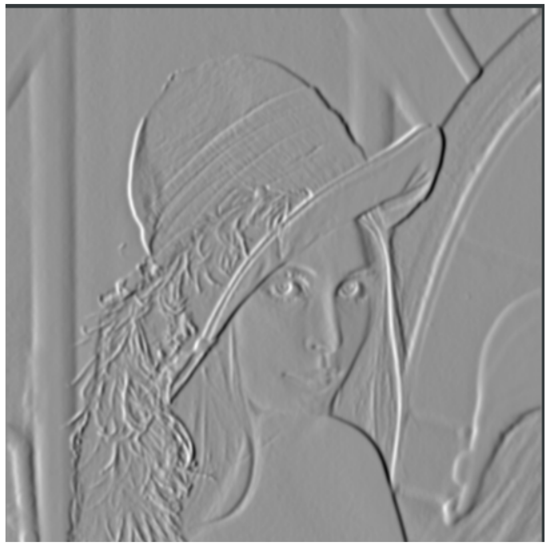

Figure 2: Derivative of image in x direction

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 neccessary 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.

```julia
# Shape the scales vector to be a vector with depth
scales3 = reshape(scales, 1, 1, length(scales))

# Definition of the gradient edge strength (magnitude)
const GL = scales3.^(gamma).*(Lx.^2+Ly.^2)

# Derivative of edge strength gradinet with respect to scale
const GLt = @. gamma*scales3^(gamma-1)*(Lx^2 + Ly^2) + scales3^gamma*(
    Lx*(Lxxx + Lxyy) + Ly*(Lxxy + Lyyy))

# Second derivative of edge strength gradinet with respect to scale
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
    Lxxxyy + Lxyyyy) + Ly*(Lxxxxy + 2Lxxyyy + Lyyyyy))) < 0
```

With these calculations we now can check all the conditions neccessary to define an edge in scale space.

Appendix

Listing 1: edge_detect.jl

```julia
using ImageFiltering
using TestImages
using ImageView
using Base.Cartesian
using ProgressMeter
using FileIO

# Parameters
gamma = 1
scales = exp.(linspace(0, log(50), 40))

# Load the image
img = float.(ColorTypes.Gray.(testimage("lena_color_512")))

# Define derivative convolution matrices
Dy = Array(parent(Kernel.ando5()[1]))
Dx = Array(parent(Kernel.ando5()[2]))

Dx /= sum(Dx .* (Dx .> 0))
Dy /= sum(Dy .* (Dy .> 0))

function combine_kernels(kers...)
    return reduce(conv2, kers)
end

function convolve_image(I, kers...)
    kernel = combine_kernels(kers...)
    return imfilter(I, centered(kernel))
end

function convolve_scale_space(L, kers...)
    return mapslices(
        scale_slice -> convolve_image(scale_slice, kers...),
        L,
        (1,2)
```

```
    )
end

function convolve_gaussian(img, sigma)
    # The dimension of the convolution matrix
    length = 8*ceil(Int, sigma) + 1
    return imfilter(img, reflect(Kernel.gaussian((sigma, sigma), (length, length))))
end

L = cat(3, (convolve_gaussian(img, sigma) for sigma in scales)...)

Lx = convolve_scale_space(L, Dx)
Ly = convolve_scale_space(L, Dy)

Lxx = convolve_scale_space(Lx, Dx)
Lxy = convolve_scale_space(Lx, Dy)
Lyy = convolve_scale_space(Ly, Dy)

Lxxx = convolve_scale_space(Lxx, Dx)
Lxxy = convolve_scale_space(Lxx, Dy)
Lxyy = convolve_scale_space(Lxy, Dy)
Lyyy = convolve_scale_space(Lyy, Dy)

# Lxxxx = convolve_scale_space(Lxxx, Dx)
# Lxxxy = convolve_scale_space(Lxxx, Dy)
# Lxxyy = convolve_scale_space(Lxxy, Dy)
# Lxyyy = convolve_scale_space(Lxyy, Dy)
# Lyyyy = convolve_scale_space(Lyyy, Dy)

Lxxxxx = convolve_scale_space(Lxxx, Dx, Dx)
Lxxxxy = convolve_scale_space(Lxxx, Dx, Dy)
Lxxxyy = convolve_scale_space(Lxxx, Dy, Dy)
Lxxyyy = convolve_scale_space(Lxxy, Dy, Dy)
Lxyyyy = convolve_scale_space(Lxyy, Dy, Dy)
Lyyyyy = convolve_scale_space(Lyyy, Dy, Dy)

const Lvv = @. Lx^2*Lxx + 2Lx*Ly*Lxy + Ly^2*Lyy
const Lvvv = @. (Lx^3*Lxxx + 3Lx^2*Ly*Lxxy + 3Lx*Ly^2*Lxyy + Ly^3*Lyyy) < 0

# Shape the scales vector to be a vector with depth
scales3 = reshape(scales, 1, 1, length(scales))

# Definition of the gradient edge strength (magnitude)
const GL = scales3.^(gamma).*(Lx.^2+Ly.^2)

# Derivative of edge strength gradinet with respect to scale
const GLt = @. gamma*scales3^(gamma-1)*(Lx^2 + Ly^2) + scales3^gamma*(Lx*(Lxxx + Lxyy) + Ly*(Lxxy + Lyyy))

# Second derivative of edge strength gradinet with respect to scale
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 + 2Lxxxyy + Lxyyyy) + Ly*(Lxxxxy + 2Lxxyyy + Lyyyyy))) < 0

Z12 = Lvvv .& GLtt

```

```
function linear_interpolate(p1, p2, v1, v2)
    return (abs(v1)*collect(p1) + abs(v2)*collect(p2))/(abs(v1) + abs(v2))
end

function segment_intersect(p1, p2, p3, p4, e)
    p13, p43, p21 = p1 - p3, p4 - p3, p2 - p1

    # If the line segments have zero length
    if norm(p43) < e || norm(p21) < e
        return Nullable()
    end

    d1343 = dot(p13, p43)
    d4321 = dot(p43, p21)
    d1321 = dot(p13, p21)
    d4343 = dot(p43, p43)
    d2121 = dot(p21, p21)

    numer = d1343 * d4321 - d1321 * d4343;
    denom = d2121 * d4343 - d4321 * d4321;

    if abs(denom) < e
        return Nullable()
    end

    mua = numer/denom
    mub = (d1343 + d4321 * mua) / d4343
    pa = p1 + mua * p21
    pb = p3 + mub * p43

    return Nullable((norm(pa - pb), (pa + pb)/2))
end

const cube_edges = [
    # bottom edges
    ((1,1,1), (1,2,1)),
    ((1,2,1), (2,2,1)),
    ((2,2,1), (2,1,1)),
    ((2,1,1), (1,1,1)),

    # side edges
    ((1,1,1), (1,1,2)),
    ((1,2,1), (1,2,2)),
    ((2,2,1), (2,2,2)),
    ((2,1,1), (2,1,2)),

    # top edges
    ((1,1,2), (1,2,2)),
    ((1,2,2), (2,2,2)),
    ((2,2,2), (2,1,2)),
    ((2,1,2), (1,1,2))
]

const cube_faces = [
    ([ 0, 0,-1], ((1,1,1), (2,1,1), (1,2,1), (2,2,1))),
    ([ 0,-1, 0], ((1,1,1), (2,1,1), (1,1,2), (2,1,2))),
    ([ 1, 0, 0], ((2,1,1), (2,1,2), (2,2,2), (2,2,1))),
    ([-1, 0, 0], ((1,1,1), (1,1,2), (1,2,1), (1,2,2))),
```

```
    ([ 0, 1, 0], ((1,2,1), (2,2,1), (1,2,2), (2,2,2))),
    ([ 0, 0, 1], ((1,1,2), (1,2,2), (2,2,2), (2,1,2)))
]

function marching_cubes(x, y, t, visited)
    if visited[x, y, t]
        return Set()
    end

    visited[x, y, t] = true
    const corners = (x:x+1, y:y+1, t:t+1)

    # Note: Maybe they don't need to be in the same corner
    if !(any(view(GLtt, corners...)) && any(view(Lvvv, corners...)))
        return Set()
    end

    @views Z1, Z2 = Lvv[corners...], GLt[corners...]
    Z1_crossings = Array{Tuple{NTuple{3,Int}, NTuple{3,Int}, Array{Float64, 1}}, 1}()
    Z2_crossings = Array{Tuple{NTuple{3,Int}, NTuple{3,Int}, Array{Float64, 1}}, 1}()

    # Find all sign crossings w/ linear interpolation
    for (a, b) in cube_edges
        if signbit(Z1[a...]) != signbit(Z1[b...])
            push!(Z1_crossings, (a, b, linear_interpolate(a, b, Z1[a...], Z1[b...])))
        end

        if signbit(Z2[a...]) != signbit(Z2[b...])
            push!(Z2_crossings, (a, b, linear_interpolate(a, b, Z2[a...], Z2[b...])))
        end
    end

    const epsilon = 10 * eps()

    face_intersections = []
    result = Set()
    for (normal, face) in cube_faces
        Z1_zeros, Z2_zeros = [], []
        for (a, b, mid) in Z1_crossings
            if a in face && b in face
                push!(Z1_zeros, mid)
            end
        end

        for (a, b, mid) in Z2_crossings
            if a in face && b in face
                push!(Z2_zeros, mid)
            end
        end

        # Reject if there are more than two crossings for either invariant
        if !(length(Z1_zeros) == length(Z2_zeros) == 2)
            continue
        end
```

```
        # Check the intersection of the segments defined by the two
            lines
        intersect = segment_intersect(Z1_zeros..., Z2_zeros..., epsilon)
        if isnull(intersect)
            continue
        end

        # Check that the intersection lies on a face
        distance, midpoint = get(intersect)
        if distance > epsilon || !all(1 - epsilon .<= midpoint .<= 2 +
            epsilon)
            continue
        end

        push!(face_intersections, normal)
    end

    if length(face_intersections) == 2
        for normal in face_intersections
            next_voxel = [x, y, t] + normal
            if all(1 .<= next_voxel .<= size(visited))
                union!(result, marching_cubes(next_voxel..., visited))
            end
        end
        push!(result, (x, y, t))
    end

    return result
end

function find_edges()
    voxel_visited = falses((x->x-1).(size(L)))
    edges = []
    p = Progress(prod(size(voxel_visited)), 1)
    @nloops 3 i voxel_visited begin
        edge = marching_cubes((@ntuple 3 i)..., voxel_visited)
        if length(edge) > 0
            push!(edges, edge)
        end
        next!(p)
    end
    return edges
end

function flatten_edges(edges)
    edge_flat = reduce(union, edges)
    edge_map = falses((x->x-1).(size(L)))
    for (x, y, t) in union(edge_flat)
        edge_map[x,y,t] = true
    end
    return edge_map
end

function planar_zeros(Lp)
    Lp_pos = signbit.(Lp)
    Lp_zeros = falses(Lp)
    for (i, scale) in enumerate(scales)
        for x in 2:(size(Lp)[1]-1)
```

```
            for y in 2:(size(Lp)[2]-1)
                @views neighbors = Lp_pos[x-1:x+1, y-1:y+1, i]
                if (Lp_pos[x,y,i] && !all(neighbors)) ||
                    (!Lp_pos[x,y,i] && any(neighbors))
                    Lp_zeros[x,y,i] = true
                end
            end
        end
    end
    return Lp_zeros
end

function scale_zeros(Lp)
    Lp_pos = signbit.(Lp)
    Lp_zeros = falses(Lp)
    for i in 2:length(scales)-1
        Lp_zeros[:,:,i] = (Lp_pos[:,:,i-1] .!= Lp_pos[:,:,i]) .| (Lp_pos
            [:,:,i] .!= Lp_pos[:,:,i+1])
    end
    return Lp_zeros
end

function scale_maxima(Lp)
    Lp_pos = signbit.(Lp)
    Lp_zeros = falses(Lp)
    for i in 2:length(scales)
        Lp_zeros[:,:,i] = (Lp_pos[:,:,i-1] .& .!Lp_pos[:,:,i])
    end
    return Lp_zeros
end

function edge_importance(edge)
    total = 0
    for (x, y, t) in edge
        total += sqrt(GL[x,y,t])
    end
    return total
end

function n_strongest_edges(edges, n)
    return sort(edges, by=edge_importance, rev=true)[1:n]
end

function flatten_scale(Lp)
    return mapslices(any, Lp, 3)
end

function main()
    edges = find_edges()
    n_strongest = n_strongest_edges(edges, 500)
    edge_flat = flatten_edges(n_strongest)
    save("output.png", flatten_scale(edge_flat))
end
```

References