

ABSTRACT

A Beefy Frangi Filter for Noisy Vascular Segmentation and Network Connection in PCSVN

By

Lucas Wukmer

December 2018

Recent statistical analysis of placental features has suggested the usefulness of studying key features of the placental chorionic surface vascular network (PCSVN) as a measure of overall neonatal health. A recent study has suggested that reliable reporting of these features may be useful in identifying risks of certain neurodevelopmental disorders at birth. The necessary features can be extracted from an accurate tracing of the surface vascular network, but such tracings must still be done manually, with significant user intervention. Automating this procedure would not only allow more data acquisition to study the potential effects of placental health on later conditions, but may ideally serve as a real-time diagnostic for neonatal risk factors as well.

Much work has been to develop reliable vascular extraction methods for well-known image domains (such as retinal MRA images) using Hessian-based filters, namely the (multiscale) Frangi filter. It is desirable to extend these arguments to placental images, but this approach is greatly hindered by the inherent irregularity of the placental surface as a whole, which introduces significant noise into the image domain. A recent attempt was made to apply an additional local curvilinear filter to the Frangi result in an effort to remove some noise from the final extraction.

Here we propose an alternate extraction method. First, we use arguments from Frangi's original paper to provide a proper selection of parameters for our particular image domain. Using the same arguments from differential geometry that gave rise to the Frangi filter, we calculate the leading principal direction (eigenvector of the Hessian) to indicate the directionality of curvilinear features at a particular scale. We are then able to apply an

appropriately-oriented morphological filter to our Frangi targets at select scales to remove noise. This approach differs significantly from previous efforts in that morphological filtering will take place at each scale space, rather than being performed one time following multiscale synthesis. Noise removal performed in this way is expected to aide in coherent interpretation of targets that should appear in a connected network.

Finally, we discuss an important advancement in implementation—scale space conversion for differentiation (i.e. gaussian blur) via Fast Fourier Transform (FFT) rather than a more traditional convolution with a gaussian kernel, which offers a significant speedup. This thesis will also contain a general, in depth summary of both multiscale Hessian filters and scale-space theory.

We demonstrate the effectiveness of our improved vascular extraction technique on several of the following image domains: a private database of barium-injected samples provided by University of Rochester, uninjected/raw placental samples from Placental Analytics LLC, a collection of simulated images, the DRIVE and STARE databases of retinal MRAs, and a new collection of computer-generated images with significant curvilinear content.

Time permitting, this research will be extended to include a method of network connection, so that a logically connected vascular network is realized (i.e. network completion).

A Beefy Frangi Filter for Noisy Vascular Segmentation and Network Connection in
PCSVN

A THESIS

Presented to the Department of Department of Mathematics and Statistics
California State University, Long Beach

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in Applied Mathematics

Committee Members:
Jen-Mei Chang, Ph.D. (Chair)
James von Brecht, Ph.D.
William Ziemer, Ph.D.

College Designee:
Tangan Gao, Ph.D.

By Lucas Wukmer
B.S., 2013, University of California, Los Angeles
December 2018

ACKNOWLEDGEMENTS

Acknowledgments go here.

TABLE OF CONTENTS

ABSTRACT	ii
ACKNOWLEDGEMENTS	ii
LIST OF TABLES	iv
LIST OF FIGURES	v
1. INTRODUCTION	1
2. MATHEMATICAL METHODS	2
Linear-shift and Rotational invariance	27
Continuity of scale parameter	27
Causality condition	28
Fourier Transform of a continuous 1D signal	34
Fourier Transform of a Discrete 1D signal	35
1D Discrete Fourier Transform (DFT)	36
2D Discrete Fourier Transform Convolution Theorem	36
FFT	38
3. IMPLEMENTATIONS	43
4. RESEARCH PROTOCOL	44
5. RESULTS AND ANALYSIS	51
6. CONCLUSION	54
APPENDICES	55
A. CODE LISTINGS	56
A CODE LISTINGS	56
BIBLIOGRAPHY	122

LIST OF TABLES

LIST OF FIGURES

1	Tangent plane of a graph	6
2	The graph of a cylindrical ridge of radius r , as given by $f(x,y)$ as given above. ...	18
3	The principal eigenvectors at a ridge like structure	23
4	A representative placental NCS sample with vascular tracing	44
5	Preprocessed files from an NCS sample	45
6	Demonstration of boundary dilation	47
7	Frangi vesselness score at several scales.....	49
8	Frangi vesselness score at several scales (inset).....	50
9	”True” false positives and “False” false positives.....	52
10	Luminance/brightness thresholding is bad.	53

CHAPTER 1

INTRODUCTION

The Applied Problem

Reference Nen's paper and latest autism risk paper.

Context in Image Processing

- brief background of math image processing methods
- what's been tried in this applied problem
 - nen [5]
 - catalina's paper
 - kara's paper
 - other domains

Research Goals

Segue from previous paragraph, talk about strengths and weaknesses of other methods and what this research aims to accomplish. Include 'research questions' that could allow a reader to answer the question "will this research work for my problem?". "Elevator pitch" maybe goes here.

Roadmap

Outline of the thesis ("firstly" bullshit)

CHAPTER 2

MATHEMATICAL METHODS

Our goal is establish a resource efficient method of finding curvilinear content in 2D grayscale digital images using concepts of differential geometry. We proceed by (i) establishing a standard method of viewing these images as 2D surfaces, (ii) developing a minimal yet rigorous distillation of differential geometry to obtain suitable quantifiers for the study of curvilinear structure in 3D surfaces, (iii) establishing a filter based on these quantifiers, and finally (iv) developing methods necessary for efficient computation of the filter.

Problem Setup in Image Processing

A digital 2D grayscale image is given by a $M \times N$ array of pixels, whose intensity is given by an integer value between 0 and 255.

Definition 2.1 (Image as a pixel matrix).

$$\mathbf{I} \in \mathbb{N}^{M \times N} \quad \text{with} \quad 0 \leq I_{ij} \leq 2^8 - 1$$

For theoretical purposes, we wish to consider any such picture to ultimately be a sampling of a 2D continuous surface. We also require that this surface is sufficiently continuous as to admit the existence of second partial derivatives.

Definition 2.2 (Image as an interpolated surface).

$$h : \mathbb{R}^2 \rightarrow \mathbb{R} \quad \text{with} \quad h \in C^2(\mathbb{R}^2), \quad \text{where} \quad h(i, j) = I_{ij} \quad \forall (i, j) \in \{0, \dots, M\} \times \{0, \dots, N\} \subset \mathbb{N}^2$$

That is, the function h is identical to the pixel matrix \mathbf{I} at all integer inputs, and simply a “smooth enough” interpolation of those points for all other values.

It is of course necessary to admit that \mathbf{I} is not really a perfect representation of the underlying “content” within the picture. Not only is information lost when \mathbf{I} is stored as an integer, there are also elements of noise and anomalies of lighting that would constitute noise

to the original signal. There are multiple treatments of image processing that do address this discrepancy in a pragmatic way [3], especially when the goal is noise reduction. However, we will be content to simply represent the pixels of \mathbf{I} as the ultimate “cause” of the surface h in definition 2.2, and worry not about how faithfully that sampling corresponds to the real world. Moreover, though our samples in the image domain have been carefully prepared (as outlined in), there are numerous shortcomings therein, and improvements to the veracity of our original signal could be made from many angles. Though we shall draw upon the notion of the pixel matrix \mathbf{I} as a sampling again to motivate our development of scale space theory in section 2.4, we ultimately use these techniques because we find them successful to our problem.

Differential Geometry

We wish to describe the structure of an image as a surface. To do this, we develop the notion of curvature of a surface in \mathbb{R}^3 in a standard way, following [8] (although any undergraduate text in Differential Geometry should prove satisfactory).

Preliminaries of Differential Geometry

Given an open subset $U \subset \mathbb{R}^2$ and a twice differentiable function $h : U \rightarrow \mathbb{R}$ (as in definition 2.2) we define the graph, f , of h in the following definition.

Definition 2.3. *The surface f is a graph (of the function h) when*

$$f : U \rightarrow \mathbb{R}^3 \quad \text{by} \quad f(u_1, u_2) = (u_1, u_2, h(u_1, u_2)), \quad u = (u_1, u_2) \in U \subset \mathbb{R}^2$$

Since the graph f is clearly one-to-one by definition, we may readily associate any input $u \in U$ with its corresponding output $p \in f[U]$, i.e. $p = f(u) = f(u_1, u_2) = (u_1, u_2, h(u_1, u_2))$, depending on whether we wish to focus on a point of a graph in terms of its input or in terms of the structure of the graph itself.

Our development of curvature ultimately will hinge upon a careful consideration of the tangent plane of f at a point p , for we will require a concrete definition of both the tangent

space within the domain and image of f , as well as the so called "differential" of f , the lattermost of which we will only define for the immediate case required. Seeing that f is one-to-one should make a lot of this futzing about complete overkill, but I've yet to find a way to distill it. That is, this development works for any parametrized surface element, not necessarily a graph. Whatever for now.

Definition 2.4 (Tangent space of U at u).

$$T_u U = \{u\} \times \mathbb{R}^2$$

Definition 2.5 (Tangent space of \mathbb{R}^3 at p).

$$T_p \mathbb{R}^3 = \{p\} \times \mathbb{R}^3$$

It is immediately clear that $T_u U$ and $T_p \mathbb{R}^3$ are isomorphic to \mathbb{R}^2 and \mathbb{R}^3 , respectively, and we can easily visualize elements of $T_u U$ are tangent vectors in \mathbb{R}^2 "originating" at the point u , and elements of $T_p \mathbb{R}^3$ are tangent vectors "originating" at the point p .

Definition 2.6 (The differential of f at a point u). $Df|_u$ is the map from $T_u U$ into \mathbb{R}^3 given by

$$Df|_u : T_u U \rightarrow T_{f(u)} \mathbb{R}^3 \quad \text{by} \quad w \mapsto J_f(u) \cdot v$$

where $J_f(u)$ is the Jacobian of f evaluated at some fixed point $u \in U$, i.e. the matrix

$$J_f(u) = \left[\frac{\partial f_i}{\partial u_j} \Big|_u \right]_{i,j}$$

Although not necessary presently, we could just as easily consider the differential of an arbitrary function as a map between tangent vectors in the function's domain and tangent vectors in its range. We could also just identify this as mapping $U \rightarrow \mathbb{R}^3$ by the obvious isomorphism described above. and then differential of f at x is simply a linear transformation of between the tangent spaces $T_u U$ and $T_p \mathbb{R}^3$ where the transformation in question is given by the Jacobian. We can define such a differential at any point u in the domain.

With these three definitions, we are equipped to give a formal definition of $T_u f$, the tangent plane of f at an input u .

Definition 2.7 (Tangent plane of a graph).

$$T_u f := Df|_u(T_u U) \subset T_{f(u)} \mathbb{R}^3 = T_p \mathbb{R}^3$$

The vectors of this plane can thus be identified as tangent vectors from $T_u U$ that have been passed through the differential mapping $Df|_u$. We shall denote a generic tangent vector $X \in T_u f$ at point p . We may expand any such vector X in terms of the basis $\left\{ \frac{\partial f}{\partial u_i} \right\}_{i=1,2}$; that is, $\text{span} \left\{ \frac{\partial f}{\partial u_1}, \frac{\partial f}{\partial u_2} \right\} = T_u f$.

Given the level of abstraction above, it may be refreshing to explicitly show the linear independence of this set in the case of an arbitrary graph f .

Lemma 2.1. *When f is a graph, for all points $u \in U$, $\left\{ \frac{\partial f}{\partial u_1}, \frac{\partial f}{\partial u_2} \right\}$ is in fact a basis for the tangent plane $T_u f$.*

Quite obviously, we're assuming $(1,0), (0,1) \in U$. If this is not the case, we pick some α small enough so that $(\alpha,0)$ and $(0,\alpha)$ are contained and this scaled version would serve as a basis instead.

Proof. Given the definition of a graph f as in definition 2.3, we can directly calculate the partial derivatives of f at a point u .

$$f_{u_1} = (1, 0, h_{u_1}(u)) \quad \text{and} \quad f_{u_2} = (0, 1, h_{u_2}(u))$$

which are obviously linearly independent. Then $Df|_u(1,0) = f_{u_1}$, and $Df|_u(0,1) = f_{u_2}$, which shows $\left\{ \frac{\partial f}{\partial u_1}, \frac{\partial f}{\partial u_2} \right\} \in T_u f$. Thus $\left\{ \frac{\partial f}{\partial u_1}, \frac{\partial f}{\partial u_2} \right\}$ is a linearly independent subset of $T_u f$, and can serve as its basis. \square

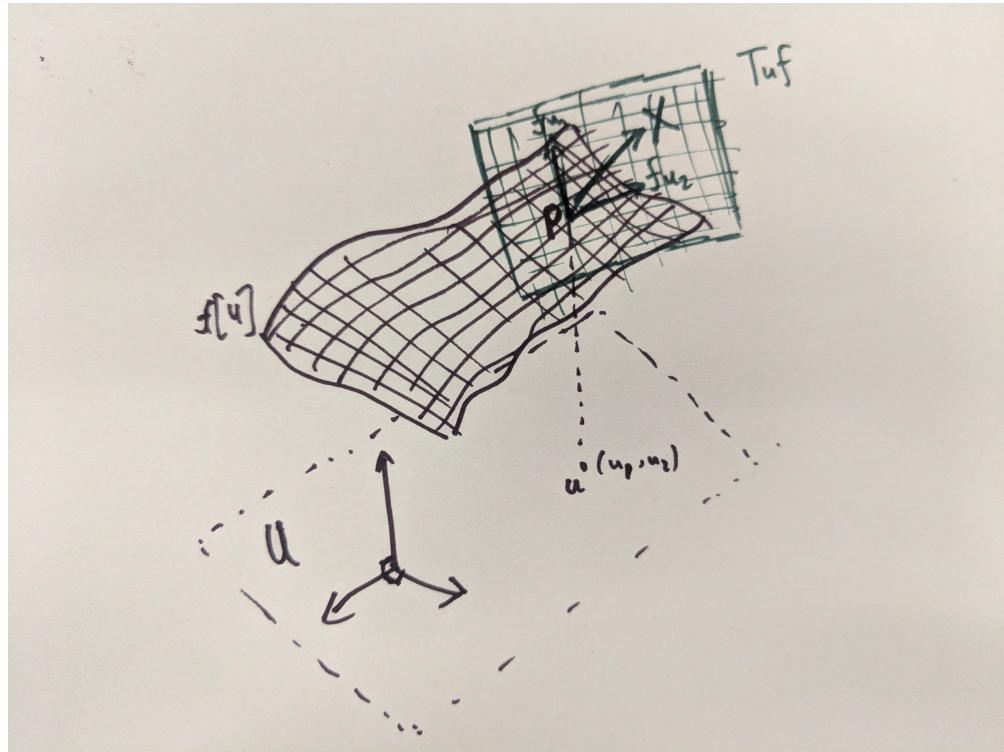


FIGURE 1: Tangent plane of a graph

The partials derivatives of f are not, in general, orthogonal at any point u , unless it happens that h_{u_1} or h_{u_2} is zero. A visualization of some of the above is given in fig. 1, although note that f_{u_1} and f_{u_2} accidentally appear orthogonal.

We now concern ourselves with developing the notion of curvature on a surface. First, we need to consider an arbitrary regular curve (i.e. differentiable, one-to-one, non-zero derivative) contained within the image of f .

Curvature of a surface and its calculation

In the context of a regular arc-length parametrized curve $c : I \rightarrow \mathbb{R}^3$ parametrized along some closed interval $I \in \mathbb{R}$ (that is, a differentiable, one-to-one curve where $c'(s) = 1 \ \forall s \in I$), curvature at a point $s \in I$ is defined simply as the magnitude of the curve's acceleration: $\kappa(s) := \|c''(s)\|$.

To extend the notion of curvature of a surface f , we can consider the curvature of such an arbitrary curve embedded within the surface.

Definition 2.8 (Surface curve). *Given a closed interval $I \subset \mathbb{R}$, we call the regular curve $c : I \rightarrow \mathbb{R}^3$ a surface curve in the event that $\text{image}(c) \subset \text{image}(f)$ entirely. The one-to-one-ness of the graph f ensures that we can define (for the given curve) an intermediary parametrization θ_c so that $c = f \circ \theta_c$. That is,*

$$\theta_c : I \rightarrow U \text{ by } \theta(t) = (\theta_1(t), \theta_2(t))$$

so that $c(t) = f(\theta_c(t)) \ \forall t \in I$, and $c[I] = f[\theta_c[I]]$.

Note as well that the velocity of this particular curve lies within $T_u f$. This can be seen by an elementary application of chain rule:

$$-\frac{dc}{dt} = -\frac{d}{dt}[f(\theta_c(t))] \tag{2.1}$$

$$= -\frac{d}{dt}[f(\theta_1(t), \theta_2(t))] \tag{2.2}$$

$$= \theta'_1(t) \left(\frac{\partial f}{\partial u_1} \right) + \theta'_2(t) \left(\frac{\partial f}{\partial u_2} \right) \in T_u f. \tag{2.3}$$

Considering a point $p \in I$ and its associated point $u = \theta_c(p)$, we wish to compare the curvatures of all (regular) surface curves passing through the point p at some particular velocity.

We now present a main result that provides a notion of curvature of a surface.

Theorem 2.2 (Theorem of Meusnier). *Given a point $u \in U$ and a tangent direction $X \in T_{uf}$, any regular curve on the surface $c : I \rightarrow \text{image}(f)$ with $p \in I : \theta_c(p) = u$ where $c'(p) = X$ will have the same curvature.*

[TODO: provide a visualization of this]

In other words, any two curves on the surface with a common velocity at a given point on the surface will have the same curvature. To prove this, we'll require one final definition.

Definition 2.9 (The Gauss Map). *The Gauss map at a point $p = f(u)$ is the unit normal to the tangent plane*

$$\nu : U \rightarrow \mathbb{R}^3 \quad \text{by} \quad \nu(u) := \frac{\frac{\partial f}{\partial u_1} \times \frac{\partial f}{\partial u_2}}{\left\| \frac{\partial f}{\partial u_1} \times \frac{\partial f}{\partial u_2} \right\|}$$

Each partial above understood to be evaluated at the input $u \in U$; that is, we calculate $\frac{\partial f}{\partial u_i}|_u$. The existence of the cross product in its definition makes it clear that $\nu \perp \frac{\partial f}{\partial u_i}$ each $i = 1, 2$. A simple dimensionality argument of \mathbb{R}^3 implies that these must exist in T_{uf} . However, we can also show it directly:

To show that $\left\{ \frac{\partial \nu}{\partial u_1}, \frac{\partial \nu}{\partial u_2} \right\} \in T_{uf}$, first note that at any particular $u \in U$, $\langle \nu, \nu \rangle = 1 \implies \frac{\partial}{\partial u_i} \langle \nu, \nu \rangle = 0$, and so by chain rule $2 \langle \frac{\partial \nu}{\partial u_i}, \nu \rangle = 0 \implies \frac{\partial \nu}{\partial u_i} \perp \nu$. Since $\nu \perp \text{span} \left\{ \frac{\partial f}{\partial u_i} \right\}$ as well (since ν its outer product), in \mathbb{R}^3 , this implies $\text{span} \left\{ \frac{\partial \nu}{\partial u_i} \right\} \parallel \text{span} \left\{ \frac{\partial f}{\partial u_i} \right\}$.

Thus, we have $\text{span} \left\{ \frac{\partial \nu}{\partial u_1}, \frac{\partial \nu}{\partial u_2} \right\} \subset T_{uf}$ as well and we can also use it as a basis.

We are finally ready to prove theorem 2.2, the Theorem of Meusnier.

Proof. Let $X \in T_{uf}$ be given and consider some curve where $\frac{dc}{dt}(u) = X$ where $X \in T_{uf}$. We wish to decompose the curve's acceleration along the orthogonal vectors X and the Gauss

map $\nu = \nu(u_1, u_2) = \frac{\frac{\partial f}{\partial u_1} \times \frac{\partial f}{\partial u_2}}{\|\frac{\partial f}{\partial u_1} \times \frac{\partial f}{\partial u_2}\|}$ as in definition 2.9. Note that X and ν are indeed orthogonal, as $X \in \text{span} \left\{ \frac{\partial f}{\partial u_i} \right\} = T_u f$, and $\nu \perp T_u f$. We then have (at this fixed point $u = \theta_c(p)$)

$$c'' = \langle c'', X \rangle X + \langle c'', \nu \rangle \nu \quad (2.4)$$

Because c is a regular curve, we either have $c'' = 0$, or $c' \perp c''$, since $\|c'\| = 1$ implies $0 = \frac{d}{dt} \langle c', c' \rangle = 2 \langle c'', c' \rangle$. Thus

$$\langle c'', X \rangle = \langle c'', c' \rangle = 0$$

and we can rewrite the second coefficient of eq. (2.4) using the chain rule:

$$\langle c'', \nu \rangle = \frac{\partial}{\partial t} [\langle c', \nu \rangle] - \langle c', \frac{\partial \nu}{\partial t} \rangle \quad (2.5)$$

$$= \frac{\partial}{\partial t} [\langle X, \nu \rangle] - \langle c', \frac{\partial \nu}{\partial t} \rangle \quad (2.6)$$

$$= 0 - \langle X, \frac{\partial \nu}{\partial t} \rangle \quad (2.7)$$

Thus, we can express the curvature at this point on our selected curve as

$$\|c''\| = \|\langle c'', X \rangle X + \langle c'', \nu \rangle \nu\| = \|0 + \langle c'', \nu \rangle \nu\| \quad (2.8)$$

$$= -\langle X, \frac{\partial \nu}{\partial t} \rangle \|\nu\| \quad (2.9)$$

$$= -\langle X, \frac{\partial \nu}{\partial t} \rangle \quad (2.10)$$

$$= \langle X, -\frac{\partial \nu}{\partial t} \rangle \quad (2.11)$$

We may compute the quantity $-\frac{\partial \nu}{\partial t}$ that appears in eq. (2.11) via chain rule:

$$-\frac{d\nu}{dt} = -\frac{d}{dt} [\nu(u_1, u_2)] \quad (2.12)$$

$$= -\frac{d}{dt} [\nu(\theta_1(t), \theta_2(t))] \quad (2.13)$$

$$= \theta'_1(t) \left(-\frac{\partial \nu}{\partial u_1} \right) + \theta'_2(t) \left(-\frac{\partial \nu}{\partial u_2} \right) \quad (2.14)$$

Identifying $\text{span} \left\{ -\frac{\partial \nu}{\partial u_i} \right\}_{i=1,2}$ as a subset of $T_u f$, we can define a linear transformation L which maps the basis $\left\{ \frac{\partial f}{\partial u_i} \right\}_{i=1,2}$ to this subset:

Definition 2.10 (The Weingarten Map).

$$L : T_u f \rightarrow T_u f \quad \text{given by the composition} \quad L = D\nu \circ (Df)^{-1}.$$

That is, $L\left(\frac{\partial f}{\partial u_i}\right) = -\frac{\partial \nu}{\partial u_i}$ for $i = 1, 2$, where the negative sign comes about from blind adherence to eq. (2.14) and eq. (2.11). This allows us to rewrite the time derivative of the Gauss map eq. (2.12) as

$$-\frac{d\nu}{dt} = \theta'_1(t) \left(-\frac{\partial \nu}{\partial u_1} \right) + \theta'_2(t) \left(-\frac{\partial \nu}{\partial u_2} \right) \quad (2.15)$$

$$= \theta'_1(t) \left(L\left(\frac{\partial f}{\partial u_1}\right) \right) + \theta'_2(t) \left(L\left(\frac{\partial f}{\partial u_2}\right) \right) \quad (2.16)$$

$$= L \left[\theta'_1(t) \left(\frac{\partial f}{\partial u_1} \right) + \theta'_2(t) \left(\frac{\partial f}{\partial u_2} \right) \right] \quad (2.17)$$

$$= L \left(\frac{d}{dt} [f(\theta(t))] \right) = L \left(\frac{d}{dt} [c(t)] \right) = L(X) \quad (2.18)$$

With this, we can re-express the curvature of our curve from eq. (2.11) as the much simpler

$$\|c''\| = \langle X, -\frac{\partial \nu}{\partial t} \rangle = \langle X, L(X) \rangle \quad (2.19)$$

The linear transformation L from definition 2.10, and thereby the computation of curvature given in eq. (2.19), depends only on the point u and the selected direction X , not on the particular curve c at all. \square

To recap, given a point u on the surface and an arbitrary vector X in the tangent plane, we can calculate the curvature of any surface curve with velocity X there. In fact, we refer to this intrinsic quantity as the normal curvature of the surface.

Definition 2.11. *The normal curvature of a surface, denoted κ_ν at point u in the direction X is given by*

$$\kappa_\nu := \langle X, L(X) \rangle$$

In fact, theorem 2.2 shows that the normal curvature is an intrinsic property of the surface—it depends only on the surface at a point, and no reference to any particular curve on the surface is necessary or implied.

The map L introduced in the proof above is known as the Weingarten map and is implicitly defined at each $u \in U$. We wish to make its existence rigorous as well as find a matrix representation for it, using the standard motivation that $L(\frac{\partial f}{\partial u_i}) = -\frac{\partial \nu}{\partial u_i}$.

That is, we may trace any $X \in T_{uf}$ which has been expanded in terms of the basis $\left\{ \frac{\partial f}{\partial u_1}, \frac{\partial f}{\partial u_2} \right\}$ and map it to the span of $\left\{ -\frac{\partial \nu}{\partial u_1}, -\frac{\partial \nu}{\partial u_2} \right\}$.

The Weingarten map can be formally shown to be well-defined, invariant under coordinate transformation [8] in the general case, which is certainly useful for surfaces f that are not graphs. The situation is much less delicate if f is a graph—the linear transformation may be simply constructed, and we proceed by simply calculating its matrix representation.

Lemma 2.3. *The Weingarten map as in definition 2.10 is well-defined for graphs.*

To find a matrix representation for L , (which we will denote $\hat{L} \in R^{2 \times 2}$) we simply wish to find a linear transformation such that $\hat{L} \frac{\partial f}{\partial u_i} \Big|_{T_{uf}} = -\frac{\partial \nu}{\partial u_i} \Big|_{T_{uf}}$ for $i = 1, 2$ where $-X|_{T_{uf}}$ denotes that $X \in T_{uf}$ is being represented in so-called 'local coordinates' for T_{uf} (Strictly speaking, of course $T_{uf} \subset \mathbb{R}^3$ and thus $\frac{\partial f}{\partial u_i} \in \mathbb{R}^3$. Thus when we say $\frac{\partial f}{\partial u_i} \Big|_{T_{uf}}$ we are referring to this 3-vector expanded with respect to a basis for T_{uf}). In matrix form, we describe this situation as

$$\begin{bmatrix} \hat{\mathbf{L}} \end{bmatrix} \begin{bmatrix} \frac{\partial f}{\partial u_1} \\ \downarrow \\ \frac{\partial f}{\partial u_2} \end{bmatrix}_{T_{uf}} = \begin{bmatrix} \hat{\mathbf{L}} \frac{\partial f}{\partial u_1} \\ \downarrow \\ \hat{\mathbf{L}} \frac{\partial f}{\partial u_2} \end{bmatrix}_{T_{uf}} \quad (2.20)$$

$$= \begin{bmatrix} \uparrow \\ -\frac{\partial \nu}{\partial u_1} \\ \downarrow \\ -\frac{\partial \nu}{\partial u_2} \end{bmatrix}_{T_{uf}} \quad (2.21)$$

Now, representing each vector in T_{uf} with respect to the basis $\left\{ \frac{\partial f}{\partial u_i} \right\}$, we have

$$\Rightarrow \begin{bmatrix} \hat{\mathbf{L}} \end{bmatrix} \begin{bmatrix} \leftarrow \frac{\partial f}{\partial u_1} \rightarrow \\ \leftarrow \frac{\partial f}{\partial u_2} \rightarrow \end{bmatrix} \begin{bmatrix} \uparrow & \uparrow \\ \frac{\partial f}{\partial u_1} & \frac{\partial f}{\partial u_2} \\ \downarrow & \downarrow \end{bmatrix} = \begin{bmatrix} \leftarrow \frac{\partial f}{\partial u_1} \rightarrow \\ \leftarrow \frac{\partial f}{\partial u_2} \rightarrow \end{bmatrix} \begin{bmatrix} \uparrow & \uparrow \\ -\frac{\partial \nu}{\partial u_1} & -\frac{\partial \nu}{\partial u_2} \\ \downarrow & \downarrow \end{bmatrix} \quad (2.22)$$

We can simplify this greatly by defining

$$g_{ij} := \langle \frac{\partial f}{\partial u_i}, \frac{\partial f}{\partial u_j} \rangle \quad \text{and} \quad h_{ij} := \langle \frac{\partial f}{\partial u_i}, -\frac{\partial \nu}{\partial u_j} \rangle \quad (2.23)$$

so that

$$\begin{bmatrix} \hat{\mathbf{L}} \end{bmatrix} \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \quad (2.24)$$

Then we rearrange to solve for $\hat{\mathbf{L}}$ as

$$\hat{\mathbf{L}} = \begin{bmatrix} h_{11} & h_{12} \\ h_{21} & h_{22} \end{bmatrix} \begin{bmatrix} g_{11} & g_{12} \\ g_{21} & g_{22} \end{bmatrix}^{-1} \quad (2.25)$$

where $[g_{ij}]$ is clearly invertible, as the set $\left\{ \frac{\partial f}{\partial u_j} \right\}$ is linearly independent.

It should be noted that this matrix representation is accurate not only for the surface of a graph, but for any *generalized* surface $f : U \rightarrow \mathbb{R}^3$ with $u \mapsto (x(u), y(u), z(u))$ as well. We shall later show that this calculation simplifies (somewhat) in the case that our surface is a graph.

Our final goal is to characterize such normal curvatures. Namely, we wish to establish a method of determining in which directions an extremal normal curvature occurs.

Principal Curvatures and Principal Directions

To do so, we shall consider the relationship between the direction X and the normal curvature κ_ν in that direction at some specified u .

First, we need the following lemma:

Lemma 2.4. *If $A \in R^{n \times n}$ is a symmetric real matrix, $v \in R^n$ and given the dot product $\langle \cdot, \cdot \rangle$, we have $\nabla_v \langle v, Av \rangle = 2Av$. In particular, when $A = I$ the identity matrix, we have $\nabla_v \langle v, v \rangle = 2v$.*

Proof. The result is uninterestingly obtained by tracking each (the ‘ith’) component of $\nabla_v \langle v, Av \rangle$:

$$\left(\nabla_v \langle v, Av \rangle \right)_i = \frac{\partial}{\partial v_i} [\langle v, Av \rangle] = \frac{\partial}{\partial v_i} \left[\sum_{j=1}^n v_j (Av)_j \right] \quad (2.26)$$

$$= \frac{\partial}{\partial v_i} \left[\sum_{j=1}^n v_j \sum_{k=1}^n a_{jk} v_k \right] \quad (2.27)$$

$$= \frac{\partial}{\partial v_i} \left[a_{ii} v_i^2 + v_i \sum_{k \neq i} a_{ik} v_k + v_i \sum_{j \neq i} a_{ji} v_j + \sum_{j \neq i} \sum_{k \neq i} v_j a_{jk} v_k \right] \\ (2.28)$$

$$= 2a_{ii} v_i + \sum_{k \neq i} a_{ik} v_k + \sum_{j \neq i} a_{ji} v_j + 0 \quad (2.29)$$

$$= 2a_{ii} v_i + 2 \sum_{k \neq i} a_{ik} v_k = 2 \sum_{k=1}^n a_{ik} v_k = 2(Av)_i \quad (2.30)$$

$$\implies \nabla_v \langle v, Av \rangle = 2Av. \quad (2.31)$$

□

We are now ready for the major result of this section, which ties the Weingarten map to

the notion of normal curvatures.

Theorem 2.5 (Theorem of Olinde Rodrigues). *Fixing a point $u \in U$, a direction $X \in T_u f$ minimizes the normal curvature $\kappa_\nu = \langle \mathbf{L}X, X \rangle$ subject to $\langle X, X \rangle = 1$ iff X is a (normalized) eigenvector of the Weingarten map \mathbf{L} .*

Proof. In the following, we will assume that $X \in T_u f$ is expanded, in local coordinates, i.e. along a two dimensional basis (such as $\left\{\frac{\partial f}{\partial u_i}\right\}_{i=1,2}$) and thus can refer to \mathbf{L} freely as the 2×2 matrix $\widehat{\mathbf{L}}$. Using the method of Lagrange multipliers, we define the Lagrangian:

$$\mathcal{L}(X; \lambda) := \langle \widehat{\mathbf{L}}X, X \rangle - \lambda(\langle X, X \rangle - 1) \quad (2.32)$$

Extremal values occur when $\nabla_{X,\lambda} \mathcal{L}(X; \lambda) = 0$, which results in the two equations

$$\begin{cases} \nabla_X \langle \widehat{\mathbf{L}}X, X \rangle - \lambda \nabla_X (\langle X, X \rangle - 1) = 0 \\ \langle X, X \rangle - 1 = 0 \end{cases} \quad (2.33)$$

The second requirement is simply the constraint that X is normalized. Using the previous lemma, we can simplify the first result as follows:

$$\begin{aligned} \nabla_X \langle \widehat{\mathbf{L}}X, X \rangle - \lambda \nabla_X (\langle X, X \rangle - 1) &= 0 \\ 2\widehat{\mathbf{L}}X - \lambda(2X) &= 0 \\ \implies \widehat{\mathbf{L}}X - \lambda X &= 0 \\ \implies \widehat{\mathbf{L}}X &= \lambda X \end{aligned} \quad (2.34)$$

which implies that X is an eigenvector of $\widehat{\mathbf{L}}$ with corresponding eigenvalue λ ($X \neq 0$ from the second equation of eq. (2.33)). Thus the two hypotheses are exactly equivalent when X is normalized. It is also worth remarking that the corresponding eigenvalue λ is the Lagrangian multiplier itself. \square

Thus, to find the directions of greatest and least curvature of a surface at a point $u \in U$, we simply must calculate the Weingarten map and its eigenvectors. We refer to these directions as follows.

Definition 2.12 (Principal Curvatures and Principal Directions). *The extremal values of normal curvature of a surface at a point $u \in U$ are referred to as **principal curvatures**. The corresponding directions at which normal curvature attains an extremal value are referred to as **principal directions**.*

Our final goal is to explicitly determine a (hopefully simplified) version of the Weingarten map in the case of a graph $f(u_1, u_2) = (u_1, u_2, h(u_1, u_2))$ and calculate the principal directions and curvatures in a simple example.

Theorem 2.6. *When $f : U \rightarrow \mathbb{R}^3$ is given by $(x, y) \mapsto (x, y, h(x, y))$, the matrix representation of the Weingarten map is exactly the Hessian matrix given in (2.1) is given by*

$$\hat{\mathbf{L}} = \text{Hess}(h)\tilde{G}, \quad \text{where } \tilde{G} := \frac{1}{\sqrt{1+h_x^2+h_y^2}} \begin{bmatrix} 1+h_y^2 & -h_x h_y \\ -h_x h_y & 1+h_x^2 \end{bmatrix} \quad (2.35)$$

In particular, given a point $u = (x, y) \in U \subset \mathbb{R}^2$ where $h_x \approx h_y \approx 0$, we have $\tilde{G} \approx \text{Id}$, and thus $\hat{\mathbf{L}} \approx \text{Hess}$.

Proof. First, we can (using chain rule) rewrite each component as in eq. (2.23):

$$h_{ij} = \left\langle \frac{\partial f}{\partial u_i}, -\frac{\partial \nu}{\partial u_j} \right\rangle = \left\langle \frac{\partial^2 f}{\partial u_i \partial u_j}, \nu \right\rangle$$

Now, given our particular surface f , we can calculate each of these components directly.

We have:

$$\begin{aligned} f_x &= (1, 0, h_x), & f_y &= (0, 1, h_y) \\ f_{xx} &= (0, 0, h_{xx}), & f_{xy} &= (0, 0, h_{xy}) = f_{yx}, & f_{yy} &= (0, 0, h_{yy}) \end{aligned} \quad (2.36)$$

and we have the unit normal vector (Gauss map)

$$\nu(u_1, u_2) = \frac{\frac{\partial f}{\partial x} \times \frac{\partial f}{\partial y}}{\left\| \frac{\partial f}{\partial x} \times \frac{\partial f}{\partial y} \right\|} \quad (2.37)$$

$$= \frac{(1, 0, h_x) \times (0, 1, h_y)}{\|\cdots\|} \quad (2.38)$$

$$= \frac{(-h_x, -h_y, 1)}{\sqrt{h_x^2 + h_y^2 + 1}} \quad (2.39)$$

We then calculate each h_{ij} as

$$\begin{aligned} h_{11} &= \left\langle \frac{\partial^2 f}{\partial x^2}, \nu \right\rangle = \frac{h_{xx}}{\sqrt{1 + h_x^2 + h_y^2}} \\ h_{12} &= \left\langle \frac{\partial^2 f}{\partial x \partial y}, \nu \right\rangle = \frac{h_{xy}}{\sqrt{1 + h_x^2 + h_y^2}} = h_{21} \\ h_{22} &= \left\langle \frac{\partial^2 f}{\partial y^2}, \nu \right\rangle = \frac{h_{yy}}{\sqrt{1 + h_x^2 + h_y^2}} \end{aligned} \quad (2.40)$$

and thus the first matrix in eq. (2.25) is given by

$$[h_{ij}] = \frac{1}{\sqrt{1 + h_x^2 + h_y^2}} \text{Hess}(h) \quad (2.41)$$

To calculate the second, we use

$$\begin{aligned} g_{ij} &= \left\langle \frac{\partial f}{\partial u_i}, \frac{\partial f}{\partial u_j} \right\rangle \\ g_{11} &= \langle f_x, f_x \rangle = 1 + h_x^2 \\ g_{12} &= \langle f_x, f_y \rangle = h_x h_y = g_{21} \\ g_{22} &= \langle f_y, f_y \rangle = 1 + h_y^2 \end{aligned} \quad (2.42)$$

and thus

$$[g_{ij}]^{-1} = \begin{bmatrix} 1 + h_x^2 & h_x h_y \\ h_x h_y & 1 + h_y^2 \end{bmatrix}^{-1} = \begin{bmatrix} 1 + h_y^2 & -h_x h_y \\ -h_x h_y & 1 + h_x^2 \end{bmatrix} \quad (2.43)$$

Combining $[h_{ij}]$ and $[g_{ij}]^{-1}$ from eq. (2.43) and eq. (2.41) we arrive at eq. (2.35). \square

Thus the matrix of the Weingarten map \hat{L} is the Hessian matrix exactly at a critical point $u \in U$, where $\nabla h(u) = (h_x(u), h_y(u)) = 0$. Of course this implies that \hat{L} and $\text{Hess}(h)$ have the same eigenvalues and eigenvectors at these points.

To make this a little more explicit, we will calculate the Weingarten map for a relatively simple graph.

The Weingarten map and Principal Curvatures of a Cylindrical Ridge

Let f be the graph given by

$$f : \mathbb{R}^2 \rightarrow \mathbb{R}^3 \text{ by } f(x, y) = (x, y, h(x, y)), \text{ with } h(x, y) = \begin{cases} \sqrt{r^2 - x^2} & -r \leq x \leq r \\ 0 & \text{else} \end{cases} \quad (2.44)$$

We calculate the necessary partial derivatives of f as follows:

$$\frac{\partial f}{\partial x} = \left(1, 0, \frac{-x}{\sqrt{r^2 - x^2}} \right) \quad , \quad \frac{\partial^2 f}{\partial x^2} = \left(0, 0, \frac{-r^2}{(\sqrt{r^2 - x^2})^3} \right) \quad (2.45)$$

$$\frac{\partial f}{\partial y} = (0, 1, 0) \quad , \quad \frac{\partial^2 f}{\partial y^2} = \frac{\partial^2 f}{\partial x \partial y} = 0 \quad (2.46)$$

The gauss map is given by

$$\nu(x, y) = \frac{\frac{\partial f}{\partial x} \times \frac{\partial f}{\partial y}}{\left\| \frac{\partial f}{\partial x} \times \frac{\partial f}{\partial y} \right\|} = \left(\frac{x}{r}, 0, \frac{\sqrt{r^2 - x^2}}{r} \right) \quad (2.47)$$

$$\Rightarrow \frac{\partial \nu}{\partial x} = \left(\frac{1}{r}, 0, \frac{-x}{r\sqrt{r^2 - x^2}} \right) \quad , \quad \frac{\partial \nu}{\partial y} = (0, 0, 0). \quad (2.48)$$

We then calculate matrix elements of the Weingarten map's construction as given in

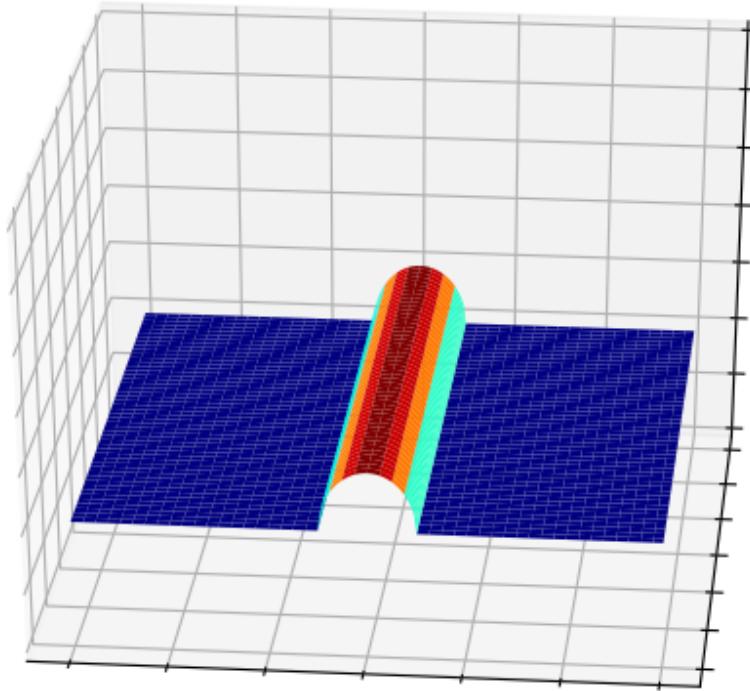


FIGURE 2: The graph of a cylindrical ridge of radius r , as given by $f(x, y)$ as given above.

eq. (2.41) and eq. (2.43) :

$$[h_{ij}] = \frac{1}{\sqrt{1+h_x^2+h_y^2}} \text{Hess}(h) = \frac{1}{\sqrt{1+\left(\frac{x^2}{r^2-x^2}\right)}} \begin{bmatrix} \frac{-r^2}{\sqrt{r^2-x^2}} & 0 \\ 0 & 0 \end{bmatrix} = \begin{bmatrix} \frac{-r}{r^2-x^2} & 0 \\ 0 & 0 \end{bmatrix} \quad (2.49)$$

$$[g_{ij}]^{-1} = \begin{bmatrix} \frac{r^2-x^2}{r^2} & 0 \\ 0 & 1 \end{bmatrix} \quad (2.50)$$

$$\implies \hat{\mathbf{L}} = [h_{ij}][g_{ij}]^{-1} = \begin{bmatrix} \frac{-r}{r^2-x^2} & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} \frac{r^2-x^2}{r^2} & 0 \\ 0 & 1 \end{bmatrix} \quad (2.51)$$

$$= \begin{bmatrix} -\frac{1}{r} & 0 \\ 0 & 0 \end{bmatrix} \quad (2.52)$$

We see that $u_2 = (0, 1)$ and $u_1 = (1, 0)$ are eigenvectors for \hat{L} with respective eigenvalues $\kappa_2 = -\frac{1}{r}, \kappa_1 = 0$. Given the theorem of Olinde Rodriguez suggests that u_2 points in the direction of maximum curvature of the surface, $-\frac{1}{r}$, which is predictably in the direction directly perpendicular to the trough, whereas the direction of least curvature is along the trough and is 0. The theorem of Meusnier suggests that the normal curvature $\kappa_2 = -\frac{1}{r}$ is reasonable—any curve on the trough perpendicular to the ridge should have the curvature of a circle (the negative simply indicates that we are on the “outside” of the surface). Finally, we note that at the ridge of the trough is exactly where $\nabla f = 0$, and the Weingarten map is exactly the Hessian matrix there.

Viewing the surface in \mathbb{R}^3 , we define the Hessian Hess of the surface L at a point (x, y) on the surface as the matrix of its second partial derivatives:

$$\text{Hess}(x, y) = \begin{bmatrix} L_{xx}(x, y) & L_{xy}(x, y) \\ L_{yx}(x, y) & L_{yy}(x, y) \end{bmatrix} \quad (2.53)$$

At any point (x, y) we denote the two eigenpairs of $\text{Hess}(x, y)$ as

$$\text{Hess}u_i = \kappa_i u_i, \quad i = 1, 2 \quad (2.54)$$

where κ_i and u_i are known as the *principal curvatures* and *principal directions* of $L(x, y)$, respectively, and we label such that $|\kappa_2| \geq |\kappa_1|$. Notably, $\text{Hess}(x, y)$ is a real, symmetric matrix (since $L_{xy} = L_{yx}$ and L is a real function) and thus its eigenvalues are real and its eigenvectors are orthonormal to each other, as given by following lemma:

Lemma 2.7 (Principal Axis Theorem?). *Let A be a real, symmetric matrix. The eigenvalues of A are real and its eigenvectors are orthonormal to each other.*

Proof. Let $x \neq 0$ so that $Ax = \lambda x$. Then

$$\begin{aligned}\|Ax\|_2^2 &= \langle Ax, Ax \rangle = (Ax)^* Ax \\ &= x^* A^* Ax = x^* A^T Ax = x * AAx \\ &= x^* A \lambda x = \lambda x^* Ax \\ &= \lambda x^* \lambda x = \lambda^2 x^* x = \lambda^2 \|x\|_2^2\end{aligned}$$

Upon rearrangement, we have $\lambda^2 = \frac{\|Ax\|_2^2}{\|x\|_2^2} \geq 0 \implies \lambda$ is real.

To prove that a set of orthonormalizable eigenvectors exists, let A be real, symmetric as above and consider the eigenpairs $Av_1 = \lambda_1 v_1$, $Av_2 = \lambda_2 v_2$ with $v_1, v_2 \neq 0$.¹

In the case that $\lambda_1 \neq \lambda_2$, we have

$$\begin{aligned}(\lambda_1 - \lambda_2)v_1^T v_2 &= \lambda_1 v_1^T v_2 - \lambda_2 v_1^T v_2 \\ &= (\lambda_1 v_1)^T v_2 - v_1^T (\lambda_2 v_2) \\ &= (Av_1)^T v_2 - v_1^T (Av_2) \\ &= v_1^T A^T v_2 - v_1^T A v_2 \\ &= v_1^T A v_2 - v_1^T A v_2 = 0\end{aligned}$$

Since $\lambda_1 \neq \lambda_2$, we conclude that $v_1^T v_2 = 0$.

In the case that $\lambda_1 = \lambda_2 =: \lambda$, we can define (as in Gram-Schmidt orthogonalization)

¹To simplify notation, we simplify our argument to consider two explicit eigenvectors only, since we're only concerned with the 2×2 matrix Hess anyway.

$u = v_2 - \frac{v_1^T v_2}{v_1^T v_1} v_1$. This is an eigenvector for $\lambda = \lambda_2$, as

$$\begin{aligned} Au &= A \left(v_2 - \frac{v_1^T v_2}{v_1^T v_1} v_1 \right) \\ &= Av_2 - \frac{v_1^T v_2}{v_1^T v_1} Av_1 \\ &= \lambda v_2 - \frac{v_1^T v_2}{v_1^T v_1} \lambda v_1 \\ &= \lambda \left(v_2 - \frac{v_1^T v_2}{v_1^T v_1} v_1 \right) = \lambda u \end{aligned}$$

and is perpendicular to v_1 , since

$$\begin{aligned} v_1^T u &= v_1^T \left(v_2 - \frac{v_1^T v_2}{v_1^T v_1} v_1 \right) \\ &= v_1^T v_2 - \left(\frac{v_1^T v_2}{v_1^T v_1} \right) v_1^T v_1 \\ &= v_1^T v_2 - v_1^T v_2 (1) = 0. \end{aligned}$$

□

Thus we see that the two principal directions form an orthonormal frame at each point (x,y) within the continuous image $L(x,y)$.

We now seek to harness the ideas of this section to the task at hand: identifying curvilinear content within images.

The Frangi Filter

The Frangi filter, first described by Alejandro Frangi et al. in [2] is a widely used (cite) Hessian-based filter within image processing. Hessian-based filters make use of the logical “proximity” of the Hessian to notions of curvature of surfaces, as developed in section 2.2. Several such Hessian-based filters exist—see [12] and [9], as well as a comparison given in [11]. These filters use information about the principal curvatures, approximated as eigenvalues of the Hessian) at each point in the image to identify regions of significant curvature within an image.

Frangi's filter was originally developed for vascular segmentation in images such as MRIs and it excels in that context.

The procedure for a single scale in a 2D image is as follows: Let λ_1, λ_2 be the two eigenvalues of the Hessian of the image at point (x, y) , ordered such that $|\lambda_1| \leq |\lambda_2|$, and define the Frangi vesselness measure as:

$$V_\sigma(x_0, y_0) = \begin{cases} 0 & \text{if } \lambda_2 > 0 \\ \exp\left(-\frac{A^2}{2\beta^2}\right) \left(1 - \exp\left(-\frac{S^2}{2\gamma^2}\right)\right) & \text{otherwise} \end{cases} \quad (2.55)$$

where

$$A := |\lambda_1/\lambda_2| \quad \text{and} \quad S := \sqrt{\lambda_1^2 + \lambda_2^2} \quad (2.56)$$

and β and γ are tuning parameters. Before we discuss appropriate values for β and γ , we first seek to highlight the significance of eq. (2.55), and in particular, the ratios defined in eq. (2.56). A and S are known as the anisotropy measure and structureness measure, respectively.

Anisotropy Measure

The anisotropy (or directionality) measure A is simply the ratio of magnitudes of λ_1 and λ_2 . Since at a ridge point of a tubular structure, we should have $\lambda_1 \approx 0$ and $|\lambda_2| \gg |\lambda_1|$, a very small value of A would be present at a ridge of a tubular structure.

In fig. 3, this situation is demonstrated. Here, u_1, u_2 form the orthogonal set of Hessian eigenvectors with corresponding eigenvalues λ_1 and λ_2 . At such a ridgelike structure, we could predict the largest change in curvature to be straight down the ridge (in the direction of u_2), and the direction of least curvature to be directly along the ridge (in the direction of u_1). $\lambda_1 \approx 0$ and λ_2 is large and negative Note that the length of these vectors in this picture is not meant to represent their magnitudes, as u_2 should have a much larger relative magnitude by design!

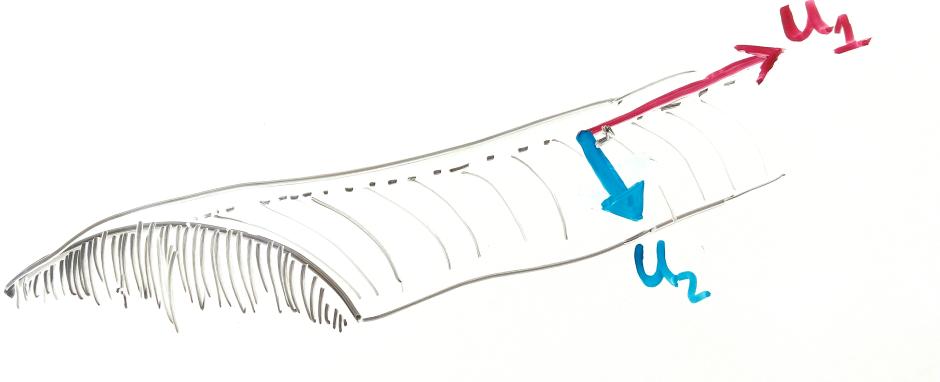


FIGURE 3: The principal eigenvectors at a ridge like structure

Of course, if the the ridge is perfectly circular along its cross section (as was in section 2.2, it is of course apparent that λ_2 would be the same value at any place along the ridge (not just at its crest), and λ_1 would likewise be 0 at any such point. One could also imagine a similar situation in which the dropoff from crest to bottom gets increasing steep. In such a case, λ_2 as a function of x would in fact be largest nearest to the bottom. This thought experiment should dispel a naive misunderstanding of the power of a Frangi filter: a high anisotropy measure (and a large structureness measure) will not in general identify the crests of a ridge-like structure—it only will highlight that such a pixel is on a ridge-like structure at all. Thus, the anisotropy measure will not necessarily be at a maximum at the crest of the ridge.

Similiarly, the vessel we we wish to identify can not be reasonably expected to behave as perfectly as our toy example. There will likely be small aberrations in a ridgelike structure, such as small divots or depressions in an overall ridge-like structure. Of importance in our data set later, there will be points where we seem to "lose" our ridgelike structure due to issues with the sample section 4.1.

Importantly, this formulation does not require λ_1 to be approximately zero, just that the

curvature in the downward direction is much more significant.

Also the crest could be really flat (“hangar shaped”), in which case both are around zero. At the crest of the ridge, we would actually expect both u_1 and u_2 to be around 0, whereas a point somewhere between the crest and the “foot” of the ridge to contain the maximum u_2 .

We will fix some of these issues by casting this as a multiscale problem in section 2.4.

Two other ideas that could fix some other discrepancies mentioned above is to identify these ridges on their own, or also where the ‘feet’ are’. We will discuss these ideas in section 6.2.

Structureness measure

There is another concern with using the pure ratio $S := |\lambda_1/\lambda_2|$ as an identifying feature of ridgelike structures apart from the ones listed above. We could still have $|\lambda_2| \gg |\lambda_1|$ in a relative sense, but still have $\lambda_2 \approx 0$. As a rather extreme example, we should certainly wish to differentiate a point on the surface where $\lambda_2 \approx 10^{-5}$ and $\lambda_1 \approx 10^{-10}$ from another point where $\lambda_2 \approx 10000$ and $\lambda_2 = 0.1$.

A natural fix to differentiate these points is to introduce a “structureness” measure to insure that there is in fact significant curvilinear activity at the point in question. Frangi used $S := \sqrt{(\lambda_1)^2 + (\lambda_2)^2}$, which is in fact the 2-norm of the Hessian matrix. Thus the Frangi filter should also prefer areas of great curvilinear content in the image first of all.

The Frangi vesselness measure

Our goal then is to attach a numerical measure to each pixel in the image (at a particular scale σ) that is large when the anisotropy measure A and the structureness measure S is sufficiently large.

The form Frangi arrived at in eq. (2.55) in which a factor of $\exp\{\dots\}$ and $(1 - \exp\{\})$ are

multiplied together are simply to ensure that the final vesselness measure V is largest when A is small and S is large enough, with rapidly decay in other situations.

Frangi further strengthened the filter by adding an additional case to in eq. (2.55), ensuring that λ_2 is not positive. If we are indeed at a curvilinear ridge, we need the second derivative of the surface in the maximal direction to be negative, which hasn't been accounted for as yet in our formulation of A and S – we wish (for our purposes) to only identify when we are finding crests. A will still be small and S will still be large however if we identify a “trough”.

The only perceivable difference is that the maximum normal curvature will be positive—we are at a local minimum in the direction of u_2 . In situations where we wish to only identify ridges (as is the case here) we simply exclude any points where there is not a negative curvature in the maximal direction.

The Frangi vesselness filter: Choosing parameters β and γ

The parameters β and γ are meant to scale so that the peaks of $\exp\{\dots\}$ and $1 - \exp\{\dots\}$ coincide enough to be statistically significant but rapidly decay in areas not associated with curvilinear structure. What values of these parameters are appropriate is ultimately dependent on the context of the problem.

Frangi suggested for γ that half of the Frobenius norm of the Hessian matrix is appropriate, simply because the minimum value of S is zero, and its maximum value is approx the 2 norm of the Hessian. For β Frangi chose an innocuous intermediate point, $\beta = 1/2$ (and thus $2\beta^2 = 1/2$). As we will show later, choosing the structureness parameter γ is rather important for the context especially if the background (non-ridgelike structure) is significant and noisy. β should be strengthened/relaxed depending on how “flat” the ridgelike structure is. If there is a lot of gain then β should be smaller. If this is not the case, a stronger filter can be created by requiring A to be much smaller.

We now take a quick tangent from our description of the Frangi filter to develop and justify our “multiscale” approach.

Linear Scale Space Theory

There is obviously a major disconnect in the ideas presented above. Although the ideas presented above require differentiation of continuous surfaces, our image is in fact a discrete pixel. That is, our previous discussions have been in terms of an image as the continuous surface in definition 2.2, rather than the more realistic discrete pixel matrix as in definition 2.1. The present section seeks to address this disconnect. In particular, we seek to mitigate the bias of our limited sampling of the “true” 3D surface. Our main goal is to counter against some of the bias of our particular sampling. In particular, we wish to not over-represent structures that are clear at our resolution without giving appropriate weight to larger structures as well. Koenderink [7] argued that ”any image can be embedded in a one-parameter family of derived images (with resolution as the parameter) in essentially only one unique way” given a few of the so-called scale space axioms. He (and others) showed that a small set of intuitive axioms imply require that any such family of images must satisfy the heat equation

$$\Delta K(x, y, \sigma) = K_\sigma(x, y, \sigma) \text{ for } \sigma \geq 0 \text{ such that } K(x, y, 0) = u_0(x, y). \quad (2.57)$$

where $K : \mathbb{R}^3 \rightarrow \mathbb{R}$ and $u_0 : \mathbb{R}^2 \rightarrow \mathbb{R}$: is the original image (viewed as a continuous surface) and σ is a resolution parameter. Much work has been done to formalize this approach [13]. There is a long list of desired properties—we will try to identify a minimal subset of axioms and show that other desired properties follow.

Axioms

To make matters manageable, we require the one-parameter family of scaled images to be generated by an operation on the original image:

$$\{ K(x, y; \sigma) = T_\sigma u_0 \mid \sigma \geq 0, K(x, y, ; 0) = u_0 \}$$

The following axioms are then requirements on what sort of operation T_σ should be.

Linear-shift and Rotational invariance

Linear-shift (or translation) invariance means that no position in the original signal is favored. This is intuitive, as our operation should apply to any image fairly, regardless of where content is found in the image. Similarly, there should be not be favoritism toward any particular orientation of content within the image.

Continuity of scale parameter

There is no reason for the scale parameter to be discrete; we may alter the resolution with whatever precision we desire. That is, we take the resolution parameter σ to be a nonzero real number (as opposed to an integer). Moreover, we require that the operator behaves continuously with respect to the scale parameter. What happens as $\sigma \downarrow 0$ is less clear though. An argument from functional analysis (see [4]) implies that there is a so-called “infinitesimal generator” A which is a limit case of our desired operator T ; that is

$$Au_0 = \lim_{\sigma \downarrow 0} \frac{T_\sigma u_0 - u_0}{\sigma} \quad (2.58)$$

and moreover that there is a resultant differential equation concerning the derivative of the family and A :

$$o\partial_\sigma K(x, y; \sigma) = \lim_{\sigma \downarrow 0} \frac{K(\cdot; \sigma + h) - K(\cdot; \sigma)}{h} = A(T_\sigma u) = A(K(\cdot, \sigma)) \quad (2.59)$$

We shall return to this idea later and more concretely describe A once characterize T_σ .

Semigroup property

The semigroup property is simply that transforming the original image by some resolution σ should have the same overall effect of two successive transformations σ_1 and σ_2 , i.e.

$$T_\sigma u = T_{\sigma_1 + \sigma_2} u \quad (2.60)$$

Causality condition

This requirement has great implication, and is also very successful in encoding our intuitive sense of “resolution”. The causality condition is the one that, as resolution decreases, no finer detail is introduced into the image. That is, as the scale increases, there will be no creation of local extrema that did not exist at a smaller scale.

In other words, if $K(x_0, y_0; \sigma_0)$ is a local maximum (at the point (x_0, y_0) , at this fixed σ_0) i.e. then an increase in scale can only weaken this peak, i.e.

$$\begin{cases} \nabla K(x_0, y_0; \sigma_0) = 0 \\ \Delta K(x_0, y_0; \sigma_0) < 0 \end{cases} \implies K(x_0, y_0; \sigma_1) \leq K(x_0, y_0; \sigma_0) \forall \sigma_1 \geq \sigma_0 \quad (2.61)$$

Similarly, if $K(x_0, y_0; \sigma_0)$ is a local minimum (with respect to space), then an increase in scale cannot make such a valley more profound, i.e.

$$\begin{cases} \nabla K(x_0, y_0; \sigma_0) = 0 \\ \Delta K(x_0, y_0; \sigma_0) > 0 \end{cases} \implies K(x_0, y_0; \sigma_1) \geq K(x_0, y_0; \sigma_0) \forall \sigma_1 \geq \sigma_0 \quad (2.62)$$

This implies that no image feature is sharpened by an decrease and resolution—the only result is a monotonic blurring of the image as scale parameter σ tends to infinity.

Uniqueness of the Gaussian Kernel

The above requirements are actually sufficient in proving not only that the operator T_σ is a convolution, but that the heat equation described in eq. (2.57) must hold. This has been

shown in various ways, both by Koenderink [7], Babaud [1], as well as Lindeberg in [13]. In fact, it is shown that the Gaussian is the unique convolution kernel that works.

[TODO: Include or cite? If we're just citing it, how much should we really have in this entire section to begin with?] To this, show that:

- a kernel satisfying the above axioms must satisfy the heat equation
- the gaussian kernel satisfies that.
- gaussian kernel is the only kernel that works.

That is,

$$K(x, y; \sigma) = T_\sigma u_0 = G_\sigma \star u_0 \quad \text{where} \quad G_\sigma := \frac{1}{2\pi\sigma^2} e^{(-|x|^2/(2\sigma^2))} \quad (2.63)$$

[TODO: remove this unless you plan to show more] We can show that this solution solves the heat equation. Given u_0 as a continuous image (unscaled), we construct PDE with this as a boundary condition.

$$u : \mathbb{R}^2 \supset \Omega \rightarrow \mathbb{R} \text{ with } u(\mathbf{x}, t) : \begin{cases} \frac{\partial u}{\partial t}(\mathbf{x}, t) = \Delta u(\mathbf{x}, t) & , t \geq 0 \\ u(\mathbf{x}, 0) = u_0(\mathbf{x}) \end{cases} \quad (2.64)$$

We show that

$$u(\mathbf{x}, t) = (G_{\sqrt{2t}} \star u_0)(\mathbf{x}) \quad (2.65)$$

solves (the above tagged equation), where

s

First, we need a quick lemma regarding differentiation a continuous convolution.

Lemma 2.8. *Derivative of a convolution is the way that it is (obviously rewrite this).*

Proof. For a single variable,

$$\frac{\partial}{\partial \alpha} [f(\alpha) \star g(\alpha)] = \frac{\partial}{\partial \alpha} \left[\int f(t)g(\alpha-t)dt \right] \quad (2.66)$$

$$= \int f(t) \frac{\partial}{\partial \alpha} [g(\alpha-t)] dt \quad (2.67)$$

$$= \int f(t) \left(\frac{\partial g}{\partial \alpha} \right) g(\alpha-t) dt \quad (2.68)$$

$$= f(\alpha) \star g'(\alpha) \quad (2.69)$$

By symmetry of convolution we can also conclude

$$\frac{\partial}{\partial \alpha} [f(\alpha) \star g(\alpha)] = f'(\alpha) \star g(\alpha)$$

If f and g are twice differentiable, we can compound this result to show a similar statement holds for second derivatives, and then, given the additivity of convolution, we may conclude

$$\Delta(f \star g) = \Delta(f) \star g = f \star \Delta(g) \quad (2.70)$$

□

Theorem 2.9. $u(\mathbf{x}, t) = (G_{\sqrt{2t}} \star u_0)(\mathbf{x})$ solves the heat equation.

Proof. We focus on the particular kernel

$$G_{\sqrt{2t}} = \frac{1}{4\pi t} e^{(-|x|^2/(4t))}$$

Then

$$\frac{\partial u}{\partial t}(\mathbf{x}, t) = \frac{\partial}{\partial t} (G_{\sqrt{2t}}(\mathbf{x}, t) \star u_0(\mathbf{x})) \quad (2.71)$$

$$= \frac{\partial}{\partial t} (G_{\sqrt{2t}}(\mathbf{x}, t)) \star u_0(\mathbf{x}) \quad (2.72)$$

$$= \frac{\partial}{\partial t} \left(\frac{1}{4\pi t} e^{(-|x|^2/(4t))} \right) \star u_0(\mathbf{x}) \quad (2.73)$$

$$= \left[-\frac{1}{4\pi t^2} e^{(-|x|^2/(4t))} + \frac{1}{4\pi t} \left(\frac{-|x|^2}{4t^2} \right) e^{-|x|^2/(4t)} \right] \star u_0(\mathbf{x}) \quad (2.74)$$

$$= -\frac{1}{4t^2} \left(e^{(-|x|^2/(4t))} + |\mathbf{x}|^2 G_{\sqrt{2t}}(\mathbf{x}, t) \right) \star u_0(\mathbf{x}) \quad (2.75)$$

and from the previous lemma,

$$\Delta u(\mathbf{x}, t) = \Delta(G_{\sqrt{2t}} \star u_0(\mathbf{x})) = \Delta(G_{\sqrt{2t}}) \star u_0(\mathbf{x})$$

We explicitly calculate the Laplacian of $G_\sigma(x, y) = A \exp(-\frac{x^2+y^2}{2\sigma^2})$ as follows:

$$\begin{aligned} \frac{\partial}{\partial x} G_\sigma(x, y) &= A \left(\frac{-2x}{2\sigma^2} \right) \exp \left(-\frac{x^2+y^2}{2\sigma^2} \right) \\ \implies \frac{\partial^2}{\partial x^2} G_\sigma(x, y) &= A \cdot \frac{\partial}{\partial x} \left[-\frac{x}{\sigma^2} \exp \left(-\frac{x^2+y^2}{2\sigma^2} \right) \right] \\ &= A \left[-\frac{1}{\sigma^2} \exp \left(-\frac{x^2+y^2}{2\sigma^2} \right) + \frac{x}{\sigma^2} \cdot \frac{2x}{2\sigma^2} \exp \left(-\frac{x^2+y^2}{2\sigma^2} \right) \right] \\ &= A \exp \left(-\frac{x^2+y^2}{2\sigma^2} \right) \left[-\frac{1}{\sigma^2} + \frac{x^2}{\sigma^4} \right] \\ &= \frac{1}{\sigma^2} G_\sigma(x, y) \left[\frac{x^2}{\sigma^2} - 1 \right] \end{aligned}$$

By symmetry of argument we also may conclude

$$\frac{\partial^2}{\partial y^2} G_\sigma(x, y) = \frac{1}{\sigma^2} G_\sigma(x, y) \left[\frac{y^2}{\sigma^2} - 1 \right]$$

and so

$$\Delta G_\sigma(x, y) = \frac{\partial^2}{\partial x^2} (G_\sigma) + \frac{\partial^2}{\partial y^2} (G_\sigma) = \frac{1}{\sigma^2} G_\sigma(x, y) \left[\frac{x^2+y^2}{\sigma^2} - 2 \right] \quad (2.76)$$

Then, given lemma 2.8, we conclude

$$\Delta [G_\sigma(x, y) \star u_0(x, y)] = \left(\frac{1}{\sigma^2} G_\sigma(x, y) \left[\frac{x^2+y^2}{\sigma^2} - 2 \right] \right) \star u_0(x, y) \quad (2.77)$$

For particular choices of $\sigma(t) = \sqrt{2t}$ and $A = \frac{1}{4\pi t}$, we see

$$\Delta [G_{\sqrt{2t}}(x, y) \star u_0(x, y)] = \left(\frac{1}{2t} G_{\sqrt{2t}}(x, y) \left[\frac{x^2+y^2}{2t} - 2 \right] \right) \star u_0(x, y) \quad (2.78)$$

$$= \left(G_{\sqrt{2t}}(x, y) \left[\frac{x^2+y^2}{4t^2} - \frac{1}{t} \right] \right) \star u_0(x, y) \quad (2.79)$$

We then calculate the time derivative, using our particular choice of $\sigma(t) = \sqrt{2t}$ and $A = \frac{1}{4\pi t}$ as:

$$\frac{\partial}{\partial t} [G_{\sigma(t)}(x, y) * u_0(x, y)] = \frac{\partial}{\partial t} [G_{\sigma(t)}(x, y)] * u_0(x, y) \quad (2.80)$$

$$= \frac{\partial}{\partial t} [G_{\sqrt{2t}}(x, y)] * u_0(x, y) \quad (2.81)$$

$$= \frac{\partial}{\partial t} \left[\frac{1}{4\pi t} \exp\left(-\frac{x^2+y^2}{4t}\right) \right] * u_0(x, y) \quad (2.82)$$

$$= \left[-\frac{1}{4\pi t^2} \exp\left(-\frac{x^2+y^2}{4t}\right) + \frac{1}{4\pi t} \left(\frac{x^2+y^2}{4t^2} \exp\left(-\frac{x^2+y^2}{4t}\right) \right) \right] * u_0(x, y) \quad (2.83)$$

$$= \left(G_{\sqrt{2t}}(x, y) \left[\frac{x^2+y^2}{4t^2} - \frac{1}{t} \right] \right) * u_0(x, y) \quad (2.84)$$

Combining these results, we find that

$$\frac{\partial}{\partial t} [G_{\sqrt{2t}} * u_0] = \Delta [G_{\sqrt{2t}} * u_0] \quad (2.85)$$

as desired. \square

We now return to our discussion of the Frangi filter.

A multiscale approach

Our ideas of scale developed in the previous section imply that, if the ridgelike structures we wish to detect are more prominent at different scales, then a multiscale approach is the natural one. Considering our developments in section 2.3, we wish to probe at multiple scales regions that would receive a high vesselness score at any range, and consider them all together. Frangi [2] approached this problem by simply aggregating vesselness measure over all scales:

$$V(x_0, y_0) = \max_{\sigma \in \Sigma} V_\sigma(x_0, y_0) \quad (2.86)$$

where $\Sigma := \{\sigma_0, \sigma_1, \dots, \sigma_N\}$ is a range of parameters at which to probe. These should be chosen to be representative enough of all scales where meaningful content is expected to be found.

Thresholding

After this procedure, we are left with a matrix with as many samples/pixels as the original image, all with a vesselness measure between 0 and 1 for each pixel in the image:

$$V_\Sigma := [V(x, y)]_{\substack{0 \leq x < M \\ 0 \leq y < N}} \quad (2.87)$$

Whereas Frangi [2] left relatively open-ended, we may wish to be final about the whole matter and ultimately say whether or not a pixel does in fact corresponds to a curvilinear structure. There are multiple methods of doing so. A straightforward enough approach is to simply threshold past some certain point. The resulting matrix can be given in terms of either eq. (2.86) or eq. (2.87)

$$V_{\Sigma, \alpha}(x, y) = \begin{cases} 1 & \text{if } V(x, y) \geq \alpha \\ 0 & \text{else} \end{cases}, \quad \alpha > 0 \text{ for } \alpha \text{ fixed.} \quad (2.88)$$

We will discuss alternatives methods of aggregating results from our multiscale method, as well as optimal values for parameters and scales in chapter 3.

First and Second Derivatives of Discrete Images

The way that you'd normally/naively describe it. Non multiscale. As in scipy or whatever.

Equivalent Methods of Calculating 2D Hessian

Note the following 6 methods should all theoretically be equivalent:

- convolve discrete image with a gaussian kernel, then take derivatives (no FFT). This is the “classical approach”

- Take derivatives of gaussian kernel, then convolve with the image/signal (again, no FFT)(due to)
- FFT image and FFT gaussian kernel then convolve in freq. space, then IFFT, then take derivatives in xy-space (my implementation)
- take derivatives of gaussian kernel, then FFT it, FFT the image, then colvolve in frequency space, then IFFT (slower due to size of large kernels)
- FFT image, then multiply by theoretical gaussian kernel in freq. space, then IFFT, then take derivatives in xy-space.
- FFT image, then multiply by theoretical gaussian kernel in freq space, then take derivatives in freq space, then IFFT.

[**TODO: big implementation note: you want to show equivalence of these, but the problem is that around the boundary the issue is annoying. can you fix this?**]

Implementation Details: Convolution Speedup via FFT

As described above, the actual computation of derivatives is achieved via convolution with a gaussian. In practice, this is very slow for large scales. We instead, we perform a fast Fourier transform, which offers a speedup of N^2 operations vs $\mathcal{O}(n \cdot \log_2 n)$ operations

Fourier Transforms

Fourier Transform of a continuous 1D signal

A periodic signal (real valued function) $f(t)$ of period T can be expanded in an infinite basis as follows:

$$f(t) = \sum_{-\infty}^{\infty} c_n e^{i \frac{2\pi n}{T} t}, \quad c_n = \frac{1}{T} \int_{-T/2}^{T/2} f(t) e^{-i \frac{2\pi n}{T} t} dt \quad (2.89)$$

The Fourier transform of a 1D continuous function is defined by

$$F(\mu) := \mathcal{F}\{f(t)\} = \int_{-\infty}^{\infty} f(t) e^{i 2\pi \mu t} dt \quad (2.90)$$

It can be shown that an inverse transform will then recover our original signal:

$$f(t) = \mathcal{F}^{-1}\{F(\mu)\} = \int_{-\infty}^{\infty} F(\mu) e^{i 2\pi \mu t} dt \quad (2.91)$$

Together, eq. (2.90) and eq. (2.91) are referred to as the *Fourier transform pair* of the signal $f(t)$.

Fourier Transform of a Discrete 1D signal

We wish to develop the Fourier transform pair for a discrete signal. We frame the situation as follows: A continuous function $f(t)$ is represented as the sampled function $\tilde{f}(t)$ by multiplying it by a sampling function (also referred to in DIP-GW as an impulse function), an infinite series of discrete impulses with equal spacing ΔT :

$$s_{\Delta T}(t) = \sum_{n=-\infty}^{\infty} \delta[t - n\Delta T], \quad \delta[t] = \begin{cases} 1, & t = 0 \\ 0, & t \neq 0 \end{cases} \quad (2.92)$$

where $\delta[t]$ is the discrete unit impulse.

The discrete sample $f(t)$ is then constructed from $f(t)$ by

$$\tilde{f}(t) = f(t)s_{\Delta T}(t) \quad (2.93)$$

From this (and actually the convolution theorem) we can calculate $\tilde{F}(t)$. [TODO: Derive this. There's some groundwork here to copy between 213-220. We don't need the sampling theorem, since our signal is given as-is... i think].

1D Discrete Fourier Transform (DFT)

Given the discrete signal \tilde{f} , we construct the transform $\tilde{F}(\mu) = \mathcal{F}\{\tilde{f}(t)\}$. by simply transforming the definition eq. (2.93).

$$\tilde{F}(\mu) = \sum_{n=-\infty}^{\infty} f_n e^{-i2\pi\mu n \Delta T}, \quad f_n = \tilde{f}(n) = f(n\Delta T) \quad (2.94)$$

The transform is a continuous function with period $1/\Delta T$.

- [TODO: find image processing papers that find hessian from FFT / who uses this?]
- [TODO: with above: downsides?]
- [TODO: side by side comparison in a toy example and/or a real problem?]

2D Discrete Fourier Transform Convolution Theorem

Theorem 2.10 (2D DFT Convolution Theorem). *Given two discrete functions are sequences with the same length [TODO: don't gloss over this] , that is: $f(x,y)$ and $h(x,y)$ for integers $0 < x < M$ and $0 < y < N$, we can take the discrete fourier transform (DFT) of each:*

$$F(u,v) := \mathcal{D}\{f(x,y)\} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x,y) e^{-2\pi i (\frac{ux}{M} + \frac{vy}{N})} \quad (2.95)$$

$$H(u,v) := \mathcal{D}\{h(x,y)\} = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} h(x,y) e^{-2\pi i (\frac{ux}{M} + \frac{vy}{N})} \quad (2.96)$$

and given the convolution of the two functions

$$(f \star h)(x,y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m,n) h(x-m, y-n) \quad (2.97)$$

then $(f \star h)(x,y)$ and $MN \cdot F(u,v)H(u,v)$ are transform pairs, i.e.

$$(f \star h)(x,y) = \mathcal{D}^{-1}\{MN \cdot F(u,v)H(u,v)\} \quad (2.98)$$

The proof follows from the definition of convolution, substituting in the inverse-DFT of f and h , and then rearrangement of finite sums.

Proof.

$$(f \star h)(x, y) = \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} f(m, n)h(x-m, y-n) \quad (2.99)$$

$$= \sum_{m=0}^{M-1} \sum_{n=0}^{N-1} \left(\sum_{p=0}^{M-1} \sum_{q=0}^{N-1} F(p, q) e^{2\pi i \left(\frac{mp}{M} + \frac{nq}{N} \right)} \right) \left(\sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H(u, v) e^{2\pi i \left(\frac{u(x-m)}{M} + \frac{v(y-n)}{N} \right)} \right) \quad (2.100)$$

$$= \left(\sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H(u, v) e^{2\pi i \left(\frac{ux}{M} + \frac{vy}{N} \right)} \right) \left(\sum_{p=0}^{M-1} \sum_{q=0}^{N-1} F(p, q) \left(\sum_{m=0}^{M-1} e^{2\pi i \left(\frac{m(p-u)}{M} \right)} \right) \left(\sum_{n=0}^{N-1} e^{2\pi i \left(\frac{n(q-v)}{N} \right)} \right) \right) \quad (2.101)$$

$$= \left(\sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H(u, v) e^{2\pi i \left(\frac{ux}{M} + \frac{vy}{N} \right)} \right) \left(\sum_{p=0}^{M-1} \sum_{q=0}^{N-1} F(p, q) (M \cdot \hat{\delta}_M(p-u)) (N \cdot \hat{\delta}_N(q-v)) \right) \quad (2.102)$$

$$= \left(\sum_{u=0}^{M-1} \sum_{v=0}^{N-1} H(u, v) e^{2\pi i \left(\frac{ux}{M} + \frac{vy}{N} \right)} \right) \cdot M N F(u, v) \quad (2.103)$$

$$= M N \cdot \sum_{u=0}^{M-1} \sum_{v=0}^{N-1} F(u, v) H(u, v) e^{2\pi i \left(\frac{ux}{M} + \frac{vy}{N} \right)} \quad (2.104)$$

$$= M N \cdot \mathcal{D}^{-1} \{ F H \} \quad (2.105)$$

where

$$\hat{\delta}_N(k) = \begin{cases} 1 & \text{when } k = 0 \pmod{N} \\ 0 & \text{else} \end{cases} \quad (2.106)$$

□

Above, we make use of the following lemma

Lemma 2.11. *Let j and k be integers and let N be a positive integer. Then*

$$\sum_{n=0}^{N-1} e^{2\pi i \left(\frac{n(j-k)}{N} \right)} = N \cdot \hat{\delta}_N(j-k) \quad (2.107)$$

Proof. Consider the complex number $e^{2\pi i(j-k)/N}$. Note first that this is an N -th root of unity, since

$$\left(e^{2\pi i(j-k)/N}\right)^N = e^{2\pi i(j-k)} = \left(e^{2\pi i}\right)^{(j-k)} = 1^{(j-k)} = 1$$

In other words, $e^{2\pi i n(j-k)/N}$ is a root of $z^N - 1 = 0$, which we can factor as

$$z^N - 1 = (z - 1)(z^{n-1} + \cdots + z + 1) = (z - 1) \sum_{n=0}^{N-1} z^n. \quad (2.108)$$

thus giving us

$$0 = \left(e^{2\pi i(j-k)/N} - 1\right) \sum_{n=0}^{N-1} e^{2\pi i n(j-k)/N} \quad (2.109)$$

To prove the claim in eq. (2.107), we consider two cases: First, if $j - k$ is a multiple of N , we of course have $e^{2\pi i n(j-k)/N} = \left(e^{2\pi i}\right)^{n(j-k)/N} = 1$ and thus the left side of eq. (2.107) reduces to

$$\sum_{n=0}^{N-1} \left(e^{2\pi i}\right)^{n(j-k)/N} = \sum_{n=0}^{N-1} (1) = N$$

In the case that $j - k$ is *not* a multiple of N , we refer to eq. (2.109). The first factor is not zero since, $\left(e^{2\pi i(j-k)/N}\right) \neq 1$ (simply since $(j - k)/N$ is not an integer), and thus it must be that the second factor is 0:

$$\sum_{n=0}^{N-1} \left(e^{2\pi i(j-k)/N}\right)^n = 0$$

We can combine these two cases by invoking the definition of eq. (2.106), giving us the result. \square

FFT

As noted, the above result applies to the Discrete Fourier Transform. As noted, we actually achieve a convolution speedup using a Fast Fourier Transform (FFT) instead. We follow the developments of DIP-GW . For clarity, we present the following theorems which allow a framework to calculate a 2D Fourier transforms quickly.

First, a 2D DFT may actually be calculated via two successive 1D DFTs, which can be seen through a basic rearrangement, as follows:

$$F(\mu, \nu) = \sum_{x=0}^{M-1} \sum_{y=0}^{N-1} f(x, y) e^{-i2\pi(\mu x/M + \nu y/N)} \quad (2.110)$$

$$= \sum_{x=0}^{M-1} e^{-i2\pi\mu x/M} \left[\sum_{y=0}^{N-1} f(x, y) e^{-i2\pi\nu y/N} \right] \quad (2.111)$$

$$= \sum_{x=0}^{M-1} e^{-i2\pi\mu x/M} \mathcal{F}_x\{f(x, y)\} \quad (2.112)$$

$$= \mathcal{F}_y\{\mathcal{F}_x\{f(x, y)\}\} \quad (2.113)$$

where $\mathcal{F}_{x'}$ refers to the 1D discrete Fourier transform of the function with respect to the variable x' only.

Thus, to calculate the fourier transform $F(u, v)$ at the point u, v requires the computation of the transform of length N for each iterated point $x \in 0..M - 1$. Thus there are MN complex multiplications and $(M - 1)(N - 1)$ complex additions in this sequence required for each point u, v that needs to be calculated. Overall, for all points that need to be calculated, the total order of calculations is on the order of $(MN)^2$. (Note: the values of $e^{-i2\pi m/n}$ can be provided by a lookup table rather than ad-hoc calculation .

We now show that a considerable speedup can be achieved through elimination of redundant calculations. In particular, we wish to show that the calculation of a 1D DFT of signal length $M = 2^n, n \in \mathbb{Z}_+$ can be reduced to calculating two half-length transforms and an additional $M/2 = 2^{n-1}$ calculations.

To "simplify" our notation we will use a new notation for the Fourier kernels/basis functions. Let the 1D Fourier transform be given by

$$F(u) = \sum_{x=0}^{M-1} f(x) W_M^{ux}, \quad \text{where } W_m := e^{-i2\pi/m} \quad (2.114)$$

We'll define $K \in \mathbb{Z}_+ : 2K = M = 2^n$ (i.e. $K = 2^{n-1}$).

We use this to rewrite the series in eq. (2.114) and split it into odd and even entries in the summation

$$F(u) = \sum_{x=0}^{2K-1} f(x) W_{2K}^{ux} \quad (2.115)$$

$$= \sum_{x=0}^{K-1} f(2x) W_{2K}^{u(2x)} + \sum_{x=0}^{K-1} f(2x+1) W_{2K}^{u(2x+1)} \quad (2.116)$$

We'll get a few identities out of the way (where $m, n, x \in \mathbb{Z}_+$ arbitrary).

$$W_{(2m)}^{(2n)} = e^{\frac{-i2\pi(2m)}{2m}} = e^{\frac{-i2\pi m}{n}} = W_m^n \quad (2.117)$$

$$W_m^{(u+m)x} = e^{\frac{-i2\pi(u+m)x}{m}} = e^{\frac{-i2\pi unx}{m}} e^{\frac{-i2\pi mx}{m}} = e^{\frac{-i2\pi ux}{m}} (1) = W_m^{ux} \quad (2.118)$$

$$W_{2m}^{(u+m)} = e^{\frac{-i2\pi(u+m)}{2m}} = e^{\frac{-i2\pi ux}{2m}} e^{-i\pi} = W_{2m}^u e^{-i\pi} = -W_{2m}^u \quad (2.119)$$

Thus we can rewrite eq. (2.116) as

$$F(u) = \sum_{x=0}^{K-1} f(2x) W_{2K}^{2ux} + \sum_{x=0}^{K-1} f(2x+1) W_{2K}^{2ux} W_{2K}^u \quad (2.120)$$

$$\implies F(u) = \left(\sum_{x=0}^{K-1} f(2x) W_K^{ux} \right) + \left(\sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \right) W_{2K}^u \quad (2.121)$$

The major advance comes via using the identities eq. (2.117) to consider the Fourier transform K frequencies later :

$$F(u+K) = \left(\sum_{x=0}^{K-1} f(2x) W_K^{(u+K)x} \right) + \left(\sum_{x=0}^{K-1} f(2x+1) W_K^{(u+K)x} \right) W_{2K}^{(u+K)} \quad (2.122)$$

$$\implies F(u+K) = \left(\sum_{x=0}^{K-1} f(2x) W_K^{ux} \right) - \left(\sum_{x=0}^{K-1} f(2x+1) W_K^{ux} \right) W_K^u \quad (2.123)$$

Comparing eq. (2.121) and eq. (2.123), we see that the expressions within parentheses are identical. What's more, these parentetical expressions are functionally identical to discrete

fourier transforms themselves! Let's notate them as follows:

$$\mathcal{D}_u\{f_{\text{even}}(t)\} := \sum_{x=0}^{K-1} f(2x)W_K^{ux} \quad (2.124)$$

$$\mathcal{D}_u\{f_{\text{odd}}(t)\} := \sum_{x=0}^{K-1} f(2x+1)W_K^{ux} \quad (2.125)$$

If we're calculating an M point transform [TODO: vocabulary also how many frequencies do we calculate? same as # samples? what do we need?] (i.e. we're wishing to calculate $F(1), \dots, F(M)$), once we've calculated the first K discrete frequencies (i.e. $F(1), \dots, F(K)$) we may simply reuse the two values we've calculated in eq. (2.124) to calculate the next $F(K+1), \dots, F(K+K) = F(M)$. Since each expression in parentheses involves K complex multiplications and $K-1$ complex additions, we are effectively saving $K(2K-1)$ calculations in computing the entire spectrum $F(1), \dots, F(M)$. When M is large, the payoff is undeniable.

In fact, through counting calculations and then doing a proof by induction, we can show that the effective number of calculations is given by $M \log_2 M$. [TODO: finish this].

Of course, since eq. (2.124) are DFTs themselves, there's nothing stopping us from reiterating this procedure; if M is substantially large, we can just as easily repeat this process a few times.

Of course, our development was for $1D$. We can extend this to $2D$ by taking note of

The one caveat is that the above development was for transforming sequences whose lengths are perfect powers of 2. Since our inputs have no reason to be this, we need to adjust for this. The explanation is [TODO: probably] that you just do the part that's a power of 2 and then do the rest manually or pick a different power. See [TODO: find out how fft-pack does this]

The inverse DFT is actually a DFT of a modified function, see [TODO: cite DIP-GW here or show that]. [TODO: check how fftpack does this]

Alternative formulations of Scale Space Theory

Mention discrete treatments, mention relaxation of some of the dubious axioms.

Morphology

Methods of merging multiscale methods.

CHAPTER 3

IMPLEMENTATIONS

Calculating the Hessian

There are 6 ways to do this in theory.

To compare, standard gaussian calculation is implemented by `scipy.ndimage.filters.gaussian_filter`

Scale Range detection

Find smallest and largest thing ...

Morphology: Plate Eroding

...

Morphology/Skeletonization

...

CHAPTER 4

RESEARCH PROTOCOL

Samples / Image Domain

We ultimately perform a PCSVN extraction on a set of 174 color placental images from a private database called NCS (from NYMH?). These are project files in GIMP which contain multiple layers. The layers together give a hand tracing of the vascular network and perimeter. A sample of overlaid layers in a representative sample is given in section 4.1

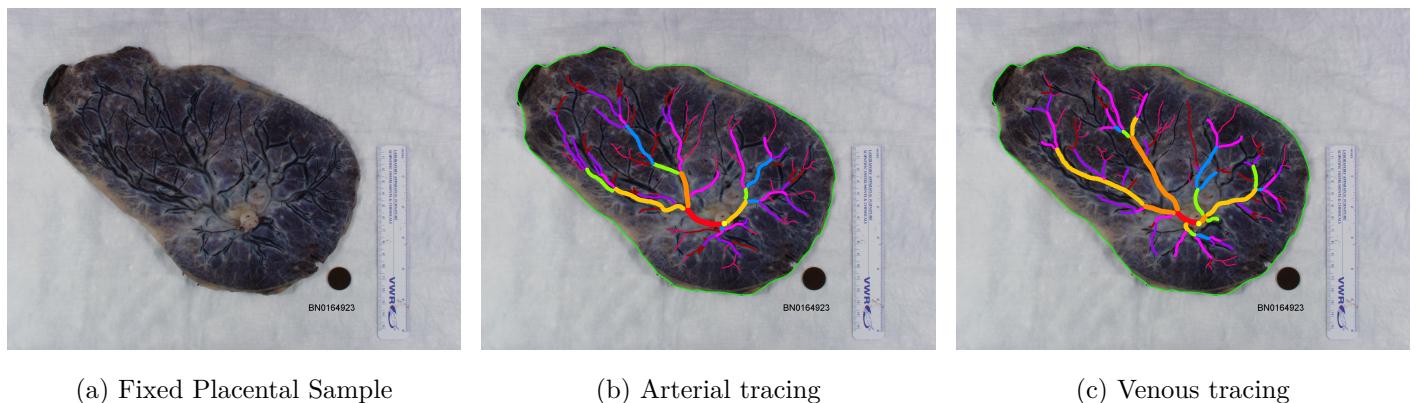


FIGURE 4: A representative placental NCS sample with vascular tracing

In fig. 4a, a cleaned, fixed placenta is shown. A detailed description of this procedure is given in [TODO: some reference]. fig. 4b and fig. 4c are both hand traces of the PCSVN, with a layer for each the arteries and veins. In our particular use case, there is no need to consider them separately, so we simply consider them together, as in fig. 4d. The coloration is meant to indicate the diameter of each vessel. There is also a cord insertion point notated, as well as the perimeter of the placental plate. These are hand-traced and rather labor intensive. A closer look at many of the samples often reveals some subjectivity in the tracings (often it's hard to see where the vein is, vascular networks are obscured, etc.)

For our procedure, we simply operate on the placental sample itself, without any un-



(a) Background Mask

(b) Sample with BG removed

(c) Grayscale

FIGURE 5: Preprocessed files from an NCS sample

derstanding of its provided tracing except for comparing the strength of our algorithm. Of course, our goal is to develop an algorithm that can produce a “ground truth” tracing such as in fig. 4d or fig. 5d without any intervention.

For our purposes however, we will use the provided placental perimeter (shown in green in section 4.1. In developing a fully automated algorithm, it would be relatively straightforward to obtain this boundary ourselves using an Active Contour Model [TODO: REF] or perhaps even any edge finding algorithm followed by morphological / watershedding as in [TODO: REF].

To build a sample suitable for use in our algorithm from section 4.1 is relatively simple. We “zero” outside the boundary of the plate (so as to not waste computational time calculating the differential geometry of a ruler, say), and also generate a binary mask to identify the plate. Finally, our vessel layers are combined and given as a binary trace.

These procedures are performed automatically on the 174 image in our data set using a custom GIMP plug-in, which performs various “bucket fill” operations, layer mergings, and thresholdings. For completeness sake, this plug-in (and an associated Scheme script which turns it into a batch operation) can be found in the Appendix. [TODO: put a link

[here\]](#) (There are actually 201 images but 27 of them have mislabeled layers and were not autoprocessed correctly)

Image Preprocessing

As a point of technicality, the grayscale image in fig. 5c is not actually produced directly by the extractor plug-in, but created when the 3 channel RGB image fig. 5b is imported at the start of the algorithm. This grayscale conversion is simply done for ease of analysis on the sample: although the Frangi filter is designed for arbitrary dimension input [?], an image with three color channels does not have 3 spatial dimensions. We therefore simply combine the information in three channels using the well-known and oft-implemented ITU-R 601-2 luma [6], or “luminance” transform:

$$L = \frac{299}{1000} R + \frac{587}{1000} G + \frac{114}{1000} B \quad (4.1)$$

[‘] All images are grayscale, M, N pixels as a masked array (of type `numpy.ma.MaskedArray`), where pixels outside of the placental region are masked so they will not be considered by the algorithm. However, some standard implementations of algorithms, namely `numpy.gradient` and `scipy.signal.convolve2d` are not designed to handle masked regions. Although it would be of some interest to create an algorithm that, say, calculates a gradient or performs a convolution by a “reflection” across an arbitrary closed boundary (as opposed to the edge of the image matrix), we opted instead to simply exclude affected areas from consideration, and zero unwanted background pixels to speed up computation. This excluding function, `plate_morphology.dilate_plate`, ultimately relies on two functions provided by the Python library `scikit-image` [14]. The first, `skimage.segmentation.find_boundaries()`, takes the mask input (such as fig. 5a) and calculates where differences in a morphological erosion and dilation occur. That boundary itself is then dilated by the desired factor. The second is a “sparse” implementation of binary dilation that is particularly efficient for our

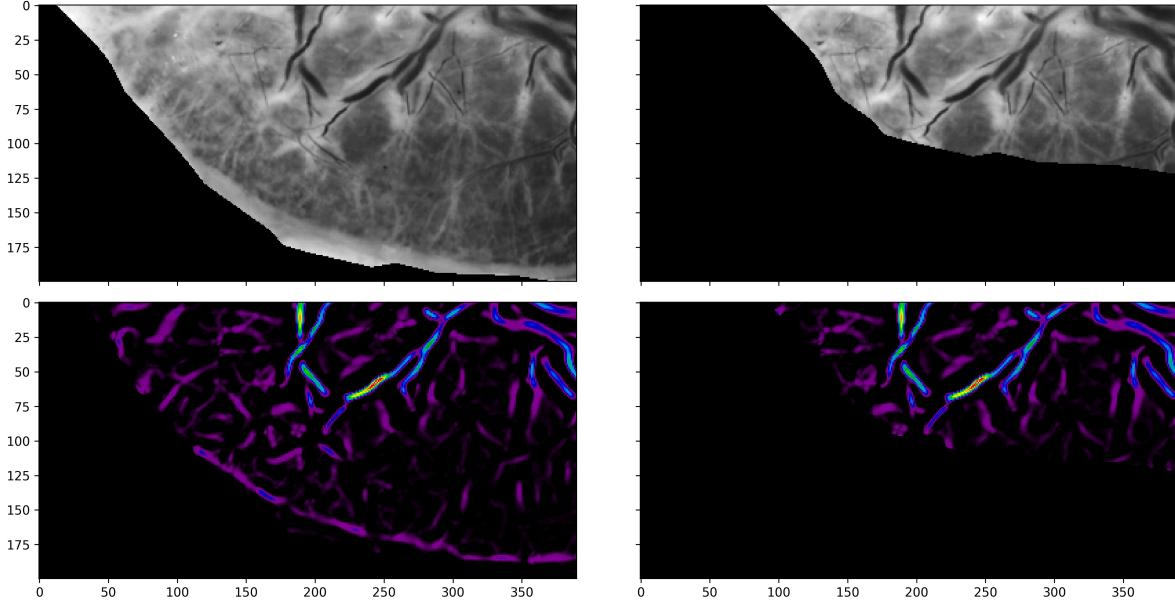


FIGURE 6: Demonstration of boundary dilation

problem. An array of indices of the image where the section 4.2 doesn't really show what I want it to, but this is what it would look like. Repeat with a smaller border. Maybe the issue doesn't occur with these as much? In the image above, $\sigma = 3$ and border radius is 80 and all it does is get rid of the stupid natural boundary, not a weird frangi response. Which is important in itself, but I was having an error just between the edge of the image.

The code for the above can be found by running `plate_morphology.py` as a top-level script (that is, within the “`if __name__ == __main__`” block of the file).

Multiscale Setup

Our multiscale Frangi filter requires a list of scales at which to probe. Each scale is chosen to accentuate features of a particular size, i.e. vessels of a particular radius. This list of scales is denoted as $\Sigma := \{\sigma_1, \sigma_2, \dots, \sigma_N\}$.

The smallest one should be an effective size where details are expected to be found, and

the largest should be an effective size as well. In fact, following [7] it is reasonable and natural to select these logarithmically; that is, for some selected inputs $m < M$ we have

$$\sigma_1 = 2^m, \sigma_j = 2^{(m + \frac{M-m}{N-1}j)}, \sigma_N = 2^M \quad (4.2)$$

That is, the exponents are spaced linearly from m to M . This is achieved by the command `np.logspace(m, M, num=N)`. The idea is that the filter will respond better at its particular scale, but there are diminishing returns as σ increases. While the filter's response may vary substantially between, say $\sigma = 2$ and $\sigma = 3$, there will be not be a substantial difference in response between, say, $\sigma = 46$ and $\sigma = 47$. There was an earlier benefit as well, that is still worth mentioning for historical reasons. Previously, computing the vesselness measure was very expensive, and thus it was simply not feasible to collect so many large scale readings. This is moot with the development of FFT-based Frangi filter.

If there is no particular care taken in selecting a minimum and maximum range at which to probe, then we should assure that there is no noise being introduced at either ends, especially if the Frangi filter at which “throw out” bad ones somehow. We will approach this issue in our discussion of “variable thresholding.”

Convolve this via fft transform to get L_{σ_i}

Applying Vesselness Measure

Calculate the Hessian matrix of and then the eigenvalues using the function `hfft.fft_hessian`.

Scale-space post-processing

Multiscale Merging

Cleanup/Postprocessing

Measurements

NOTE DUMP

Erode plate / dilate boundary

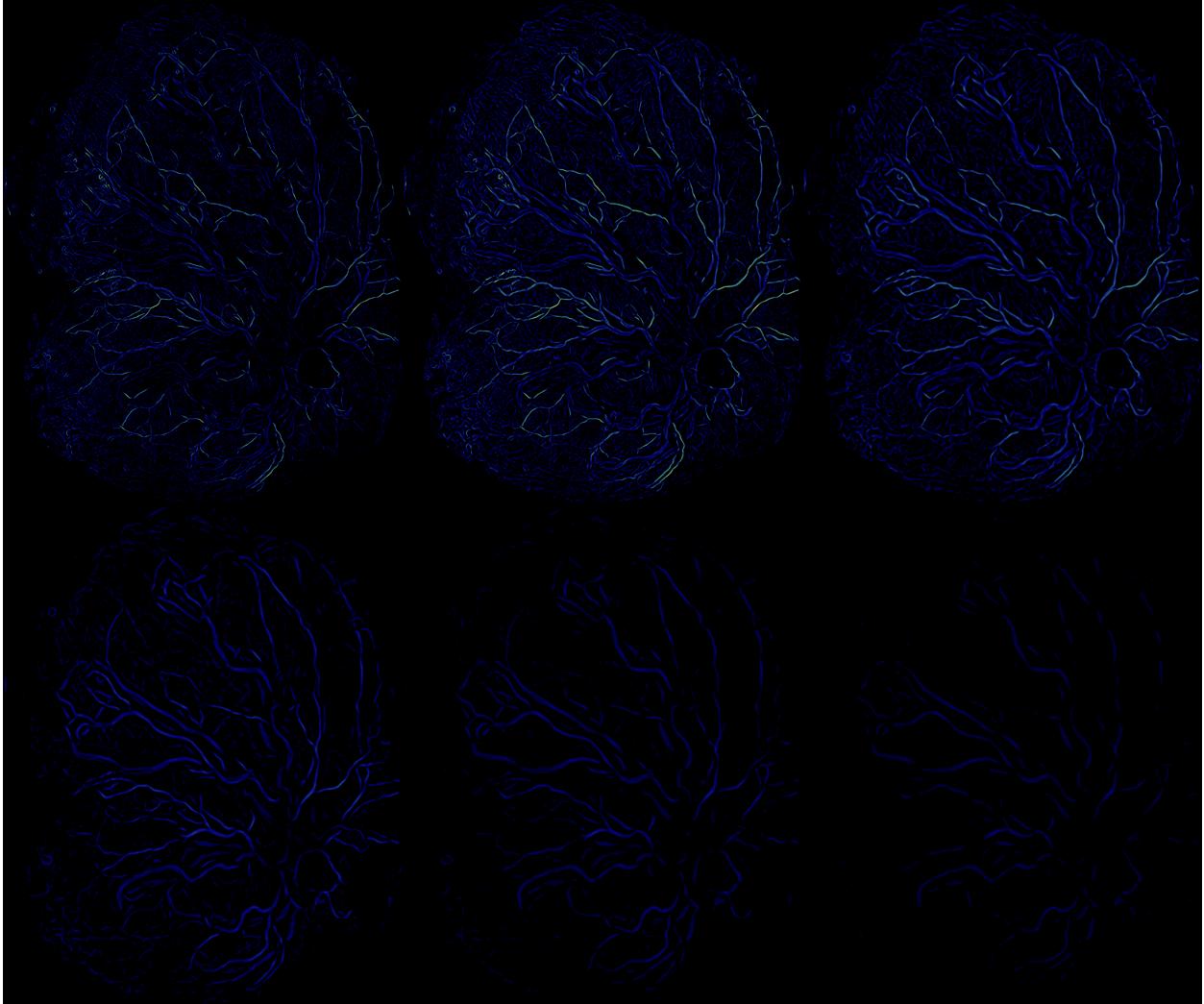


FIGURE 7: Frangi vesselness score at several scales

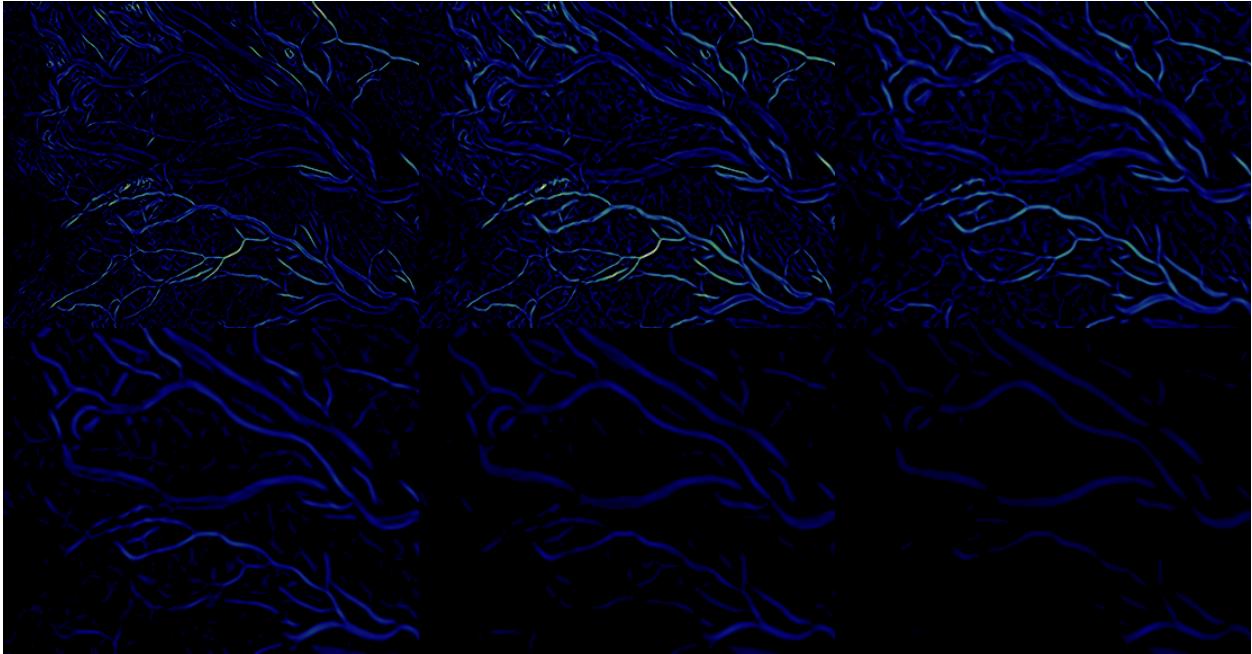


FIGURE 8: Frangi vesselness score at several scales (inset)

Our function considers the placenta as a nonzero surface but the surface outside is zero (or, in many situations, masked). We're currently not implementing any way to "reflect" along the border, so instead the second degree behavior of the surface there will be incorrect in an area proportional to the scale size.

Describe how that function works. Earlier efforts are wrong, whatever.

The area which is affected should be larger than just the standard dropoff of the gaussian however, since we're interested in second derivative information.

CHAPTER 5

RESULTS AND ANALYSIS

use MCC [10]

Sample visual output

The confusion matrix

A Source of “False Negatives” in the NCS data set

Sometimes the output doesn’t agree with the trace, i.e. “the ground truth” is not 100% correct. sometimes either there’s a false negative (reported) but something just wasn’t traced in the original 1602443.

1. Collar is stupid and should really be considered like a error in marking the perimeter. Throw these away or edit. Maybe make a section called discarded samples that’s stupid but yeah.
2. Vessels suck sometimes. In the portion above, 1602443, there’s a random blood clot which gets identified at large σ . But also the small forked shaped thing which is obviously a vessel doesn’t get defined.
3. Too much blood (not enough?? no idea) is left in the vessels. leading to the weird white border around some vessels. you could identify these along with black center and combine them somehow. no idea. Also, holy shit, some of the white vessel “sleeves” ARE identified in the tracing, and some aren’t. Find an example of this and whine about it.
4. Umbilical cord insertion point is stupid and obscures a lot. The tracer guesses but there’s no real guiding principle AFAIK..

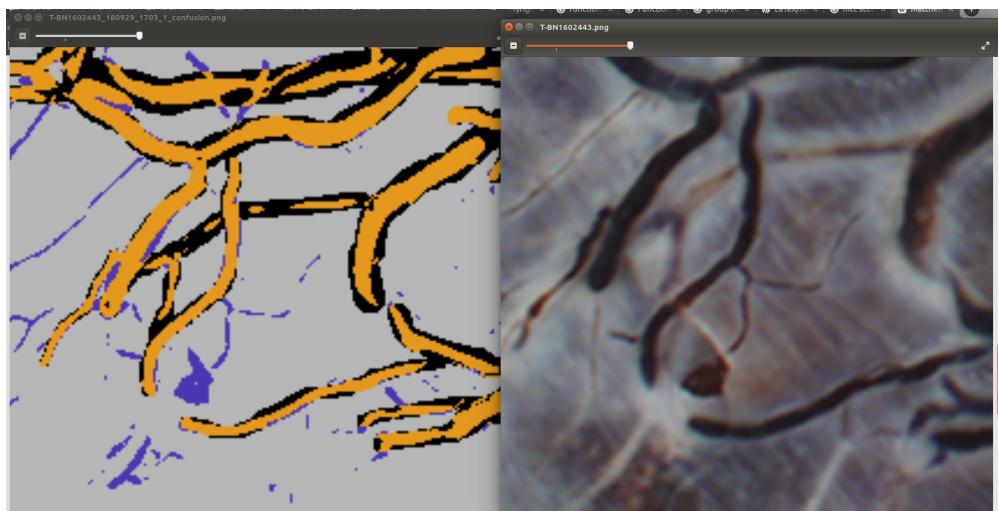


FIGURE 9: "True" false positives and "False" false positives

5. Small vessels aren't accounted for at all. Not sure how to coincide measurement in terms of scale space anymore, but should figure out how to cut off those values before running MCC metric.

Results

Answer Research Questions



FIGURE 10: Luminance/brightness thresholding is bad.

CHAPTER 6

CONCLUSION

Brief recap.

Review of Work

Estimation of success. Areas of success and struggle.

Future research directions

- Solve the Network Connection Problem (PICTURE OF GAPS)
- Refine variable thresholding and automate.
- Combine with a ridge search.
- Use this as pre-processing for a Neural Network or something.
- Apply to more image domains (STARE, WORSE PLACENTAS, ETC.)
- Automate Measurements (more quantitative results too)
- Optimize; Better Use of Scales
- Use of Color Data

APPENDICES

APPENDIX A
CODE LISTINGS

The following python scripts and modules were developed with the following packages:

- `python 3.6`
- `numpy, version 1.12.0`
- `scipy, version 0.19.0`
- `scikit-image, version 0.13.0`
- `matplotlib, version 2.02`

Earlier versions of these packages may be compatible but are not guaranteed to be so. The scripts listed in this appendix are also hosted at github.com/wukm/pycake.

listings/pcsvn.py

```
1 #!/usr/bin/env python3

3 from get_placenta import get_named_placenta
4 from score import compare_trace
5 from hfft import fft_hessian
6 from diffgeo import principal_curvatures, principal_directions
7 from frangi import get_frangi_targets
8 import numpy as np
9 import numpy.ma as ma

11 from skimage.morphology import label, skeletonize
13 from plate_morphology import dilate_boundary

15 import matplotlib.pyplot as plt
16 import matplotlib as mpl
17
18 import os.path
19 import json
```

```

import datetime
21
from get_placenta import cropped_args
23
24
def make_multiscale(img, scales, betas, gammas, find_principal_directions=False,
25                      dilate=True, dark_bg=True, VERBOSE=True):
    """returns an ordered list of dictionaries for each scale
27
    multiscale.append(
        {'sigma': sigma,
29
         'beta': beta,
30
         'gamma': gamma,
31
         'H': hesh,
32
         'F': targets,
33
         'k1': k1,
34
         'k2': k2,
35
         't1': t1,
36
         't2': t2
37     }
38
    """
39
# store results of each scale (create as empty list)
40
multiscale = list()
41
42
img = img / 255.
43
44
45
for i, sigma, beta, gamma in zip(range(len(scales)), scales, betas, gammas):
    if dilate:
        if sigma < 2.5:
            radius = 10
47
        else:
            radius = int(sigma*4) # a little aggressive
48
    else:
        radius = None
49
50
    if VERBOSE:
51
        print('σ={}'.format(sigma))
52
53
# get hessian components at each pixel as a triplet (Lxx, Lxy, Lyy)
54
hesh = fft_hessian(img, sigma)
55

```

```

59
60     if VERBOSE:
61         print('finding principal curvatures')
62
63     k1, k2 = principal_curvatures(img, sigma=sigma, H=hesh)
64
65     # area of influence to zero out
66     if dilate:
67         collar = dilate_boundary(None, radius=radius, mask=img.mask)
68
69     k1[collar] = 0
70     k2[collar] = 0
71
72
73     # set anisotropy parameter if not specified
74     if gamma is None:
75         # Frangi suggested 'half the max Hessian norm' as an empirical
76         # half the max spectral radius is easier to calculate so do that
77         # shouldn't be affected by mask data but should make sure the
78         # mask is *well* far away from perimeter
79         #gamma_alt = .5 * np.abs(k2).max()
80         #print("half of k2 max is", gamma_alt)
81
82
83         # or actually calculate half of max hessian norm
84         # using frob norm = sqrt(trace(AA^T))
85         hxx, hxy, hyx, hyy = hesh
86         max_hessian_norm = np.sqrt((hxx**2 + 2*hxy**2 + hyx**2 + hyy**2).max())
87         gamma = .5*max_hessian_norm
88
89
90         #print("gamma (half of max hessian norm is)", gamma)
91     if VERBOSE:
92         print('finding Frangi targets with beta={} and gamma={:.2}'.format(beta, gamma))
93
94     # calculate frangi targets at this scale
95     targets = get_frangi_targets(k1,k2,
96                                  beta=beta, gamma=gamma, dark_bg=dark_bg, threshold=False)
97
98     #store results as a dictionary
99     this_scale = {'sigma': sigma,

```

```

        'beta': beta,
99
        'gamma': gamma,
100
        'H': hesh,
101
        'F': targets,
102
        'k1': k1,
103
        'k2': k2,
104
        'border_radius': radius
105
    }

107
    if find_principal_directions:
108
        # principal directions will only be computed for significant regions
109
        pd_mask = np.bitwise_or(targets < (targets.mean() + targets.std()),
110
                                img.mask).filled(1)

111
        if VERBOSE:
112
            percentage_calculated = (pd_mask.size - pd_mask.sum()) / pd_mask.size
113
            print('finding principal directions for {:.2%} of the image'.format(
114
                percentage_calculated))
115
116
        t1, t2 = principal_directions(img, sigma=sigma, H=hesh, mask=pd_mask)
117
118
        this_scale['t1'] = t1
119
        this_scale['t2'] = t2
120
    else:
121
        if VERBOSE:
122
            print('skipping principal direction calculation')
123
124
        # store results as a dictionary
125
        multiscale.append(this_scale)

126
127
    return multiscale

128
def match_on_skeleton(skeleton_of, layers, VERBOSE=True):
129
    """using the computed skeleton of ``skeleton_of``,
130
    return a composite image where blobs layers are incrementally added to the
131
    composite image if that blob coincides at some location of the skeleton
132
    """
133
134
    if ma.is_masked(skeleton_of):
135

```

```

    skeleton_of = skeleton_of.filled(0)

137
skel = skeletonize(skeleton_of)
matched_all = np.zeros_like(skel)

141 # in reverse order (largest to smallest)
for n in range(layers.shape[-1]-1, -1, -1):
    print('matching in layer {}'.format(n))
    current_layer = layers[:, :, n]

145 # only care about things in the current layer above the mean of that
147 # layer (AAAAH)
current_layer = current_layer > current_layer.mean()
#current_layer = current_layer > 0.2

151 # don't match anything that's been matched already
current_layer[matched_all] = 0

153 # label each connected blob
155 el, nl = label(current_layer, return_num=True)
matched = np.zeros_like(current_layer)

157 for region in range(1, nl+1):
    if np.logical_and(el==region, skel).any():
        matched = np.logical_or(matched, el==region)

161 matched_all = np.logical_or(matched_all, matched)

163
return matched_all

165

167 def apply_threshold(targets, alphas, return_labels=True):
    """
    169 if return_labels is True, return 1,...,n for scale at which
    max target was found (or 0 if no match),
    171 otherwise simply return a binary matrix

    173 targets is a (M,N,n) shape matrix (like F_all)
    alphas is a list / 1d array of alphas of length n

```

```

175

177     if alphas is a single number, then this should work just fine

179     for convenience,

181     if return labels is true, this will return both the final
182         threshold and the labels as two separate matrices
183     """

185     # you could make this work in 2D if you wanted to so alphas is a
186     # constant and targets is only 2D but that's a later day
187
188     # make it an array (even if it's a single element)
189     alphas = np.array(alphas)

191     # if it's just a MxN matrix, expand it trivially so it works below

193     if targets.ndim == 2:

195         targets = np.expand_dims(targets,2)

197         # either there's an alpha for each channel or there's a single
198         # alpha to be broadcast across all channels
199         assert (targets.shape[-1] == alphas.size) or (alphas.size == 1)

201         # pixels that passed the threshold at any level
202         passed = (targets >  alphas).any(axis=-1)

203
204     if not return_labels:

205         # works by broadcasting
206         return passed

207
208
209     # get label of where maximum occurs
210     wheres = targets.argmax(axis=-1)

211
212     # reserve 0 label for no match
213     wheres += 1

```

```

215     # then remove anything that didn't pass the threshold
216     wheres[np.invert(passed)] = 0
217
218     assert np.all( passed == (wheres > 0) )
219
220     return passed, wheres
221
222 def extract_pcsvn(filename, alpha=.15, alphas=None,
223                     log_range=None, scales=None, betas=None,
224                     DARK_BG=True, dilate_per_scale=True, n_scales=20,
225                     verbose=True, generate_graphs=True,
226                     generate_json=True, output_dir=None):
227
228     raw_img = get_named_placenta(filename, maskfile=None)
229
230     ##### Multiscale & Frangi Parameters #####
231
232     # set range of sigmas to use
233
234     log_min, log_max = log_range
235
236     if scales is None:
237
238         #log_min = -1 # minimum scale is 2**log_min
239         #log_max = 4.5 # maximum scale is 2**log_max
240
241         scales = np.logspace(log_min, log_max, n_scales, base=2)
242
243     #alpha = 0.15 # Threshold for vesselness measure
244
245     if betas is None:
246
247         betas = [0.5 for s in scales] #anisotropy measure
248
249     # set gammas
250
251     # declare None here to calculate half of hessian's norm
252     gammas = [None for s in scales] # structureness parameter
253
254     #####Do preprocessing (e.g. clahe)#####
255
256     img = raw_img
257     bg_mask = img.mask

```

```

253
254     ##### Logging #####
255
256     if verbose:
257         print(" Running pcsvn.py on the image file", filename,
258               "with frangi parameters as follows:")
259         print("alpha (vesselness threshold): ", alpha)
260         print("scales:", scales)
261         print("betas:", betas)
262         print("gammas will be calculated as half of hessian norm")
263
264     ##### Multiscale Frangi Filter #####
265
266     multiscale = make_multiscale(img, scales, betas, gammas,
267                                   find_principal_directions=False,
268                                   dilate=dilate_per_scale,
269                                   dark_bg=DARK_BG,
270                                   VERBOSE=verbose)
271
272     gammas = [scale['gamma'] for scale in multiscale]
273
274     border_radii = [scale['border_radius'] for scale in multiscale]
275
276     ##### Process Multiscale Targets #####
277
278     # fix targets misreported on edge of plate
279     # wait are we doing this twice?
280
281     if dilate_per_scale:
282
283         if verbose:
284             print('trimming collars of plates (per scale)')
285
286         for i in range(len(multiscale)):
287
288             f = multiscale[i]['F']
289
290             # twice the buffer (be conservative!)
291
292             radius = int(multiscale[i]['sigma']*2)
293
294             if verbose:
295
296                 print('dilating plate for radius={}'.format(radius))
297
298                 f = dilate_boundary(f, radius=radius, mask=img.mask)
299
300                 multiscale[i]['F'] = f.filled(0)
301
302             else:
303
304                 for i in range(len(multiscale)):
```

```

# harden mask (best way to do this??)
293 multiscale[i]['F'] = multiscale[i]['F'].filled(0)

295 #####Extract Multiscale Features#####
296
297 pass

298 #####Make Composite#####
299
300
301 F_all = np.dstack([scale['F'] for scale in multiscale])
302
303 if generate_graphs:
304
305     analyze_targets(F_all, img)
306
307 if generate_json:
308
309     time_of_run = datetime.datetime.now()
310     timestamp = time_of_run.strftime("%y%m%d_%H%M")
311
312     if alphas is None:
313         alphas_out = 'None'
314     else:
315         alphas_out = list(alphas)
316
317     logdata = {'time': timestamp,
318                'filename': filename,
319                'DARK_BG': DARK_BG,
320                'fixed_alpha': alpha,
321                'VT_alphas': alphas_out,
322                'betas': list(betas),
323                'gammas': gammas,
324                'sigmas': list(scales),
325                'log_min': log_min,
326                'log_max': log_max,
327                'n_scales': n_scales,
328                'border_radii': border_radii
329            }
330
331 if output_dir is None:

```

```

331     output_dir = 'output'

333     base = os.path.basename(filename)
334     *base, suffix = base.split('.')
335     dumpfile = os.path.join(output_dir,
336                             ''.join(base) + '_{}.' + str(timestring)
337                             + '.json')

339     with open(dumpfile, 'w') as f:
340         json.dump(logdata, f, indent=True)

341     return F_all, img, scales, alphas

343
344     def get_outname_lambda(filename, output_dir=None, timestring=None):
345         """
346         return a lambda function which can build output filenames
347         """
348
349     if output_dir is None:
350         output_dir = 'output'

351     base = os.path.basename(filename)
352     *base, suffix = base.split('.')

353
354     if timestring is None:
355         time_of_run = datetime.datetime.now()
356         timestring = time_of_run.strftime("%y%m%d_%H%M")

357
358     outputstub = ''.join(base) + '_' + timestring + '_{}.' + suffix
359     return lambda s: os.path.join(output_dir, outputstub.format(s))

360
361     def analyze_targets(F_all, img):
362
363         #####The max Frangi target#####
364
365         # for display purposes
366         F_max = F_all.max(axis=-1)
367         F_max = ma.masked_array(F_max, mask=img.mask)

368
369         # is the frangi vesselness measure strong enough

```

```

#F_cumulative = (F_max > alpha)
371

373 # Variable threshold
N = min(img.shape) // 2
375 #alphas = np.logspace(-2,0, num=len(scales))*.7
alphas = np.sqrt(1.2*scales / N)
377 # try a logistic curve
#alphas = 1 / (1+np.exp(-.2*(scales-np.sqrt(N))))
379 #alphas = scales/32
#alphas = np.logspace(-2.5,-1, num=len(scales))
381 #alphas = np.linspace(0.01,1,num=len(scales))
#alphas = np.sqrt(scales / scales.max())
383

385 #time_of_run = datetime.datetime.now()
#timestring = time_of_run.strftime("%y%m%d_%H%M")

387 # Process Composite #####3

389 # (deprecated, doesn't change much and takes forever)
#matched_all = match_on_skeleton(F_cumulative, F_all)
391 #wheres[np.invert(matched_all)] = 0 # first label is stuff that didn't match

393 FT, wheres = apply_threshold(F_all, alpha)
394 VT, wheres_VT = apply_threshold(F_all, alphas)
395 #####
396 ###### THE REST IS JUST OUTPUT AND LOGGING
397

399 print('generating outputs!')
400 crop = cropped_args(img)
401 """
402 OUTPUT_DIR = 'output'
403 base = os.path.basename(filename)

405 *base, suffix = base.split('.')
406

407 # make this its own function and just do a partial here.
outputstub = ''.join(base) + '_' + timestring + '_{}.' + suffix

```

```

409     outname = lambda s: os.path.join(OUTPUT_DIR, outputstub.format(s))
410     """
411
412     outname = get_outname_lambda(filename)
413
414     # SKELETONIZED OUTPUT
415     plt.imsave(outname('skel'), skeletonize(FT[crop]),
416                cmap=plt.cm.gray)
417     plt.imsave(outname('fmax_threshholded'), FT[crop],
418                cmap=plt.cm.gray_r)
419     plt.imsave(outname('fmax_variable_threshold'), VT[crop],
420                cmap=plt.cm.gray_r)
421
422     # Max Frangi score
423     fig, ax = plt.subplots()
424     plt.imshow(F_max[crop], cmap=plt.cm.gist_ncar)
425     #plt.title(r'Max Frangi vesselness measure below threshold $\alpha={:.2f}{}'.format(alpha))
426
427     plt.title('Maximum Frangi vesselness score')
428     plt.axis('off')
429     c = plt.colorbar()
430     c.set_ticks(np.linspace(0,1,num=11))
431     plt.clim(0,1)
432     plt.tight_layout()
433     plt.savefig(outname('fmax'), dpi=300)
434
435     plt.close()
436
437     scale_label_figure(wherees, scales,
438                        outname('labeled_test'),
439                        crop=crop)
440
441     scale_label_figure(wherees_VT, scales,
442                        outname('labeled_VT_test'),
443                        crop=crop)
444
445     confusion_matrix = compare_trace(FT, filename=filename)
446
447     plt.imsave(outname('confusion'), confusion_matrix[crop])

```

```

447     confusion_matrix = compare_trace(VT, filename=filename)

449     plt.imsave(outname('confusion_VT'), confusion_matrix[crop])

451

453 #####Make Connected Graph#####
454
455 pass

457 #####Measure#####
458
459 pass

461 """
462 """
463 """
464
465 def scale_label_figure(wheres, scales, savefilename=None,
466                         crop=None, show_only=False, image_only=False):
467     """
468     crop is a slice object.
469     if show_only, then just plt.show, not save
470     """
471     if crop is not None:
472         wheres = wheres[crop]

473     fig, ax = plt.subplots() # not sure about figsize
474     N = len(scales)+1 # number of scales / labels
475
476     # discrete sample of color map
477     #cmap = plt.get_cmap('nipy_spectral', N)

478     # get 20 samples from the colormap [R,G,B,A] array
479     tab = plt.cm.viridis_r(np.linspace(0,1,num=N))
480     tabe = np.vstack(([0,0,0,1], tab)) # add black as first entry
481     tabemap = mpl.colors.ListedColormap(tabe)
482
483     if image_only:
484         plt.imsave(savefilename, wheres, cmap=tabemap)

```

```

    else:
487     imgplot = ax.imshow(where, cmap=tabemap)
        # discrete colorbar
489     cbar = plt.colorbar(imgplot)

491     # this is apparently hackish, beats me
492     tick_locs = (np.arange(N) + 0.5)*(N-1)/N
493
494     cbar.set_ticks(tick_locs)
495     # label each tick with the sigma value
496     scalelabels = [r"\sigma = {:.2f}{}".format(s) for s in scales]
497     scalelabels.insert(0, "(no match)")
498     # label with their sigma value
499     cbar.set_ticklabels(scalelabels)
500     #ax.set_title(r"Scale ($\sigma$) of maximum vesselness ")
501     plt.tight_layout()

503     #plt.savefig(outname('labeled'), dpi=300)
504     if show_only or (savefilename is None):
505         plt.show()
506     else:
507         plt.savefig(savefilename, dpi=300)

509     plt.close()

511
512
513

515
516     if __name__ == "__main__":
517
518
519     from get_placenta import list_placentas
520
521     show = plt.show
522     imshow = plt.imshow
523
524     placentas = list_placentas('T-BN')[:15]

```

```

525     N_samples = len(placentas)

527     print(N_samples, "samples total!")
528     for i, filename in enumerate(placentas):
529         print('*'*80)
530         print('extracting PCSVN of', filename,
531               '\t({} of {})'.format(i,N_samples))

533     alpha = .08
534     DARK_BG = False
535     log_range = (-2,3)
536     dilate_per_scale = False
537
538     F, img, scales = extract_pcsvn(filename, DARK_BG=DARK_BG,
539                                     alpha=alpha, log_range=log_range,
540                                     dilate_per_scale=dilate_per_scale,
541                                     verbose=False, generate_graphs=False)

542
543     break

```

listings/get_placenta.py

```

1 #!/usr/bin/env python3

3 # change this module to placenta instead of get_placenta

5 """
7 Get registered, unpreprocessed placental images. No automatic registration
9 (i.e. segmentation of placental plate) takes place here. The background,
11 however, *is* masked.

11 Again, there is no support for unregistered placental pictures.
12 Any region outside of the placental plate MUST be black.

13
14 There is currently no support for color images.

15
16 TODO:
17     - Build sample base & organize data :v)

```

```

    - Test on many other images.

19   - Think of how the interface should really work, esp for get_named_placenta
    - Fix logic in mask_background
21   - Catch errors better.
    - Support for color images
23   - Show a better test
    - Be able to grab trace files too.
25   - Cache masked samples.

    """
27
import numpy as np
29 import numpy.ma as ma
from skimage import segmentation, morphology
31 import os.path
import os

33
from scipy.ndimage import imread
35
36 def open_typefile(filename, filetype, sample_dir=None):
    """
37
        filetype is either 'mask' or 'trace'
39
    """
40
        # try to open what the mask *should* be named
41
        # this should be done less hackishly
42
        # for example, if filename is 'ncs.1029.jpg' then
43
        # this would set the maskfile as 'ncs.1029.mask.jpg'

44
        if filetype not in ("mask", "trace"):
45            raise NotImplementedError("Can only deal with mask or trace files.")
46
47
        *base, suffix = filename.split('.')
48        base = ''.join(base)
49        filetype = '.'.join((base, filetype, suffix))
50
51
        if sample_dir is None:
52            sample_dir = 'samples'
53
54
        typefile = os.path.join(sample_dir, typefile)

```

```

57     try:
58
59         M = imread(typefile, mode='L')
60
61     except FileNotFoundError:
62
63         print('Could not find file', typefile)
64
65         raise
66
67
68     return M
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95

```

try:

M = imread(typefile, mode='L')

except FileNotFoundError:

print('Could not find file', typefile)

raise

return M

def open_tracefile(tracefile):

"""

open up the trace matrix with filename 'tracefile'

#TODO: expand this later to handle arterial traces and venous traces

"""

if sample_dir is None:

sample_dir = 'samples'

tracefile = os.path.join(sample_dir, tracefile)

trace = imread(tracefile, mode='L')

return 1's and 0's (or convert to binary instead)

return trace != 0

def get_named_placenta(filename, sample_dir=None, masked=True,

maskfile=None):

"""

This function is to be replaced by a more ingenious/natural

way of accessing a database of unregistered and/or registered

placental samples.

INPUT:

filename: name of file (including suffix?) but NOT directory

masked: return it masked.

maskfile: if supplied, this use the file will use a supplied 1-channel

mask (where 1 represents an invalid/masked pixel, and 0

represents a valid/unmasked pixel. the supplied image must be

the same shape as the image. if not provided, the mask is

calculated (unless masked=False)

```

    the file must be located within the sample directory
97
    If maskfile is 'None' then this function will look for
99
        a default maskname with the following pattern:
101
            test.jpg -> test.mask.jpg
102         ncs.1029.jpg -> ncs.1029.mask.jpg
103
        sample_directory: Relative path where sample (and mask file) is located.
105         defaults to './samples'
106
107     if masked is true (default), this returns a masked array.
108
109     NOTE: A previous logical incongruity has been corrected. Masks should have
110         1 as the invalid/background/mask value (to mask), and 0 as the
111         valid/plate/foreground value (to not mask)
112
113     """
114
115     if sample_dir is None:
116         sample_dir = 'samples'
117
118     full_filename = os.path.join(sample_dir, filename)
119
120     raw_img = imread(full_filename, mode='L')
121
122     if maskfile is None:
123         # try to open what the mask *should* be named
124         # this should be done less hackishly
125         # for example, if filename is 'ncs.1029.jpg' then
126         # this would set the maskfile as 'ncs.1029.mask.jpg'
127         *base, suffix = filename.split('.')
128         test_maskfile = ''.join(base) + '.mask.' + suffix
129         test_maskfile = os.path.join(sample_dir, test_maskfile)
130         try:
131             mask = imread(test_maskfile, mode='L')
132         except FileNotFoundError:
133             print('Could not find maskfile', test_maskfile)
134             print('Please supply a maskfile. Autogeneration of mask',
135                  'files is slow and buggy and therefore not supported.')
136             raise

```

```

135         #return mask_background(raw_img)
136     else:
137
138         # set maskfile name relative to path
139         maskfile = os.path.join(sample_dir, maskfile)
140         mask = imread(maskfile, mode='L')
141
142     return ma.masked_array(raw_img, mask=mask)
143
144 def check_filetype(filename, assert_png=True, assert_standard=False):
145     """
146     'T-BN8333878.raw.png' returns 'raw'
147     'T-BN8333878.mask.png' returns 'mask'
148     'T-BN8333878.png' returns 'base'
149
150     if assert_png is True, then raise assertion error if the file
151     is not of type png
152
153     if assert_standard, then assert the filetype is
154     mask, base, trace, or raw.
155
156     etc.
157     """
158
159     basename, ext = os.path.splitext(filename)
160
161     if ext != '.png':
162         if assert_png:
163             assert ext == '.png'
164
165     sample_name, typestub = os.path.splitext(basename)
166
167     if typestub == '':
168         # it's just something like 'T-BN8333878.png'
169         return 'base'
170     elif typestub in ('.mask', '.trace', '.raw'):
171         # return 'mask' or 'trace' or 'raw'
172         return typestub.strip('.')
173     else:
174         print('unknown filetype:', typestub)
175         print('is it a weird filename?')

```

```

175     print('warning: lookup failed, unknown filetype:' + typestub)

177     return typestub

179 def list_placentas(label=None, sample_dir=None):
180     """
181     label is the specifier, basically just ''.startswith()
182
183     only real use is to find all the T-BN* files
184
185     this is hackish, if you ever decide to use a file other than
186     png then this needs to change
187     """
188
189     if sample_dir is None:
190         sample_dir = 'samples'
191
192     if label is None:
193         label = '' # str.startswith('') is always True
194
195     placentas = list()
196
197     for f in os.listdir(sample_dir):
198
199         if f.startswith(label):
200             # oh man they gotta be png files
201             if check_filetype(f) == 'base':
202                 placentas.append(f)
203
204     return sorted(placentas)

205
206 def mask_background(img):
207     """
208
209     Warning: this function is slow and buggy and therefore deprecated
210     as "out of scope". Please fix or remove.
211
212     Masks all regions of the image outside the placental plate.

```

```

213     INPUT:
214
215         img:
216
217             A color or grayscale array corresponding to an image of a placenta
218             with the plate in the 'middle.' Outer regions should be black.
219
220     OUTPUT:
221
222         masked_img:
223
224             A numpy.ma.masked_array with the same dimensions.
225
226         """
227
228         print("""
229             Warning, this function is slow and buggy and therefore
230             deprecated. Please supply a mask file yourself.
231
232             """
233
234         )
235
236
237         if img.ndim == 3:
238
239             #mark any pixel with with content in any channel
240             bg_mask = img.any(axis=-1)
241             bg_mask = np.invert(bg_mask)
242
243             # make the mask multichannel to match dim of input
244             bg_mask = np.repeat(bg_mask[:, :, np.newaxis], 3, axis=2)
245
246
247         else:
248
249             # same as above
250             bg_mask = (img != 0)
251             bg_mask = np.invert(bg_mask)
252
253
254             # the above approach will probably work for any real image (i.e. a
255             # photograph). it will obviously fail for any image where there is true black
256             # in the placental plane. This should work instead:
257
258
259             # find the outer boundary and mark outside of it.
260             # run with defaults, sufficient
261             bound = morphology.convex_hull_image(bg_mask)
262             bound = segmentation.find_boundaries(bg_mask, mode='inner', background=1)
263             bg_mask[bound] = 1

```

```

253     #remove any small holes found inside the plate (regions or single pixels
254     #that happen to be black). run with defaults, sufficient
255     holes = morphology.remove_small_holes(bg_mask)
256     bg_mask[holes] = 1
257
258     return ma.masked_array(img, mask=bg_mask)
259
260
261     def show_mask(img):
262         """
263             show a masked grayscale image with a dark blue masked region
264
265             custom version of imshow that shows grayscale images with the right colormap
266             and, if they're masked arrays, sets makes the mask a dark blue
267             a better function might make the grayscale value dark blue
268             (so there's no confusion)
269
270             """
271
272             from numpy.ma import is_masked
273             from skimage.color import gray2rgb
274             import matplotlib.pyplot as plt
275
276
277             if not is_masked(img):
278                 plt.imshow(img, cmap=plt.cm.gray)
279             else:
280
281                 mimg = gray2rgb(img.filled(0))
282                 # fill blue channel with a relatively dark value for masked elements
283                 mimg[img.mask, 2] = 60
284                 plt.imshow(mimg)
285
286
287             if __name__ == "__main__":
288
289                 """test that this works on an easy image."""
290
291                 from scipy.ndimage import imread
292                 import matplotlib.pyplot as plt
293                 test_filename = 'barium1.png'

```

```

291 #test_maskfile = 'barium1.mask.png'

293 img = get_named_placenta(test_filename, maskfile=None)

295 print('showing the mask of', test_filename)
print('run plt.show() to see masked output')
297 show_mask(img)

299
300
301 def _cropped_bounds(img, mask=None):
302
303     if mask is not None:
304
305         img = ma.masked_array(img, mask=mask)
306
307         X,Y = (np.argwhere(np.invert(img.mask)).any(axis=k)).squeeze() for k in (0,1))
308
309         if X.size == 0:
310             X = [None,None] # these will slice correctly
311
312         if Y.size == 0:
313             Y = [None,None]
314
315     return Y[0],Y[-1],X[0],X[-1]

316
317 def cropped_args(img, mask=None):
318     """
319     get a slice that would crop image
320     i.e. img[cropped_args(img)] would be a cropped view
321     """
322
323     x0, x1, y0,y1 = _cropped_bounds(img, mask=None)
324
325     return np.s_[x0:x1,y0:y1]

326
327 def cropped_view(img, mask=None):
328     """
329     removes entire masked rows and columns from the borders of a masked array.
330     will return a masked array of smaller size

```

```

    don't ask me about data
331
    the name sucks too
333
"""
335
# find first and last row with content
x0, x1, y0, y1 = _cropped_bounds(img, mask=mask)
337
return img[x0:x1, y0:y1]

```

listings/alpha_sweep_demo.py

```

#!/usr/bin/env python3
2
"""
4 alpha_sweep_demo.py
6 show how much variable alphas affect the output.
8
10 from get_placenta import get_named_placenta, cropped_args, cropped_view
11 from get_placenta import list_placentas, open_typefile
12
13 from score import compare_trace
14
15 from pcsvn import extract_pcsvn, scale_label_figure, apply_threshold
16 from pcsvn import get_outname_lambda
17
18 import numpy as np
19 import numpy.ma as ma
20
21 import matplotlib.pyplot as plt
22
23 from hfft import fft_gradient
24 from score import mcc
25 #filename = 'T-BN0033885.png'
26 placentas = list_placentas('T-BN')
27 n_samples = len(placentas)

```

```

28
29     OUTPUT_DIR = 'output/newalpha'
30
31     DARK_BG = True
32
33     log_range = (-2, 4.5)
34
35     n_scales = 10
36
37     scales = np.logspace(log_range[0], log_range[1], num=n_scales, base=2)
38
39     alphas = scales**(2/3) / scales[-1]
40
41     #alphas = [0.1 for s in scales]
42
43     #betas = np.linspace(.5, .9, num=n_scales)
44
45     betas = None
46
47     print(n_samples, "samples total!")
48
49     for i, filename in enumerate(placentas):
50
51         print('*'*80)
52
53         print('extracting PCSVN of', filename,
54               '\t({} of {})'.format(i, n_samples))
55
56         F, img, _, _ = extract_pcsvn(filename, DARK_BG=DARK_BG,
57
58                                         alpha=.1, alphas=alphas, betas=betas,
59                                         scales=scales, log_range=log_range,
60                                         verbose=False, generate_graphs=False,
61                                         n_scales=n_scales, generate_json=True,
62                                         output_dir=OUTPUT_DIR)
63
64
65     #G = list()
66
67
68     #for s in scales:
69
70     #     g = fft_gradient(img, s)
71
72     #     G.append(g)
73
74     #G = np.dstack(G)
75
76     #f = F.copy()
77
78     crop = cropped_args(img)
79
80     print("...making outputs")
81
82     outname = get_outname_lambda(filename, output_dir=OUTPUT_DIR)
83
84
85
86     approx, labs = apply_threshold(F, alphas, return_labels=True)
87
88     scale_label_figure(labs, scales, crop=crop, savefilename=outname('2_labeled'),
89                         image_only=True)
90
91
92     confusion = compare_trace(approx, filename=filename)

```

```

    trace = open_typefile(filename, 'trace').astype('bool')
68   trace = np.invert(trace)

70   m_score, counts = mcc(approx, trace, img.mask, return_counts=True)

72   TP, TN, FP, FN = counts

74   total = np.invert(img.mask).sum()
    print('TP: {} \t TN: {} \nFP: {} \t FN: {}'.format(TP, TN, FP, FN))
76   print('TP+TN+FP+FN={} \ntotal pixels={}'.format(TP+TN+FP+FN, total))

78   print("MCC for {}: \t".format(filename), m_score)

80   plt.imsave(outname('1_confusion'), confusion[crop])

82   plt.imsave(outname('0_raw'), img[crop].filled(0), cmap=plt.cm.gray)

84   plt.close('all') # something's leaking :(
    if i > 5:
86     break

```

listings/diffgeo.py

```

#!/usr/bin/env python3
2

4 import numpy as np
import numpy.ma as ma
6

from skimage.feature import hessian_matrix, hessian_matrix_eigvals
8 from numpy.linalg import eig
from functools import partial
10

from hfft import fft_hessian
12

14 def principal_curvatures(img, sigma=1.0, H=None):
    """
16     Return the principal curvatures {fz1, fz2} of an image, that is, the

```

```

eigenvalues of the Hessian at each point (x,y). The output is arranged such
18    that  $|k_1| \leq |k_2|$ .
```

20 Input:

```

22      img: An ndarray representing a 2D or multichannel image. If the image
24          is multichannel (e.g. RGB), then each channel will be processed
26          individually. Additionally, the input image may be a masked
28          array-- in which case the output will preserve this mask
29          identically.
```

30 PLEASE ADD SOME INFO HERE ABOUT WHAT SORT OF DTYPES ARE

32 EXPECTED/REQUIRED, IF ANY

34 sigma: (optional) The scale at which the Hessian is calculated.

36 H: (optional) provide sigma (else it will be calculated)

38 Output:

```

39 (K1, K2): A tuple where K1, K2 each are the exact dimension of the
41     input image, ordered in magnitude such that  $|k_1| \leq |k_2|$ 
43     in all locations. If *signed* option is used, then elements
44     of K1, K2 may be negative.
```

46 Example:

```

48
49     >>> K1, K2 = principal_curvatures(img)
50
51     >>> K1.shape == img.shape
52
53     True
54
55     >>> (K1 <= K2).all()
56
57     True
58
59
60     >> K1.mask == img.mask
61
62     True
63
64 """
65
66 # determine if multichannel
67 multichannel = (img.ndim == 3)
```

```

56     if not multichannel:
57         # add a trivial dimension
58         img = img[:, :, np.newaxis]
59
60     K1 = np.zeros_like(img, dtype='float64')
61     K2 = np.zeros_like(img, dtype='float64')
62
63     for ic in range(img.shape[2]):
64
65         channel = img[:, :, ic]
66
67         # returns the tuple (Hxx, Hxy, Hyy)
68         if H is None:
69             H = hessian_matrix(channel, sigma=sigma)
70
71         # returns tuple (l1,l2) where l1 >= l2 but this *includes* sign*
72         L = hessian_matrix_eigvals(*H)
73
74         L = np.vstack(L)
75
76         mag = np.argsort(abs(L), axis=-1)
77
78         # just some slice nonsense
79         ix = np.ogrid[0:L.shape[0], 0:L.shape[1], 0:L.shape[2]]
80
81         L = L[ix[0], ix[1], mag]
82
83         # now k2 is larger in absolute value, as consistent with Frangi paper
84
85         K1[:, :, ic] = L[:, :, 0]
86         K2[:, :, ic] = L[:, :, 1]
87
88     try:
89         mask = img.mask
90     except AttributeError:
91         pass
92     else:
93         K1 = ma.masked_array(K1, mask=mask)
94         K2 = ma.masked_array(K2, mask=mask)

```

```

# now undo the trivial dimension
96 if not multichannel:
97     K1 = np.squeeze(K1)
98     K2 = np.squeeze(K2)

100    return K1, K2

102 def reorder_eigs(L1,L2):
103     """
104         L1, L2 contain a 2D matrix of eigenvalues at each point
105         so that L1 <= L2 at each element.
106
107         this reorders this so that |L1| <= |L2| instead.
108
109         this could (if desired) also return the permutation array
110         but does not do so presently
111     """
112
113     L = np.dstack((L1,L2))
114     mag = np.argsort(abs(L), axis=-1)

116     # just some slice nonsense

118     ##### FINISH T HISSSS
119
120     def principal_directions(img, sigma, H=None, mask=None):
121         """
122             will ignore calculation of principal directions of masked areas
123
124             despite the name, this function actually returns the theta corresponding to
125             leading and trailing principal directions, i.e. angle w / x axis
126
127             FIX THE MASK BUSINESS
128
129         """
130
131         if H is None:
132             H = hessian_matrix(img, sigma)

133         Hxx, Hxy, Hyx, Hyy = H

```

```

134
135     # is no mask provided
136
137     if mask is None:
138
139         try:
140
141             mask = img.mask
142
143         except AttributeError:
144
145             masked = False
146
147         else:
148
149             masked = True
150
151     else:
152
153         masked = True
154
155
156     dims = img.shape
157
158
159     # where to store
160     trailing_thetas = np.zeros_like(img, dtype='float64')
161     leading_thetas = np.zeros_like(img, dtype='float64')
162
163
164     # maybe implement a small angle correction
165
166     for i, (xx, xy, yy) in enumerate(np.nditer([Hxx, Hxy, Hyy])):
167
168
169         # grab the (x,y) coordinate of the hxx, hxy, hyy you're using
170         subs = np.unravel_index(i, dims)
171
172
173         # ignore masked areas (if masked array)
174
175         if masked and mask[sub]:
176
177             continue
178
179
180         h = np.array([[xx, xy], [xy, yy]]) # per-pixel hessian
181         l, v = eig(h) # eigenvectors as columns
182
183
184         # reorder eigenvectors by (increasing) magnitude of eigenvalues
185         v = v[:, np.argsort(np.abs(l))]
186
187
188         # angle between each eigenvector and positive x-axis
189         # arccos of first element (dot product with (1,0) and eigvec is already
190         # normalized)
191
192         trailing_thetas[sub] = np.arccos(v[0,0]) # first component of each

```

```

    leading_thetas[sub] = np.arccos(v[0,1]) # first component of each
174
175     if masked:
176
177         leading_thetas = ma.masked_array(leading_thetas, mask)
178         trailing_thetas = ma.masked_array(trailing_thetas, mask)
179
180
181
182
183
184     return trailing_thetas, leading_thetas
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210

```

```

212     #     tic = time.time()
213     #     print('calculating hessian via FFT (F)')
214     #     h = fft_hessian(img, sigma)
215
216     #     toc = time.time()
217     #     print('time elapsed: ', toc - tic)
218     #     tic = time.time()
219     #     print('calculating principal curvatures for  $\sigma={}$ '.format(sigma))
220     #     K1,K2 = principal_curvatures(img, sigma=sigma, H=H)
221     #     toc = time.time()
222     #     print('time elapsed: ', toc - tic)
223     #     tic = time.time()
224     #     print('calculating principal curvatures for  $\sigma={}$  (fast)'.format(sigma))
225     #     k1,k2 = principal_curvatures(img, sigma=sigma, H=h)
226
227     #     toc = time.time()
228     #     print('time elapsed: ', toc - tic)
229     #     tic = time.time()
230
231     # ######
232
233     #     print('calculating targets for  $\sigma={}$ '.format(sigma))
234     #     T = get_targets(K1,K2, threshold=False)
235
236     #     toc = time.time()
237     #     print('time elapsed: ', toc - tic)
238     #     tic = time.time()
239
240     #     print('calculating targets for  $\sigma={}$  (fast)'.format(sigma))
241     #     t = get_targets(k1,k2, threshold=False)
242
243     #     toc = time.time()
244     #     print('time elapsed: ', toc - tic)
245
246     # ######
247
248     #     print('extending masks')
249
250     #     # extend mask over nontargets items

```

```

252     #     img1 = ma.masked_where( T < T.mean(), img)
253     #     img2 = ma.masked_where( t < t.mean(), img)

254     #     tic = time.time()
255     #     print('calculating principal directions for ||= {}'.format(sigma))
256     #     T1,T2 = principal_directions(img1, sigma=sigma, H=H)
257     #     toc = time.time()
258     #     print('time elapsed: ', toc - tic)
259     #     tic = time.time()

260     #     print('calculating principal directions for ||= {} (fast)'.format(sigma))
261     #     t1,t2 = principal_directions(img2, sigma=sigma, H=h)
262     #     toc = time.time()
263     #     print('time elapsed: ', toc - tic)

```

listings/hfft.py

```

#!/usr/bin/env python3

2

import numpy as np
4 from scipy import signal
import scipy.fftpack as fftpack

6

"""
8 hfft.py is the implementation of calculating the hessian of a real

10 image based in frequency space (rather than direct convolution with a gaussian
as is standard in scipy, for example).

12 TODO: PROVIDE MAIN USAGE NOTES
14 """

16 def gauss_freq(shape, ||=1.):
    """
18     DEPRECATED

20     NOTE:
        this function is/should be? for illustrative purposes only--
22         we can actually build this much faster using the builtin

```

```

    scipy.signal.gaussian rather than a roll-your-own
24
build a shape=(M,N) sized gaussian kernel in frequency space
26 with size  $\sqrt{C}$ 
27
28 (due to the convolution theorem for fourier transforms, the function
29 created here may simply be *multiplied* against the signal.
30
31 """
32
33 M, N = shape
34 fgauss = np.fromfunction(lambda i, j: ((i + M + 1) / 2)**2 + ((j + N + 1) / 2)**2, shape=shape)
35
36 # is this used?
37 coeff = (1 / (2 * np.pi * C**2))
38
39 return np.exp(-fgauss / (2 * C**2))
40
41 def blur(img, sigma):
42 """
43 DEPRECATED
44 a roll-your-own FFT-implemented gaussian blur.
45 fft_gaussian below is preferred (it is more efficient)
46 """
47
48 I = fftpack.fft2(img) # get 2D transform of the image
49
50 # do whatever
51
52 I *= gauss_freq(I.shape, sigma)
53
54 return fftpack.ifft2(I).real
55
56 def fft_gaussian(img, sigma, A=None):
57 """
58
59 https://docs.scipy.org/doc/scipy/reference/generated/scipy.signal.fftconvolve.html

```

```

62     in particular the example in which a gaussian blur is implemented.

64     along with the comment:
65
66     "Gaussian blur implemented using FFT convolution. Notice the dark borders
67     around the image, due to the zero-padding beyond its boundaries. The
68     convolve2d function allows for other types of image boundaries, but is far
69     slower"
70
71
72     (i.e. doesn't use FFT).

73
74     note that here, you actually take the FFT of a gaussian (rather than
75     build it in frequency space). there are ~6 ways to do this.
76
77     """
78
79     #create a 2D gaussian kernel to take the FFT of
80
81     # scale factor!
82
83     A = 1 / (2*np.pi*sigma**2)
84
85     kernel = np.outer(A*signal.gaussian(img.shape[0], sigma),
86                         A*signal.gaussian(img.shape[1], sigma))

87
88     return signal.fftconvolve(img, kernel, mode='same')

89
90
91
92
93
94
95
96
97
98
99
100

```

```

102 gaussian_filtered = fft_gaussian(image, sigma=sigma)

104 Lx, Ly = np.gradient(gaussian_filtered)

106 Lxx, Lxy = np.gradient(Lx)
107 Lxy, Lyy = np.gradient(Ly)

108 return (Lxx, Lxy, Lyy)

110

111 def fft_gradient(image, sigma=1.):
112     """ returns gradient norm """
113

114 gaussian_filtered = fft_gaussian(image, sigma=sigma)

116 Lx, Ly = np.gradient(gaussian_filtered)

118 return np.sqrt(Lx**2 + Ly**2)

119 def _old_test():
120     """
121     old main function for testing.
122
123     This simply tests fft_gaussian on a test image, exemplifying the speedup
124     compared to a traditional gaussian.
125     """
126
127     import matplotlib.pyplot as plt
128
129     from skimage.data import camera
130
131     img = camera() / 255.

132 sample_sigmas = (.2, 2, 10, 30)

133 outputs = (fft_gaussian(img, sample_sigmas[0]),
134             fft_gaussian(img, sample_sigmas[1]),
135             fft_gaussian(img, sample_sigmas[2]),
136             fft_gaussian(img, sample_sigmas[3]),
137             )

```

```

140
fig, axes = plt.subplots(nrows=2, ncols=2, figsize=(10, 10))

142
axes[0, 0].imshow(outputs[0], cmap='gray')
144 axes[0, 0].set_title('fft_gaussian IC={}'.format(sample_sigmas[0]))
145 axes[0, 0].axis('off')

146
axes[0, 1].imshow(outputs[1], cmap='gray')
148 axes[0, 1].set_title('fft_gaussian IC={}'.format(sample_sigmas[1]))
149 axes[0, 1].axis('off')

150
axes[1, 0].imshow(outputs[2], cmap='gray')
152 axes[1, 0].set_title('fft_gaussian IC={}'.format(sample_sigmas[2]))
153 axes[1, 0].axis('off')

154
axes[1, 1].imshow(outputs[3], cmap='gray')
156 axes[1, 1].set_title('fft_gaussian IC={}'.format(sample_sigmas[3]))
157 axes[1, 1].axis('off')

158
plt.tight_layout()
159
160 plt.show()

162 if __name__ == "__main__":
163
164     pass

```

listings/farm_samples.py

```

#!/usr/bin/env python

2
"""
4 This should be a plugin to take images from the folder NCS_vessel_GIMP_xcf
and create trace, mask, and backgrounded images from each xcf file.

6
What follows is actually without plugin syntax. The following commands are what
8 you would type directly into the python console in gimp.

10 """
from gimpfu import *

```

```

12 import os.path

14 # can't use this because we can't run this outside of gimp i think?
# need to write a batch file to run this and then run # gimp -b ...

16

17 #for i, xcfffile in enumerate(glob('*xcf')):

18

19 #basefile, ext = os.path.splitext(xcfffile)

20

21 # get active image

22 img = gimp.image_list()[0]

23

24 # Go to perimeter layer.

25 perimeter = pdb.gimp_image_get_layer_by_name(img, 'perimeter')

26 # could also iterate and say if layer.name = 'perimeter' ...

27

28 # Copy perimeter layer & focus new layer (only visible).

29

30 # .copy() has optional arg of "add_alpha_channel"

31 M = perimeter.copy()

32

33 # set all other layers non visible

34 for layer in img.layers:
    layer.visible = False

35

36 # add in position 0 (top)

37 img.add_layer(M, 0)

38

39 # Remove Alpha Channel.

40 pdb.gimp_layer_flatten(M)

41 ## Invert Colors

42 #pdb.gimp_invert(M)

43

44

45

46 # color exchange yellow & blue to black
pdb.plug_in_exchange(img,m,255,255,0,0,0,0,1,1,1)
47 pdb.plug_in_exchange(img,m,0,0,255,0,0,0,0,1,1,1)

48

49

50 # set FG color to black

```

```

gimp.set_foreground(0,0,0)

52
cx, cy = img.height // 2 , img.width // 2

54
# Bucket Fill Inside black (middle x,y is probably OK)
56 pdb.gimp_edit_bucket_fill(m,0,0,100,0,cx,cy)
# Color Exchange Green to White.

58 pdb.plug_in_exchange(img,m,0,255,0,255,255,255,1,1,1)

60 # Color Exchange Cyan (00ffff) to White.
61 pdb.plug_in_exchange(img,m,0,255,255,255,255,255,1,1,1)

62
# Export Layer as Image called "f".mask.png

64
65 pdb.gimp_file_save(img,m, '/home/luke/test.png', '')

66
# invert (so back exterior now)
67 pdb.invert(m)

m.mode = DARKEN_ONLY_MODE # the constant 9

68
69 # set bottom layer (placenta) to visible
70 img.layers[-1].visible = True

71
72 # now make a new layer called 'main' from visible
73 raw_img = pdb.gimp_layer_new_from_visible(img,img,'raw_img')
74 img.add_layer(main,0)
75 pdb.gimp_file_save(img ,raw_img , '/home/luke/test2.png' , '')

76
77 # now set the veins/artery layers as only visible
78 for layer in img.layers:
    layer.visible = (layer.name in ("arteries", "veins"))

79
80 trace = pdb.gimp_layer_new_from_visible(img,img,'trace')
81 img.add_layer(trace,0)

82
83 pdb.gimp_layer_flatten(trace) # remove alpha channel

84
85 pdb.gimp_desaturate(trace) # turn to grayscale
86 pdb.gimp_threshold(trace,255,255) # anything not 255 turns black

```

```

90
91     pdb.gimp_file_save(img, trace, '/home/luke/testtrace.png', '')
92 # Visible to New Layer called Trace & focus (only visible)
93 # Remove Alpha Channel of layer.
94 # Threshold (127 is default and OK)
95 # Invert Colors
96 # Export Layer as "f".trace.png
97 # Make "Background" Layer and Mask Layer Visible
98 # Move Mask Layer in Front of "Background" Layer
99 # Change Mask Layer Mode to "Darken Only"
100 # Export Visible as *.png

```

listings/frangi.py

```

import numpy as np
2 import numpy.ma
from hfft import fft_hessian
4 from diffgeo import principal_curvatures
from plate_morphology import dilate_boundary
6
def frangi_from_image(img, sigma, beta=0.5, gamma=None, dark_bg=True,
8                     dilation_radius=None, threshold=None,
10                    return_debug_info=False):
    """
    Perform a frangi filtering on img
12    if None, gamma returns half of Frobenius norm on the image
13    if dilation radius is specified, that amount is dilated from the
14    boundary of the image (mask must be specified)

16    input image *must* be a masked array. To implement: supply mask
17    or create a dummy mask if not specified so this can work out of the
18    box on arbitrary images.

20    return_debug info will return anisotropy, structureness measures, as
21    well as the calculated gamma. will return a tuple of
22    (R, S, gamma) where R and S are matrices of shape img.shape
23    and gamma is a float.
24

```

```

26     BIGGER TODO:

28         THIS OVERLAPS WITH pcsvn.make_multiscale
29         USE THIS THERE
30
31     """
32
33     # principal_directions() calculates the frangi filter with
34     # standard convolution and takes forever. FIX THIS!
35
36     hesh = fft_hessian(img, sigma) # the triple (Hxx,Hxy,Hyy)
37
38     k1, k2 = principal_curvatures(img, sigma, H=hesh)
39
40     if dilation_radius is not None:
41
42         # pass None to just get the mask back
43         collar = dilate_boundary(None, radius=dilation_radius,
44                               mask=img.mask)
45
46         # get rid of "bad" K values before you calculate gamma
47         k1[collar] = 0
48         k2[collar] = 0
49
50     if gamma is None:
51
52         gamma = .5 * max_hessian_norm(hesh)
53
54         if np.isclose(gamma,0):
55             print("WARNING: gamma is close to 0. should skip this layer.")
56
57     targets = get_frangi_targets(k1, k2, beta=beta, gamma=gamma,
58                                  dark_bg=dark_bg, threshold=threshold)
59
60     if not return_debug_info:
61
62         return targets
63
64     else:
65
66         return targets, (R, S, gamma)

67
68     def get_frangi_targets(K1,K2, beta=0.5, gamma=None, dark_bg=True, threshold=None):
69
70         """
71
72         returns results of frangi filter. eigenvalues are inputs
73
74

```

```

    if gamma is not supplied, use half of L2 norm of hessian
66    if you want to use half of frobenius norm, calculate it outside here
       """
68
R = anisotropy(K1,K2)
70 S = structureness(K1,K2)

72 if gamma is None:
    # half of max hessian norm (using L2 norm)
74    gamma = .5 * np.abs(K2).max()
    if np.isclose(gamma,0):
        print("warning! gamma is very close to zero. maybe this layer isn't worth it...")
76        print("sigma={:.3f}, gamma={}".format(sigma,gamma))
78        print("returning an empty array")
        return np.zeros_like(img)

80
F = np.exp(-R / (2*beta**2))
82 F *= 1 - np.exp( -S / (2*gamma**2))

84 if dark_bg:
    F = (K2 < 0)*F
86 else:
    F = (K2 > 0)*F

88
if numpy.ma.is_masked(K1):
    F = numpy.ma.masked_array(F, mask=K1.mask)
90
if threshold:
92
    return F < threshold
94 else:
    return F

96
def max_hessian_norm(hesh):
98
    hxx, hxy, hyy = hesh
100
    # frob norm is just sqrt(trace(AA^T)) which is easy for a 2x2
102    max_norm = np.sqrt((hxx**2 + 2*hxy*2 + hyy**2).max())

```

```

104     return max_norm
106
107 def anisotropy(K1,K2):
108     """
109     according to Frangi (1998) this is technically A**2
110     """
111
112     return (K1/K2) **2
113
114 def structureness(K1,K2):
115     """
116     according to Frangi (1998) this is technically S**2
117     """
118
119     return K1**2 + K2**2

```

listings/hfft_demo.py

```

1 #!/usr/bin/env python3
2
3 import numpy as np
4 from skimage.data import camera
5 from skimage.io import imread
6
7 import matplotlib.pyplot as plt
8 from hfft import gauss_freq, blur, fft_gaussian, fft_hessian
9 from scipy.ndimage import gaussian_filter
10
11 from scipy.linalg import norm
12 import timeit
13
14 #img = camera() / 255.
15 img = imread('samples/barium1.png', as_grey=True) / 255.
16 mask = imread('samples/barium1.mask.png', as_grey=True)
17
18 # compare computation speed over sigmas
19
20 # N logarithmically spaced scales between 1 and 2^m
21 N = 5
22 m = 8

```

```

23 sigmas = np.logspace(0,m, num=N, base=2)

25 fft_results = list()
std_results = list()

27

for sigma in sigmas:
    # test statements to compare (fft-based gaussian vs convolution-based)
    fft_test_statement = 'fft_gaussian(img,{})'.format(sigma)
    std_test_statement = 'gaussian_filter(img,{})'.format(sigma)
    # run each statement 1 times (with 2 runs in each trial)
    # returns/appends the average of 3 runs
    fft_results.append(timeit.timeit(fft_test_statement,
                                    number=1, globals=globals()))
    std_results.append(timeit.timeit(std_test_statement,
                                    number=1, globals=globals()))

39     # now actually evaluate both to compare
40     f = eval(fft_test_statement)
41     s = eval(std_test_statement)

43     # normalize each matrix by frobenius norm and take difference
44     # ideally should try to zero out the "mask" area
45     diff = np.abs(f / norm(f) - s / norm(s))
46     raw_diff = np.abs(f - s)
47     # don't care if it's the background
48     diff[mask==1] = 0
49     raw_diff[mask==1] = 0

51     # should format this stuff better into a legible table
52     print(sigma, diff.max(), raw_diff.max())

53

54     lines = plt.plot(sigmas, fft_results, 'go', sigmas, std_results, 'bo')
55     plt.xlabel('sigma (gaussian blur parameter)')
56     plt.ylabel('run time (seconds)')
57     plt.legend(lines, ('fft-gaussian', 'conv-gaussian'))
58     plt.title('Comparision of Gaussian Blur Implementations')

```

listings/plate_morphology.py

```

#!/usr/bin/env python3

from skimage.morphology import disk, binary_erosion, binary_dilation
from skimage.morphology import convex_hull_image
from skimage.segmentation import find_boundaries

import numpy as np
import numpy.ma as ma

def erode_plate(img, erosion_radius=20, plate_mask=None):
    """
    Manually remove (erode) the outside boundary of a plate.

    The goal is remove any influence of the zeroed background
    on reporting derivative information

    NOTE: this is an old deprecated function. use dilate_boundary instead.
    """

    if plate_mask is None:
        # grab the mask from input image
        try:
            plate_mask = img.mask
        except AttributeError:
            raise('Need to supply mask information')

    # convex_hull_image finds white pixels
    plate_mask = np.invert(plate_mask)

    # find convex hull of mask (make erosion calculation easier)
    plate_mask = np.invert(convex_hull_image(plate_mask))

    # this is much faster than a disk. a plus sign might be better even.
    #selem = square(erosion_radius)

    # also this correctly has erosion radius as the RADIUS!
    # input for square() and disk() is the diameter!
    selem = np.zeros((erosion_radius*2 + 1, erosion_radius*2 + 1),
                     dtype='bool')
    selem[erosion_radius, :] = 1
    selem[:, erosion_radius] = 1
    eroded_mask = binary_erosion(plate_mask, selem=selem)

```

```

40
41     # this is by default additive with whatever
42     return ma.masked_array(img, mask=eroded_mask)
43
44 def dilate_boundary(img, radius=10, mask=None):
45     """
46     grows the mask by a specified radius of a masked 2D array
47     Manually remove (erode) the outside boundary of a plate.
48     The goal is remove any influence of the zeroed background
49     on reporting derivative information.
50
51     There is varying functionality here (maybe should be multiple functions
52     instead?)
53
54     If img is a masked array and mask=None, the mask will be dilated and a
55     masked array is outputted.
56
57     If img is any 2D array (masked or unmasked), if mask is specified, then
58     the mask will be dilated and the original image will be returned as a
59     masked array with a new mask.
60
61     If the img is None, then the specified mask will be dilated and returned
62     as a regular 2D array.
63
64     """
65
66     if mask is None:
67         # grab the mask from input image
68         # if img is None this will break too but not handled
69         try:
70             mask = img.mask
71         except AttributeError:
72             raise('Need to supply mask information')
73
74     perimeter = find_boundaries(mask, mode='inner')
75
76     maskpad = np.zeros_like(perimeter)
77
78     M,N = maskpad.shape

```

```

    for i,j in np.argwhere(perimeter):
        # just make a cross shape on each of those points
        # these will silently fail if slice is OOB thus ranges are limited.
        maskpad[max(i-radius,0):min(i+radius,M),j] = 1
        maskpad[i,max(j-radius,0):min(j+radius,N)] = 1

84
new_mask = np.bitwise_or(maskpad, mask)

86
if img is None:
    return new_mask # return a 2D array
else:
    # replace the original mask or create a new masked array
    return ma.masked_array(img, mask=new_mask)

92
#####
94 # DEMO FOR SHOWING OFF DILATE_BOUNDARY EFFECT

96 if __name__ == "__main__":
97
98     from get_placenta import get_named_placenta
99     from frangi import frangi_from_image
100    import matplotlib.pyplot as plt
101    import numpy as np
102    from skimage.exposure import rescale_intensity
103
104    import os.path
105
106    dest_dir = 'demo_output'
107    img = get_named_placenta('T-BN0164923.png')
108
109    #radius = 30
110    sigma = 3
111    radius = 80
112    # IMG WITHOUT DILATING, THEN IMAGE WITH DILATING
113    #sigma = radius/4
114
115    #inset = np.s_[:, :]
116    inset = np.s_[800:1000,500:890]
117    #inset = np.s_[100:300,300:500]

```

```

118     D = dilate_boundary(img, radius=radius)

120

122     # SAVE IT IN THE RIGHT DIRECTORY, ETC plt.savefig(
123
124     # NOW SHOW FRANGI ON THESE IMAGES
125
126     # NOW SHOW THE SAME PICTURE
127
128     Fimg = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=None)
129     #FD = frangi_from_image(img, sigma, dark_bg=False, dilation_radius=radius)
130     FD = frangi_from_image(D, sigma, dark_bg=False)
131
132     #Fimg = rescale_intensity(Fimg)
133     #FD = rescale_intensity(FD)
134
135     fig, axes = plt.subplots(ncols=2, nrows=2, figsize=(14, 7))
136
137     axes[0,0].imshow(img[inset].filled(0), cmap=plt.cm.gray)
138     axes[0,1].imshow(D[inset].filled(0), cmap=plt.cm.gray)
139
140     axes[1,0].imshow(Fimg[inset].filled(0), cmap=plt.cm.nipy_spectral)
141     axes[1,1].imshow(FD[inset].filled(0), cmap=plt.cm.nipy_spectral)
142
143     # this is a hack directly from matplotlib, that's why it's so ugly.
144     plt.setp([a.get_xticklabels() for a in axes[0, :]], visible=False)
145     plt.setp([a.get_yticklabels() for a in axes[:, 1]], visible=False)
146
147     fig.tight_layout()
148
149     plt.savefig(os.path.join(dest_dir, "boundary_dilation_demo.png"), dpi=300)

```

listings/process_NCS_xcfs.py

```

#!/usr/bin/env python
2
"""
4 This should be a plugin to take images from the folder NCS_vessel_GIMP_xcf

```

```

and create trace, mask, and backgrounded images from each xcf file.

6
7     chmod +x and then copy or link to ~/gimp-2.x/plug-ins/
8 """
9

10 from gimpfu import *
11 import os.path
12 from functools import partial

13 #basefile, ext = os.path.splitext(xcffile)

16 def _outname(base, s=None):
17     if s is None:
18         stubs = (base, 'png')
19     else:
20         stubs = (base, s, 'png')

22     return '.'.join(stubs)

24 # get active image
25
26 def process_NCS_xcf(timg, tdrawable):
27     img = timg
28     basename, _ = os.path.splitext(img.name)
29     print "*" * 80
30     print '\n\n'
31
32     print "Processing " , img.name
33     # generate output names easier
34     outname = partial(_outname, base=basename)

35     # get coordinates of the center
36     cx, cy = img.height // 2 , img.width // 2

37     # disable the undo buffer
38     #img.disable_undo()

39
40     #perimeter = pdb.gimp_image_get_layer_by_name(img, 'perimeter')
41
42     for layer in img.layers:

```

```

44     if layer.name.lower() in ('perimeter', 'perimeters'):
45         # .copy() has optional arg of "add_alpha_channel"
46         mask = layer.copy()
47         break
48     else:
49         print("Could not find a perimeter layer.")
50         print("Layers of this image are:")
51         for n,layer in enumerate(img.layers):
52             print "\t", n, ":", layer.name
53         print("Skipping this file.")
54
55     return
56
57     for layer in img.layers:
58         layer.visible = False
59
60     mask.name = "mask" # name the new layer
61     img.add_layer(mask,0) # add in position 0 (top)
62
63     pdb.gimp_layer_flatten(mask) # Remove Alpha Channel.
64
65     # remove unneeded (i hope) annotations
66     # color exchange yellow & blue to black
67     pdb.plug_in_exchange(img,mask,255,255,0,0,0,0,1,1,1)
68     pdb.plug_in_exchange(img,mask,0,0,255,0,0,0,1,1,1)
69
70     # set FG color to black (for tools, not of image)
71     gimp.set_foreground(0,0,0)
72
73
74     # Bucket Fill Inside black (center pixel is hopefully fine)
75     pdb.gimp_edit_bucket_fill(mask,0,0,100,0,0,cx,cy)
76
77     # Color Exchange Green to White.
78     pdb.plug_in_exchange(img,mask,0,255,0,255,255,255,1,1,1)
79
80     # Color Exchange Cyan (00ffff) to White.
81     pdb.plug_in_exchange(img,mask,0,255,255,255,255,255,1,1,1)
82

```

```

# Export Layer as Image called "f".mask.png
84 pdb.gimp_file_save(img,mask, outname(s="mask"), '')

86 # invert (so exterior is now black)
87 pdb.gimp_invert(mask)
88 mask.mode = DARKEN_ONLY_MODE # the constant 9

89 # set bottom layer (placenta) to visible
90 raw = img.layers[-1]
91 raw.visible = True

92 # now make a new layer called 'raw_img' from visible
93 base = pdb.gimp_layer_new_from_visible(img,img,'base')
94 img.add_layer(base,0)
95 pdb.gimp_file_save(img , base, outname(s=None) , '')

96 # now get rid of mask and save the raw image
97 mask.visible = False
98 pdb.gimp_file_save(img, raw, outname(s='raw') , '')

99 # now set the veins/artery layers as only visible
100 for layer in img.layers:
101     layer.visible = (layer.name.lower() in ("arteries", "veins"))

102 # now set the veins/artery layers as only visible
103 trace = pdb.gimp_layer_new_from_visible(img,img,'trace')
104 img.add_layer(trace,0)

105 pdb.gimp_layer_flatten(trace) # remove alpha channel

106 pdb.gimp_desaturate(trace) # turn to grayscale
107 pdb.gimp_threshold(trace,255,255) # anything not 255 turns black

108 pdb.gimp_file_save(img , trace, outname(s='trace') , '')

109 print "Saved. "

110

111

112 register(
113     "process_NCS_xcf",

```

```

122     "Create base image + trace + mask from an NCS xcf file",
123     "Create base image + trace + mask from an NCS xcf file",
124     "Luke Wukmer",
125     "Luke Wukmer",
126     "2018",
127     "<Image>/Image/Process_NCS_xcf...",
128     "RGB*, GRAY*",
129     [] ,
130     [] ,
131     process_NCS_xcf)
132
main()

```

listings/scale_sweep_demo.py

```

1 #!/usr/bin/env python3

3 from get_placenta import get_named_placenta, list_placentas, _cropped_bounds, cropped_view,
4     cropped_args, show_mask
5
6 from frangi import frangi_from_image
7
8 import numpy as np
9 import numpy.ma as ma
10 from plate_morphology import dilate_boundary
11
12 import os.path
13 import matplotlib.pyplot as plt

14 #imgfile = list_placentas('T-BN')[32]

15 img = get_named_placenta('T-BN2315363.png')

16 img = dilate_boundary(img, radius=5)
17 F = list()
18 fi = list()

20 #scales = np.logspace(-3,3,base=2,num=8)
21 #scales = np.linspace(.25,8,num=8)
22
23

```

```

scales = np.linspace(.25,4,num=6)
25 for n, sigma in enumerate(scales, 1):
    target = frangi_from_image(img, sigma, dark_bg=False)
27 plate = cropped_view(target).filled(0)
    inset = target[370:660,530:900]
29 F.append(plate)
    fi.append(inset)
31 for label in ['plate','inset']:
    if label == 'inset':
        printable = inset
    else:
        printable = plate
35
37 plt.imshow(printable, cmap=plt.cm.gist_earth)
    plt.title(r'$\sigma={:.2f}$'.format(sigma))
39 plt.tight_layout()
    c = plt.colorbar()
41 c.set_ticks = np.linspace(0,0.6, num=7)
    plt.clim(0,0.6)
43 outname = 'demo_output/scale{sweep_{}}_{}.png'.format(n,label)
    plt.savefig(outname, dpi=300, bbox_inches='tight')
45 print('saved', outname)
    plt.close()
47
# now make a stitched together version
49 for label in ['plate', 'inset']:
    if label == 'inset':
51        L = fi
    else:
53        L = F
    top = np.concatenate(L[:3], axis=1)
55 bottom = np.concatenate(L[3:], axis=1)
    stitched = np.concatenate((top,bottom), axis=0)
57 imga = plt.imshow(stitched, cmap=plt.cm.gist_earth)
    plt.imsave('demo_output/sweep_stitched_{}.png'.format(label),
59         stitched, cmap=plt.cm.gist_earth)
    #plt.colorbar(); plt.clim(0,0.3)

```

listings/vessel_filters.py

```
#!/usr/bin/env python3
2
"""
4 NOTE THIS IS THE OLD CODE (FOR BAD PARAMETER FRANGI).
IT'S HERE FOR LEGACY OR POSSIBLE DEMONSTRATION
6 some function / file names have changed and this code probably won't
work anymore without minor alterations
8
10 import scipy.ndimage as ndi
11 import matplotlib.pyplot as plt
12 from functools import partial
13 from skimage.morphology import *
14 from skimage.exposure import rescale_intensity, equalize_adapthist
15
16 from skimage.color import label2rgb
17
18 import os
19 import os.path
20 import datetime
21
22 import numpy as np
23 import numpy.ma as ma
24
25 from skimage.transform import rotate
26 from skimage.feature import hessian_matrix, hessian_matrix_eigvals
27 from numpy.linalg import eig
28
29 def rotating_box_filter(img, thetas, sigma, length_ratio=4, verbose=True):
30     """
31         runs a curvilinear filter at the given scale space 'sigma'
32
33     INPUT:
34         img:          a binary 2D array
35         sigma:        the scale space
36         length_ratio: a rectangular filter will be applied with size
37         steps:        the range of rotations (0,180) is divided into this
```

```

38             many steps (default: 16 or 12 degrees)
40             verbose:           default True
41
42     OUTPUT:
43
44     METHODS:
45         (todo)
46
47     IMPLEMENTATION:
48         (todo)
49
50     WARNINGS/BUGS:
51
52         this may be supremely wasteful for large step sizes. you should check
53         in the anticipated range of sigmas that there is sufficient variation
54         in the rotated structure elements to warrant that amount of step sizes.
55         print('cleaning up scale space')
56
57         furthermore, this filter should be used carefully. there are probably
58         bugs in the logic and implementation.
59
60     """
61
62     sigma = int(sigma) # round down to a integer
63
64     mask = img.mask
65     extracted = np.zeros_like(img)
66     img = binary_erosion(img, selem=disk(sigma))
67     #img = remove_small_objects(img, min_size=sigma**3)
68     img = binary_dilation(img, selem=disk(sigma))
69
70     width, length = int(2*sigma), int(sigma*length_ratio)
71
72     if length == 0:
73         length = 1
74
75     rect = rectangle(width, length)
76     outer_rect = rectangle(int(width+2*sigma+4),int(length))
77     outer_rect[sigma:-sigma,:] = 0

```

```

    thetas = np.round(thetas*180 / np.pi)

78
# this should behave the same as thetas[thetas==180] = 0
80
# but not return a warning
81 thetas.put(thetas==180, 0) # these angles are redundant
82

83 if verbose:
84     print('running vessel_filter with iC={}: w={}, l={}'.format(
85         sigma, width, length), flush=True)

86
87 if verbose:
88     print('building rotated filters...', end=' ')
89
90 srot = partial(rotate, resize=True, preserve_range=True) # look at order
91 rotated = [srot(rect, theta) for theta in range(180)]
92
93 if verbose:
94     print('done.')
95
96 if verbose:
97     print('building outer filters...', end=' ')
98 outer_rotated = [srot(outer_rect, theta) for theta in range(180)]
99
100 for theta in range(180):
101     if verbose:
102         print('i= ', theta, end='\t', flush=True)
103         if theta % 6 == 0:
104             print()
105
106         vessels = binary_erosion(img, selem=rotated[theta])
107         #margins = binary_dilation(img, selem=outer_rotated[theta])
108         #margins = np.invert(margins)
109         #vessels = np.logical_and(vessels, margins)
110         extracted = np.logical_or(extracted, (thetas == theta) * vessels)
111     if verbose:
112         print() # new line
113
114     extracted = binary_dilation(extracted, selem=disk(sigma))
115     extracted[mask] = 0

```

```

116     return extracted

118 def get_frangi_targets(K1,K2, beta=0.5, c=15, dark_bg=True, threshold=None):
119     """
120     returns results of frangi filter
121     """
122
123     R = (K1/K2) ** 2 # anisotropy
124     S = (K1**2 + K2**2) # structureness

125
126     F = np.exp(-R / (2*beta**2))
127     F *= 1 - np.exp( -S / (2*c**2))

128
129     if dark_bg:
130         F = (K2 < 0)*F
131     else:
132         F = (K2 > 0)*F

133
134     if threshold:
135
136         return F < threshold
137     else:
138
139         return F

140 def get_targets(K1,K2, method='F', threshold=True):
141     """
142     returns a binary threshold (conservative)
143
144     F -> frangi filter with default arguments. greater than mean.
145     R -> blobness measure. greater than median.
146     S -> anisotropy measure (greater than median)
147     """
148
149     if method == 'R':
150         R = (K1 / K2) ** 2
151
152         if threshold:
153             T = R < ma.median(R)
154         else:
155             T = R
156
157     elif method == 'S':

```

```

    S = (K1**2 + K2**2)/2
156
    if threshold:
        T = S > ma.median(S)
158
    else:
        T = S
160
    elif method == 'F':
        R = (K1 / K2) ** 2
162
        S = (K1**2 + K2**2)/2
        beta, c = 0.5, 15
164
        F = np.exp(-R / (2*beta**2))
        F *= 1 - np.exp(-S / (2*c**2))
166
        T = (K2 < 0)*F

168
    if threshold:
        T = T > (T[T != 0]).mean()
170
    else:
        raise('Need to select method as "F", "S", or "R"')
172
    return T
174
b = partial(plt.imshow, cmap=plt.cm.Blues)
176 s = plt.show

178
if __name__ == "__main__":
180
#raw = get_preprocessed(mode='G')
182 raw = ndi.imread('samples/clahé_raw.png')
    raw = preregister(raw)
184 img = preprocess(raw)
    img = raw
186
# which īC to use (in order)
188 scale_range = np.logspace(0,5, num=30, base=2)

190 frangi_only = np.zeros((img.shape[0],img.shape[1],len(scale_range)))
192 all_targets = np.zeros((img.shape[0],img.shape[1],len(scale_range)))
    extracted_all = np.zeros((img.shape[0],img.shape[1],len(scale_range)))

```

```

194     OUTPUT_DIR = 'fpd_new_output'
196
198     n = datetime.datetime.now()
200
202     SUBDIR = ''.join((_, n.strftime('%y%m%d_%H%M')))

204     print('saving outputs in', os.path.join(OUTPUT_DIR, SUBDIR))

206     try:
208         os.mkdir(os.path.join(OUTPUT_DIR, SUBDIR))
210
212     except FileExistsError:
214         ans = input('save path already exists! would you like to continue? [y/N]')
216         if ans != 'y':
218             print('aborting program. clean up after yourself.')
220
222         exit(0)
224
226     else:
228         print('your files will be overwritten (but artifacts may remain!)')
230
232     finally:
234         print('\n')

236     for n, sigma in enumerate(scale_range):
238
240         beta = min(.09*sigma - .04, .5)
242         print('-' * 80)
244         print('I={}'.format(sigma))

246
248         print('finding hessian')
250         h = fft_hessian(img, sigma)

252
254         print('finding curvatures')
256         k1, k2 = principal_curvatures(img, sigma=sigma, H=h)

258
260         print('finding targets with I={}'.format(beta))
262         t = get_frangi_targets(k1, k2,
264             beta=beta, dark_bg=False, threshold=False)
266         t = t > t.mean()

```

```

#t = remove_small_objects(t, min_size=100)

234
# extend mask
236 timg = ma.masked_where(t < t.mean(), img)
percentage = timg.count() / img.size
238
238 print('finding p. directions {}'.format(np.round(percentage*100)))
240 t1,t2 = principal_directions(timg, sigma=sigma, H=h)

242 extracted = vessel_filter(t, t1, sigma, length_ratio=.5, verbose=True)

244 extracted_all[:, :, n] = extracted

246 savefile = ''.join(( '%02d' % sigma, '.png'))
plt.imsave(os.path.join(OUTPUT_DIR, SUBDIR, savefile),
248 extracted, cmap=plt.cm.Blues)

250 all_targets[:, :, n] = (timg!=0) * t1

252 #all_targets[:, :, n]= timg!=0

254 A = all_targets.sum(axis=-1)
a = all_targets
256
256 sys.exit(0)

258 #new_labels = sigma*np.logical_and(extracted != 0, cumulative == 0)
260 #cumulative += new_labels.astype('uint8')

262 full_skel = skeletonize(cumulative!=0)
skel = remove_small_objects(full_skel, min_size=50, connectivity=2)
264
264 matched_all = np.zeros_like(skel)
266
266 for i, scale in enumerate(scale_range):
268
e = extracted_all[:, :, i]
270 el, nl = label(e, return_num=True)
matched = np.zeros_like(matched_all)

```

```

272     for region in range(1, nl+1):
273         if np.logical_and(el==region, skel).any():
274             matched = np.logical_or(matched, el==region)
275
276     matched_all = np.logical_or(matched_all, matched)

```

listings/hfft_accuracy.py

```

1 """
3 here you want to show the accuracy of hfft.py
5 BOILERPLATE
7 show that gaussian blur of hfft is accurate, except potentially around the
boundary proportional to sigma.
9
10 or if they're off by a scaling factor, show that the derivates
11 (taken the same way) are proportional.
13 pseudocode
15 A = gaussian_blur(image, sigma, method='conventional')
16 B = gaussian_blue(image, sigma, method='fourier')
17
18 zero_order_accurate = isclose(A, B, tol)
19
20 J_A= get_jacobian(A)
21 J_B = get_jacobian(B)
22
23 first_order_accurate = isclose(J_A, J_B, tol)
24
25 A_eroded = zero_around_plate(A, sigma)
26 B_eroded = zero_around_plate(B, sigma)
27
28 J_A_eroded = zero_around_plate(A, sigma)
29 J_B_eroded = zero_around_plate(B, sigma)

```

```

31 zero_order_accurate_no_boundary = isclose(A_eroded, B_eroded, tol)
32 first_order_accurate = isclose(J_A_eroded, J_B_eroded, tol)
33
34 """
35
36     from get_placenta import get_named_placenta
37
38     from hfft import fft_hessian, fft_gaussian
39     from scipy.ndimage import gaussian_filter
40     import matplotlib.pyplot as plt
41     from get_placenta import mimshow
42
43     from score import mean_squared_error
44     import numpy as np
45     from scipy.ndimage import laplace
46     import numpy.ma as ma
47
48     from skimage.segmentation import find_boundaries
49     from skimage.morphology import disk, binary_dilation
50
51     from diffgeo import principal_curvatures
52     from frangi import structureness, anisotropy, get_frangi_targets
53
54     def erode_plate(img, sigma, mask=None):
55         """
56             Apply an eroded mask to an image
57             assume (if helpful) that the boundary of the placenta is a connected loop
58             that is, there is a single inside and outside of the shape, and that
59             the placenta is more or less convex
60
61             alternatively, if img is None, simply erode the mask
62             this function should probably be renamed "dilate mask"
63             and erode plate should be one that just acts on masked inputs
64         """
65
66         if mask is None:
67             mask = img.mask
68
69         # get a boolean array that is 1 along the border of the mask, zero elsewhere

```

```

# default mode is 'thick' which is fine
71 bounds = find_boundaries(mask)

73 # structure element to dilate by is a disk of diameter sigma
74 # rounded up to the nearest integer. this may be too conservative.
75 selem = disk(np.ceil(sigma))
76 dilated_border = binary_dilation(bounds, selem=selem)

77 new_mask = np.logical_or(mask, dilated_border)
78

79 # see comment in docstring. alternatively, the behavior here
80 # could be handled by an "apply mask" parameter
81 if img is None:
82     return new_mask
83 else:
84     return ma.masked_array(img, mask=new_mask)

87
88 # FIX SOME ISSUES, BINARY DILATION IS TAKING HELLA LONG AND ALSO
89 # THERE ARE RANDOM BLIPS INSIDE THE MASK!!!
90 # FIX IN GIMP!:

91
92 imgfile = 'barium1.png'
93 maskfile = 'barium1.mask.png'

94 img_raw = get_named_placenta(imgfile, maskfile=maskfile)

95 # so that scipy.ndimage.gaussian_filter doesn't use uint8 precision (jesus)
96 img = img_raw / 255.

97
98 # convenience function to show a matrix with img.mask mask
99 ms = lambda x: mimshow(ma.masked_array(x, img.mask))

100
101 sigma = 5

102 print('applying standard gauss blur')
103 # THIS USES THE SAME DTYPE AS THE INPUT SO DEAR LORD MAKE SURE IT'S A FLOAT
104 A = gaussian_filter(img.astype('f'), sigma, mode='constant') #zero padding
105 print('applying fft gauss blur')

```

```

109 B = fft_gaussian(img, sigma)
110 B_unnormalized = B.copy()
111 B = B / (2*(sigma**2)*np.pi)

113 #A = erode_plate(A, sigma, mask=img.mask)
114 #B = erode_plate(B, sigma, mask=img.mask)
115 print('calculating first derivatives')

117 # zero the masks before calculating derivates if they're masked
118 Ax, Ay = np.gradient(A)
119 Bx, By = np.gradient(B)

121
122 print('calculating second derivatives')

123
124 # you can verify np.isclose(Axy,Ayx) && np.isclose(Bxy,Byx) -> True
125 Axx, Axy = np.gradient(Ax)
126 Ayx, Ayy = np.gradient(Ay)
127
128 Bxx, Bxy = np.gradient(Bx)
129 Byx, Byy = np.gradient(By)

131

133 print('calculating eigenvalues of hessian')
134 ak1, ak2 = principal_curvatures(A, sigma=sigma, H=(Axx,Axy,Ayy))
135 bk1, bk2 = principal_curvatures(B, sigma=sigma, H=(Bxx,Bxy,Byy))

137
138 ##R1 = anisotropy(ak1,ak2)
139 ##R2 = anisotropy(bk1,bk2)
140 #
141 #S1 = structureness(ak1, ak2)
142 #S2 = structureness(bk1, bk2)
143 #print('done.')
144 #
145 ## ugh, apply masks here. too large to be conservative?
146 ## otherwise structureness only shows up for small sizes
147 new_mask = erode_plate(None, 3*sigma, mask=img.mask)

```

```

#R1[new_mask] = 0
149 #R2[new_mask] = 0
#S1[new_mask] = 0
151 #S2[new_mask] = 0

153 FA = get_frangi_targets(ak1,ak2)
FB = get_frangi_targets(bk1,bk2)

155
FA[new_mask] = 0
157 FB[new_mask] = 0

159 # even without scaling (which occurs below) the second derivates should be
# close. normalize matrices using frobenius norm of the hessian?
161 # note: A & B are off but have the same shape

163
# rescale to [0,255] (actually should keep as 0,1? )
165 #A_unscaled = A.copy()
#B_unscaled = B.copy()

167
#Ascaled = (A-A.min())/(A.max()-A.min())
169 #Bscaled = (B-B.min())/(B.max()-B.min())

171 # the following shows a random vertical slice of A & B (when scaled)
# the results are even more fitting when you scale B to coincide with A's max
173 # (which obviously isn't feasible in practice)

175 # FIXEDISH AFTER SCALING!

177 plt.plot(np.arange(A.shape[1]),A[A.shape[0]//2,:],
           label='scipy.ndimage,gaussian_filter')
179 plt.plot(np.arange(B.shape[1]), B[B.shape[0]//2,:],
           label='fft_gaussian')
181 plt.legend()

183 #MSE = ((A-B)**2).sum() / A.size
MSE = mean_squared_error(A,B)

```

BIBLIOGRAPHY

BIBLIOGRAPHY

[1] J. Babaud, M. Baudin, R. O. Duda, and A. P. Witkin. Uniqueness of the gaussian kernel for scale-space filtering. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 8:26–33, 01 1986.

[2] Alejandro F Frangi, Wiro J Niessen, Koen L Vincken, and Max A Viergever. Multiscale vessel enhancement filtering. In *International Conference on Medical Image Computing and Computer-Assisted Intervention*, pages 130–137. Springer, 1998.

Starting point.

[3] Rafael C Gonzalez and Richard E Woods. Digital image processing prentice hall. *Upper Saddle River, NJ*, 2002.

Check this citation, find a better one.

[4] E. Hille and R.S. Phillips. *Functional Analysis and Semi-groups*. American Mathematical Society: Colloquium publications. American Mathematical Society, 1957.

cited within Sporring just for one thing

[5] Nen Huynh. *A filter bank approach to automate vessel extraction with applications*. PhD thesis, California State University, Long Beach, 2013.

[6] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed `today`].

- [7] Jan J. Koenderink. The structure of images. *Biological Cybernetics*, 50(5):363–370, Aug 1984.
- [8] W. Kühnel, B. Hunt, and American Mathematical Society. *Differential Geometry: Curves - Surfaces - Manifolds*. Student mathematical library. American Mathematical Society, 2006.

Could use the 3rd edition as well.

- [9] C. Lorenz, I. C. Carlsen, T. M. Buzug, C. Fassnacht, and J. Weese. Multi-scale line segmentation with automatic estimation of width, contrast and tangential direction in 2d and 3d medical images. In Jocelyne Troccaz, Eric Grimson, and Ralph Mösges, editors, *CVRMed-MRCAS'97*, pages 233–242, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.
- [10] B.W. Matthews. Comparison of the predicted and observed secondary structure of t4 phage lysozyme. *Biochimica et Biophysica Acta (BBA) - Protein Structure*, 405(2):442–451, 1975.
- [11] Sílvia Delgado Olabarriaga, M Breeuwer, and WJ Niessen. Evaluation of hessian-based filters to enhance the axis of coronary arteries in ct images. In *International Congress Series*, volume 1256, pages 1191–1196. Elsevier, 2003.

Comparison of Hessian-based vesselness enhancement filters (which they call HBVF). Introduction is very good. Also note, they use “exponentially distributed” scales for each filter! I don’t necessarily agree with their results though, and I don’t see what images they’re using. They do some parameter searching though, if I can understand their figures it might be useful to do myself in the thesis.

- [12] Yoshinobu Sato, Shin Nakajima, Nobuyuki Shiraga, Hideki Atsumi, Shigeyuki Yoshida, Thomas Koller, Guido Gerig, and Ron Kikinis. Three-dimensional multi-scale line filter for segmentation and visualization of curvilinear structures in medical images. *Medical image analysis*, 2(2):143–168, 1998.

This was cited in Frangi I believe as a different Hessian filter. Look into this!

Gets compared in Olabarriaga, above. I am ignorantly assuming that this is strictly inferior to Frangi's filter. Worth looking into.

- [13] Jon Sporring. *Gaussian Scale-Space Theory*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

- [14] Stéfan van der Walt, Johannes L. Schönberger, Juan Nunez-Iglesias, François Boulogne, Joshua D. Warner, Neil Yager, Emmanuelle Gouillart, Tony Yu, and the scikit-image contributors. scikit-image: image processing in Python. *PeerJ*, 2:e453, 6 2014.