Scale Space Edge Detection

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

Abstract

Scale space representation of an image 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 Introduction

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

Common methods for detecting edges in images usually consist of convolving a differential matrix with the image that calculates the image's derivatives in the horizontal or vertical direction. Algorithms like the Sobel, Canny, Prewitt, and Robert operators all work in a similar fashion, the only difference being which matrix they convolve with the image. The problem with these methods is that they are not very adaptable. They generally will convolve a static matrix that is not dependent on the image, and could be potentially missing out on a lot of information. For example, given an image **F**, the following computes the gradient of the image in the x and y direction using the Sobel operator

$$\mathbf{G}_x = \begin{bmatrix} +1 & 0 & -1 \\ +2 & 0 & -2 \\ +1 & 0 & -1 \end{bmatrix} * \mathbf{F},\ \mathbf{G}_y = \begin{bmatrix} +1 & +2 & +1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} * \mathbf{F} \tag{1}$$

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 Motivation

3 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

$$L(x;t) = g(x, y; t) * f(x, y) \tag{2}$$

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

$$g(x, y; t) = \frac{1}{2\pi t} e^{-(x^2+y^2)/(2t)} \tag{3}$$

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

$$L_{x^\alpha y^\beta}(\cdot; t) = \partial_{x^\alpha y^\beta} L(\cdot; t) = g_{x^\alpha y^\beta}(\cdot; t) * f(\cdot) \tag{4}$$

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

$$\partial_t L = \frac{1}{2}\nabla^2 L = \frac{1}{2}(\partial_{xx} + \partial_{yy})L \tag{5}$$

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 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 operating on the function $f(x)$, which only has a single spatial variable.

$$L(x; t) = (T(\cdot; t) * f(\cdot))(x; t) \tag{6}$$

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

$$T(n; t) = e^{-t} I_n(t) \tag{7}$$

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

$$I_n(x) = i^{-n} J_n(ix) = \sum_{m=0}^{\infty} \frac{1}{m!\Gamma(m + n + 1)} \left(\frac{x}{2}\right)^{2m+n} \tag{8}$$

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

$$L(x, y; t) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} T(m; t)T(n; t)f(x - m, y - n) \tag{9}$$

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

$$\partial_t L = \frac{1}{2}((1 - \gamma)\nabla_5^2 L + \gamma \nabla_\times^2 L) \tag{10}$$

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

$$(\nabla_5^2 f)_{0,0} = f_{-1,0} + f_{+1,0} + f_{0,-1} + f_{0,+1} - 4f_{0,0} \tag{11}$$

$$(\nabla_\times^2 f)_{0,0} = \frac{1}{2}(f_{-1,-1} + f_{-1,+1} + f_{+1,-1} + f_{+1,+1} - 4f_{0,0}) \tag{12}$$

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.

4 Defining an Edge in Scale Space

Table 1: Gradient magnitude

	Lena	Mandrill	Peppers
$t = 0.1$	Images/lena/lena_GL_0.1.png	Images/mandrill/mandrill_GL_0.1.png	Images/peppers/peppers_GL_0.1.png
$t = 1.7$	Images/lena/lena_GL_1.7.png	Images/mandrill/mandrill_GL_1.7.png	Images/peppers/peppers_GL_1.7.png
$t = 18.7$	Images/lena/lena_GL_18.7.png	Images/mandrill/mandrill_GL_18.7.png	Images/peppers/peppers_GL_18.7.png
$t = 93.6$	Images/lena/lena_GL_93.6.png	Images/mandrill/mandrill_GL_93.6.png	Images/peppers/peppers_GL_93.6.png
$t = 256.0$	Images/lena/lena_GL_256.0.png	Images/mandrill/mandrill_GL_256.0.png	Images/peppers/peppers_GL_256.0.png

Table 2: First scale derivative of gradient magnitude

	Lena	Mandrill	Peppers
$t = 0.1$	Images/lena/lena_GLt_0.1.png	Images/mandrill/mandrill_GLt_0.1.png	Images/peppers/peppers_GLt_0.1.png
$t = 1.7$	Images/lena/lena_GLt_1.7.png	Images/mandrill/mandrill_GLt_1.7.png	Images/peppers/peppers_GLt_1.7.png
$t = 18.7$	Images/lena/lena_GLt_18.7.png	Images/mandrill/mandrill_GLt_18.7.png	Images/peppers/peppers_GLt_18.7.png
$t = 93.6$	Images/lena/lena_GLt_93.6.png	Images/mandrill/mandrill_GLt_93.6.png	Images/peppers/peppers_GLt_93.6.png
$t = 256.0$	Images/lena/lena_GLt_256.0.png	Images/mandrill/mandrill_GLt_256.0.png	Images/peppers/peppers_GLt_256.0.png

Table 3: Second scale derivative of the gradient magnitude

	Lena	Mandrill	Peppers
$t = 0.1$	`Images/lena/lena_GLtt_0.1.png`	`Images/mandrill/mandrill_GLtt_0.1.png`	`Images/peppers/peppers_GLtt_0.1.png`
$t = 1.7$	`Images/lena/lena_GLtt_1.7.png`	`Images/mandrill/mandrill_GLtt_1.7.png`	`Images/peppers/peppers_GLtt_1.7.png`
$t = 18.7$	`Images/lena/lena_GLtt_18.7.png`	`Images/mandrill/mandrill_GLtt_18.7.png`	`Images/peppers/peppers_GLtt_18.7.png`
$t = 93.6$	`Images/lena/lena_GLtt_93.6.png`	`Images/mandrill/mandrill_GLtt_93.6.png`	`Images/peppers/peppers_GLtt_93.6.png`
$t = 256.0$	`Images/lena/lena_GLtt_256.0.png`	`Images/mandrill/mandrill_GLtt_256.0.png`	`Images/peppers/peppers_GLtt_256.0.png`

Table 4: Maxima of scale space in the v direction

	Lena	Mandrill	Peppers
$t = 0.1$	`Images/lena/lena_Lv_maxima_0.1.png`	`Images/mandrill/mandrill_Lv_maxima_0.1.png`	`Images/peppers/peppers_Lv_maxima_0.1.png`
$t = 1.7$	`Images/lena/lena_Lv_maxima_1.7.png`	`Images/mandrill/mandrill_Lv_maxima_1.7.png`	`Images/peppers/peppers_Lv_maxima_1.7.png`
$t = 18.7$	`Images/lena/lena_Lv_maxima_18.7.png`	`Images/mandrill/mandrill_Lv_maxima_18.7.png`	`Images/peppers/peppers_Lv_maxima_18.7.png`
$t = 93.6$	`Images/lena/lena_Lv_maxima_93.6.png`	`Images/mandrill/mandrill_Lv_maxima_93.6.png`	`Images/peppers/peppers_Lv_maxima_93.6.png`
$t = 256.0$	`Images/lena/lena_Lv_maxima_256.0.png`	`Images/mandrill/mandrill_Lv_maxima_256.0.png`	`Images/peppers/peppers_Lv_maxima_256.0.pn`

Table 5: The second derivative of the scale space with respect to the v direction

	Lena	Mandrill	Peppers
$t = 0.1$	Images/lena/lena_Lvv_0.1.png	Images/mandrill/mandrill_Lvv_0.1.png	Images/peppers/peppers_Lvv_0.1.png
$t = 1.7$	Images/lena/lena_Lvv_1.7.png	Images/mandrill/mandrill_Lvv_1.7.png	Images/peppers/peppers_Lvv_1.7.png
$t = 18.7$	Images/lena/lena_Lvv_18.7.png	Images/mandrill/mandrill_Lvv_18.7.png	Images/peppers/peppers_Lvv_18.7.png
$t = 93.6$	Images/lena/lena_Lvv_93.6.png	Images/mandrill/mandrill_Lvv_93.6.png	Images/peppers/peppers_Lvv_93.6.png
$t = 256.0$	Images/lena/lena_Lvv_256.0.png	Images/mandrill/mandrill_Lvv_256.0.png	Images/peppers/peppers_Lvv_256.0.png

Table 6: The third derivative of the scale space with respect to the v direction

	Lena	Mandrill	Peppers
$t = 0.1$	Images/lena/lena_Lvvv_0.1.png	Images/mandrill/mandrill_Lvvv_0.1.png	Images/peppers/peppers_Lvvv_0.1.png
$t = 1.7$	Images/lena/lena_Lvvv_1.7.png	Images/mandrill/mandrill_Lvvv_1.7.png	Images/peppers/peppers_Lvvv_1.7.png
$t = 18.7$	Images/lena/lena_Lvvv_18.7.png	Images/mandrill/mandrill_Lvvv_18.7.png	Images/peppers/peppers_Lvvv_18.7.png
$t = 93.6$	Images/lena/lena_Lvvv_93.6.png	Images/mandrill/mandrill_Lvvv_93.6.png	Images/peppers/peppers_Lvvv_93.6.png
$t = 256.0$	Images/lena/lena_Lvvv_256.0.png	Images/mandrill/mandrill_Lvvv_256.0.png	Images/peppers/peppers_Lvvv_256.0.png

A useful thing to do when dealing with image features is to set up a coordinate system in terms of the local directional derivatives. In this case, we do the same thing as the paper does and setup a coordinate system (u, v) at any point (x_0, y_0) on the image. Here, the v-axis is parallel to the gradient direction at the point (x_0, y_0), and the u-axis is perpendicular to the gradient direction at the same point. So then we define necessary angles as

$$\begin{pmatrix}\cos\alpha \\ \sin\alpha\end{pmatrix} = \frac{1}{\sqrt{L_x^2 + L_y^2}} \begin{pmatrix} L_x \\ L_y \end{pmatrix} \Bigg|_{(x_0,y_0)} \tag{13}$$

With our angles defined, we can then define our (u, v) coordinate system by taking the respective partial derivatives of our spatial coordinate system.

$$\partial_u = \sin\alpha\ \partial_x - \cos\alpha\ \partial_y \tag{14}$$
$$\partial_v = \cos\alpha\ \partial_x + \sin\alpha\ \partial_y \tag{15}$$

So then we can define our partials of L with respect to this new (u, v) coordinate system. So then we define an edge in this coordinate system with the following conditions.

$$\begin{aligned} L_{vv} &= 0 \\ L_{vvv} &< 0 \end{aligned} \tag{16}$$

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

$$\begin{aligned} L_{vv} &= L_x^2 L_{xx} + 2L_x L_y L_{xy} + L_y^2 L_{yy} = 0 \\ L_{vvv} &= L_x^3 L_{xxx} + 3L_x^2 L_y L_{xxy} + 3L_x L_y^2 L_{xyy} + L_y^3 L_{yyy} < 0 \end{aligned} \tag{17}$$

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.

$$\begin{aligned} \partial_t(\varepsilon_{norm} L(x, y; t)) &= 0 \\ \partial_{tt}(\varepsilon_{norm} L(x, y; t)) &< 0 \end{aligned} \tag{18}$$

Then equation 16 in conjunction with 18 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

$$G_\gamma L = L_{g,\gamma}^2 \tag{19}$$
$$= t^\gamma (L_x^2 + L_y^2) \tag{20}$$

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

$$\partial_t(G_\gamma L) = \gamma t^{\gamma-1}(L_x^2 + L_y^2) + t^\gamma (L_x(L_{xxx} + L_{yyy}) + L_y(L_{xxy} + L_{yyy})) \tag{21}$$

Then the second scale derivative of the gradient magnitude is

$$\begin{aligned} \partial_{tt} &= \gamma(\gamma-1)t^{\gamma-2}(L_x^2 + L_y^2) \\ &\quad + 2\gamma t^{\gamma-1}(L_x(L_{xyy} + L_{xxx}) + L_y(L_{xxy} + L_{yyy})) \\ &\quad + \frac{t^\gamma}{2}\big((L_{xxx} + L_{xyy})^2 + (L_{xxy} + L_{yyy})^2\big) \\ &\quad + L_x(L_{xxxxx} + 2L_{xxxyy} + L_{xyyyy}) \\ &\quad + L_y(L_{xxxxy} + 2L_{xxyyy} + L_{yyyyy})\big) \end{aligned} \tag{22}$$

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

$$G(\Gamma) = \int_{(x;t)\in\Gamma} \sqrt{(G_\gamma L)(x;t)}\ \ \mathrm{d}s \tag{23}$$

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

5 Implementation

5.1 Introduction

We have chosen to implement our edge detector in the Julia scientific computing language. We chose Julia because it is very fast and has strong support for linear algebra and image processing through third party packages. Our implementation can be found in full at `https://github.com/PythonNut/diffgeo-edge-detector`.

5.2 Convolution Framework

Both our smoothing and derivative operators are based on discrete approximations represented by convolution matrices. Therefore, we implement several convenience functions to apply convolutions to images quickly and with a high degree of numerical stability.

```julia
function combine_kernels(kers...)
    return reduce(fastconv, 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
```

Our strategy is to apply convolutions to images in batches. For a set k of convolution matrices, we first merge them into a single larger matrix K by reduction via convolution. We can do this in any order, since convolutions are associative and commutative. We do this using a highly numerically stable convolution algorithm described in `https://arxiv.org/abs/1612.08825`. We do this to avoid accumulating numerical errors by repeatedly convolving the image using a less stable algorithm, since the matrices we will be combining are generally small, it is reasonable to use a slower, but more exact algorithm on them.

We then convolve the image with K using a fast tiled FFT algorithm provided by ImageFiltering.jl, which is less numerically stable but features faster asymptotic times for large matrices which is helpful as our images can have millions of pixels.

5.3 Scale Space Generation

Now, we can generate a tensor representation of scale space.

```julia
function gaussian_kernel(t, length)
    G = hcat(besselix.(-length:length, t))
    return combine_kernels(G, G')
end

function convolve_gaussian(img, t)
    # The dimension of the convolution matrix
    length = 4*ceil(Int, sqrt(t))
    kernel = gaussian_kernel(t, length)
    return convolve_image(img, kernel)
end
```

First, we compute a discrete approximation of the gaussian kernel using the bessel function. To do this, we take advantage of the separability property of the gaussian kernel, which states that an N-dimensional gaussian kernel g_N can be written:

$$G_N(x_1, \cdots, x_N, t) = G(x_1, t) * \cdots * G(x_N, t)$$

This allows us to compute a 1-dimensional gaussian kernel G, and produce a 2-dimensional kernel by convolving G with its transpose.

We then compute an appropriately large kernel size, such that the values of the matrix near the edges are close to the machine epsilon. This is important, as we wish to avoid blocking artifacts in our smoothing as a result of our discrete approximation. We then apply the generated gaussian kernel to the image.

```julia
img = float.(ColorTypes.Gray.(load("Images/block.jpg")))
scales = exp.(linspace(log(0.1), log(256), 40))
L = cat(3, (convolve_gaussian(img, t) for t in scales)...)
```

To generate the scale space itself, we choose a set of 40 scales with uniform effective scale distribution, as described in `http://www.diva-portal.org/smash/get/diva2:473381/FULLTEXT01.pdf`. We then create 40 copies of the image, and smooth each images by their respective gaussian kernels. We then concatenate them to form a rank 3 tensor representing the scale space.

5.4 Discrete Derivatives

We now define our matrix approximations of the derivative operators.

```julia
Dy = Array(parent(Kernel.ando5()[1]))
Dx = Array(parent(Kernel.ando5()[2]))

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

The derivative matrices used for our project are the 5×5 matrices defined in `http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=841757&tag=1`. Written out, they are:

$$[D_x] = \begin{bmatrix} -0.003776 & -0.010199 & 0.0 & 0.010199 & 0.003776 \\ -0.026786 & -0.070844 & 0.0 & 0.070844 & 0.026786 \\ -0.046548 & -0.122572 & 0.0 & 0.122572 & 0.046548 \\ -0.026786 & -0.070844 & 0.0 & 0.070844 & 0.026786 \\ -0.003776 & -0.010199 & 0.0 & 0.010199 & 0.003776 \end{bmatrix} \tag{24}$$
$$[D_y] = [D_x]^T$$

Additionally, we normalize the magnitudes of the derivative matrices, such that the sum of the positive values of the matrix sum to 1, as described in `https://link.springer.com/article/10.1007/BF01664794`. This is so the overall magnitude of the derivative remains comparable across derivatives of different order. We then define a framework for taking the partial derivatives of a scale space in the spatial directions using these operators.

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

function spatial_derivative(L, x, y)
    return convolve_scale_space(L, fill(Dx, x)..., fill(Dy, y)...)
end
```

To perform spatial convolution, we convolve the scale space independently at every scale. To compute $L_{x^i y^j}$ we spatially convolve i times by $[D_x]$ and j times by $[D_y]$. Thus, we can compute the higher order partial derivatives of L like so:

```
Lx = spatial_derivative(L, 1, 0)
Ly = spatial_derivative(L, 0, 1)
# ...
Lxxyyy = spatial_derivative(L, 2, 3)
# ...
```

We calculate all first, second, third, and fifth order spatial partial derivatives of L for use in the future.

5.5 Differential Entities

Now that we have computed the spatial derivatives of the scale space, we can check if we fulfill the conditions described in (Eq. 16) and (Eq. 18).

5.5.1 Spatial edge conditions

Here is a direct translation of (Eq. 16) into Julia.

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

Here, L_{vv} contains the first derivative of the gradient magnitude in the gradient (i.e. v) direction, and L_{vvv} contains the inverted sign of the second derivative of the magnitude in the v direction. These definitions describe the local maxima of of the edge strength with respect to the spatial direction.

5.5.2 Scale edge conditions

Here is a direct translation of (Eq. 18) into Julia.

```
scales3 = reshape(scales, 1, 1, length(scales))
gamma = 1
GL = scales3.^(gamma).*(Lx.^2+Ly.^2)
GLt = @. gamma*scales3^(gamma-1)*(Lx^2 + Ly^2) + scales3^gamma*(Lx*(
    Lxxx + Lxyy) + Ly*(Lxxy + Lyyy))
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
```

To do this, we transpose the scales array into the third matrix, and then broadcast it across the scale space to form the normalization coefficients.

$\mathcal{G}_\gamma L$ represents the normalized gradient magnitude, which serves as our edge strength measure, $\partial_t(\mathcal{G}_\gamma L)$ represents the first partial of $\mathcal{G}_\gamma L$ with respect to the scale parameter as described in the Theory, and $\partial_{tt}(\mathcal{G}_\gamma L)$ represents the inverted sign of the second partial of $\mathcal{G}_\gamma L$ with respect to scale. These definitions describe the local maxima of of the edge strength with respect to the scale parameter.

5.6 Marching Cubes

5.6.1 Introduction

We note that the equations $\partial_t(\mathcal{G}_\gamma L) = 0$ and $L_{vv} = 0$ define isosurfaces $\mathcal{Z}_1$ and $\mathcal{Z}_2$, embedded in L. Computing the intersection of these two surfaces and requiring the local maxima conditions $\partial_{tt}(\mathcal{G}_{\gamma\text{-norm}} L) < 0$ and $L_{vvv} < 0$ yields a family of isocurves, also embedded in L, which directly correspond to the edges in the image.

However, since we only have access to $\partial_t(\mathcal{G}_\gamma L)$ and L_{vv} at the lattice points of L, we cannot solve for their intersection analytically. Instead, we rely on an approximation algorithm to find the edges.

We have chosen to implement the algorithm sketched by Lindeberg, a variation of the marching cubes algorithm.

5.6.2 Geometric Constants

To help us, we define the following constants describing the geometry of a cube.

```julia
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)))
]
```

The first defines each edge of the cube as pairs of coordinates, indexing at 1, as Julia is 1 indexed. The second defines each edge of face of the edge as a pair of the normal vector and the vertices on the corners of that face.

5.6.3 Recursive Flood Fill

In our implementation, L is broken down into voxels with a value at every corner. We then use recursion to fill out the edge connected at that point.

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

    visited[x, y, t] = true
```

Since edges are connected sets of points, we know that if a given voxel has already been visited it cannot be part of this edge, so we skip it immediately. We next check that our sign conditions $L_{vvv} < 0$ and $\partial_{tt}(\mathcal{G}_\gamma L) < 0$ are each negative in at least one vertex of the voxel.

```julia
    const corners = (x:x+1, y:y+1, t:t+1)

    if !(any(view(GLtt, corners...)) && any(view(Lvvv, corners...)))
      return Set()
    end
```

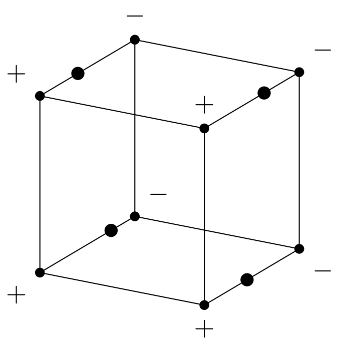
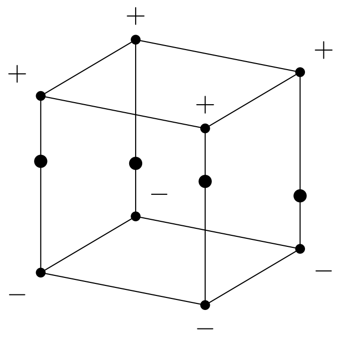
We know that if there is no negative value in the second derivatives, there can be no maxima of L_v and $\mathcal{G}_\gamma L$ inside this voxel, so we skip it immediately. Next, we check for zeros of L_{vv} and $\partial_t(\mathcal{G}_\gamma L)$ as indicated by sign changes.

```
    @views Z1, Z2 = Lvv[corners...], GLt[corners...]
    Z1_crossings = []
    Z2_crossings = []

    # 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
```

To do this, we iterate over every edge. Then for each differential entity we check if the endpoints of the edge differ in sign. If they do, we estimate the position of the zero crossing of the surface of that point using linear interpolation and save that point along with the edge along with the endpoints of the edge. Here is a visualization of one such result:

L_{vv}	$\mathcal{G}L_t$
	

Then, for each face, we determine if there is an intersection of the two isosurfaces on that face.

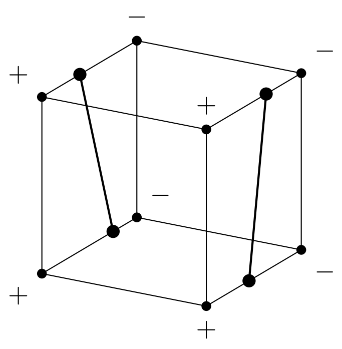
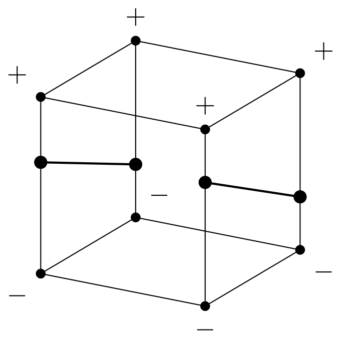
```
    for (normal, face) in cube_faces
        Z1_zeros, Z2_zeros = [], []
        for (a, b, mid) in Z1_crossings
            if a in face && b in face
                push!(Z1_zeros, mid)
            end
        end

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

We collect all of the edges with a zero crossing contained in that face, as defined by its endpoints.

```
        if !(length(Z1_zeros) == length(Z2_zeros) == 2)
            continue
        end
```

If there is only one edge with a zero crossing on that face, the face cannot have a zero crossing inside it, so we skip further processing on this face. If there are four zero crossings of either invariant on this face, then the orientation of the zero crossing on that face is ambiguous, so we skip it[1]. If there are exactly two zero crossings on the edges of the face, then we can approximate the zero crossing on that face with the line connecting the two points. Here is a visualization of one case satisfying these requirements:

L_{vv}	$\mathcal{G}L_t$
	

Next, we check if the lines defined by our two zero conditions intersect.

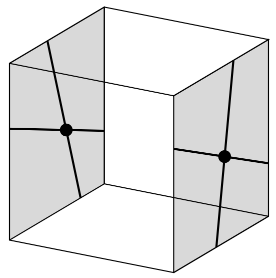
```
        intersect = segment_intersect(Z1_zeros..., Z2_zeros..., epsilon)
        if isnull(intersect)
            continue
        end
```

To do this, we calculate the distance between the lines defined by each isosurface. If the segments are parallel, the distance is not defined, so we skip the face. Next, we check that the lines intersect.

```
        const epsilon = 10 * eps()
        distance, midpoint = get(intersect)
        if distance > epsilon || !all(1 - epsilon .<= midpoint .<= 2 +
            epsilon)
            continue
        end

        push!(face_intersections, normal)
```

We do so by checking that the distance between the two lines is close to the machine epsilon, additionally we require that the intersection point of the two lines lies within the face itself. If the face fails these conditions, we skip it otherwise we register an intersection on that face. Here is a visualization of a case that satisfies our criterion:

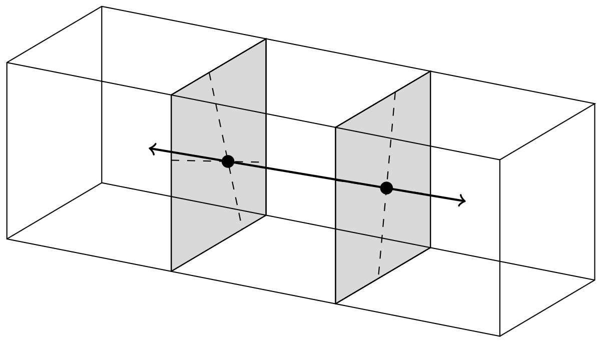


If there are exactly two faces containing intersections, then the voxel contains an edge segment.

[1]In the future, we could try to resolve this situation, since there are only two possible cases.

```
    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
```

We extend the line defined by connecting the estimated intersection points on each face into the neighboring voxels as shown here:



The process is then repeated on the neighboring voxels as defined by the extrapolation. This allows us to recursively trace out the full connected edge at this point.

Appendix

References